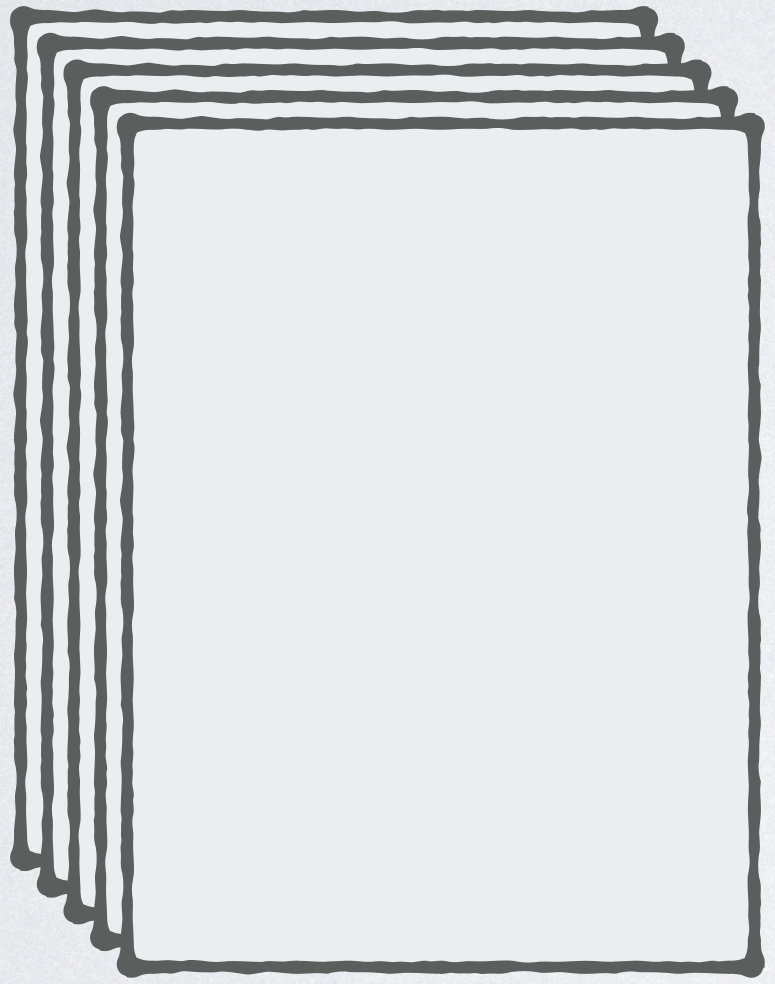
Type-Based Resource-Guided Search

Di Wang
Carnegie Mellon University

About Me

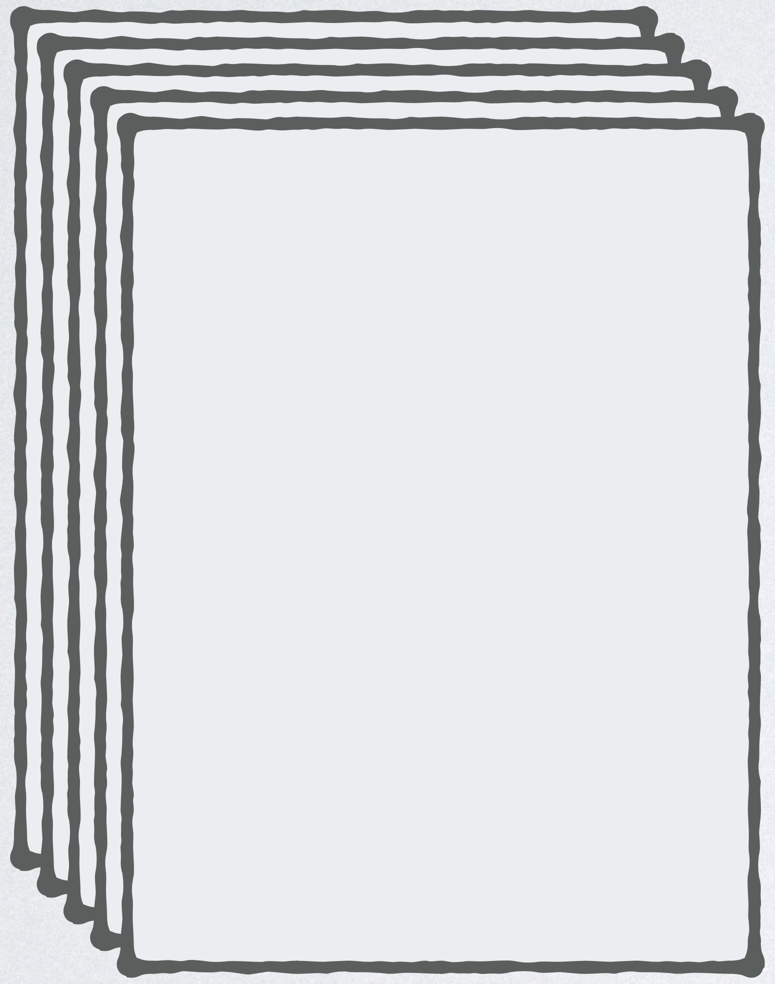
- I am a doctoral student at Carnegie Mellon University.
- I am interested in programming languages and software engineering.
- My focuses are **probabilistic programming** and **static resource analysis**.

Resource Analysis



Programs

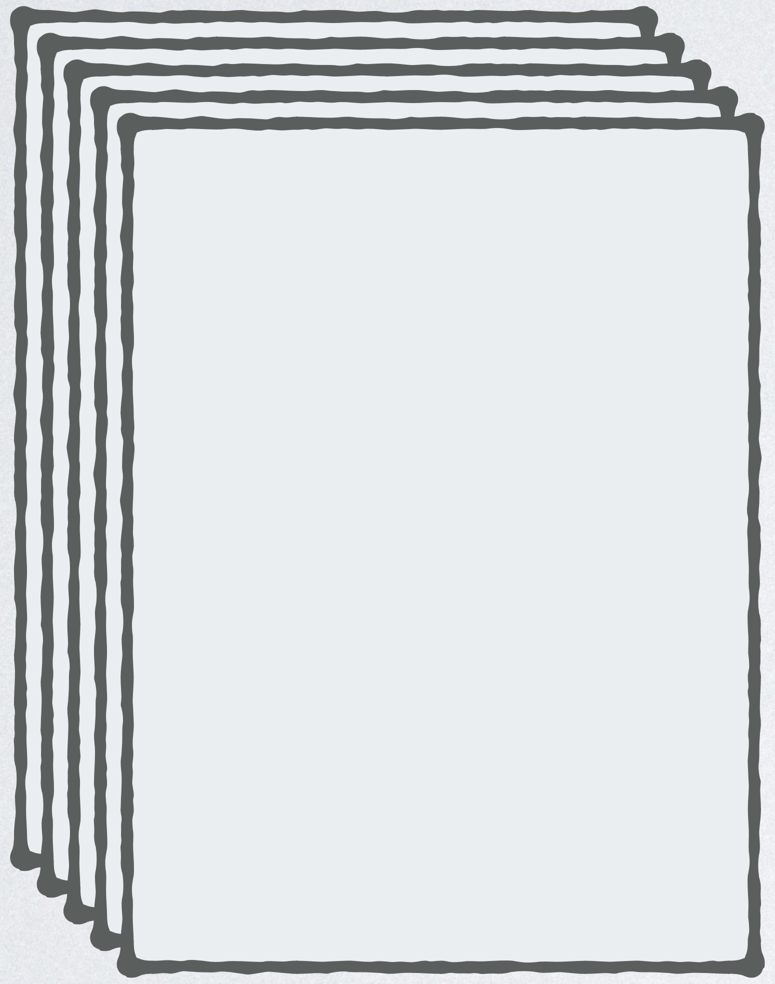
RESOURCE ANALYSIS



Programs

Performance

Resource Analysis

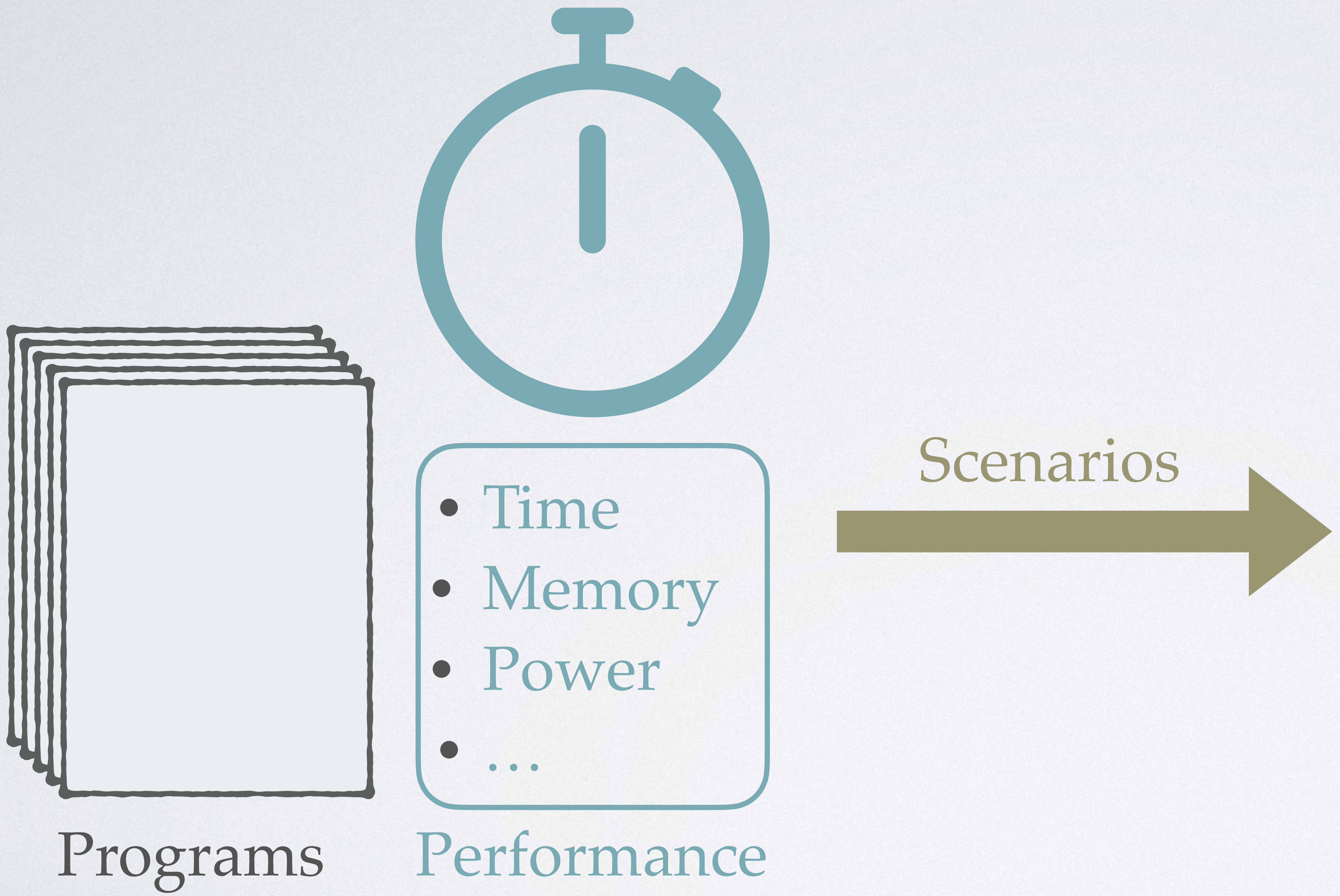


- Time
- Memory
- Power
- …

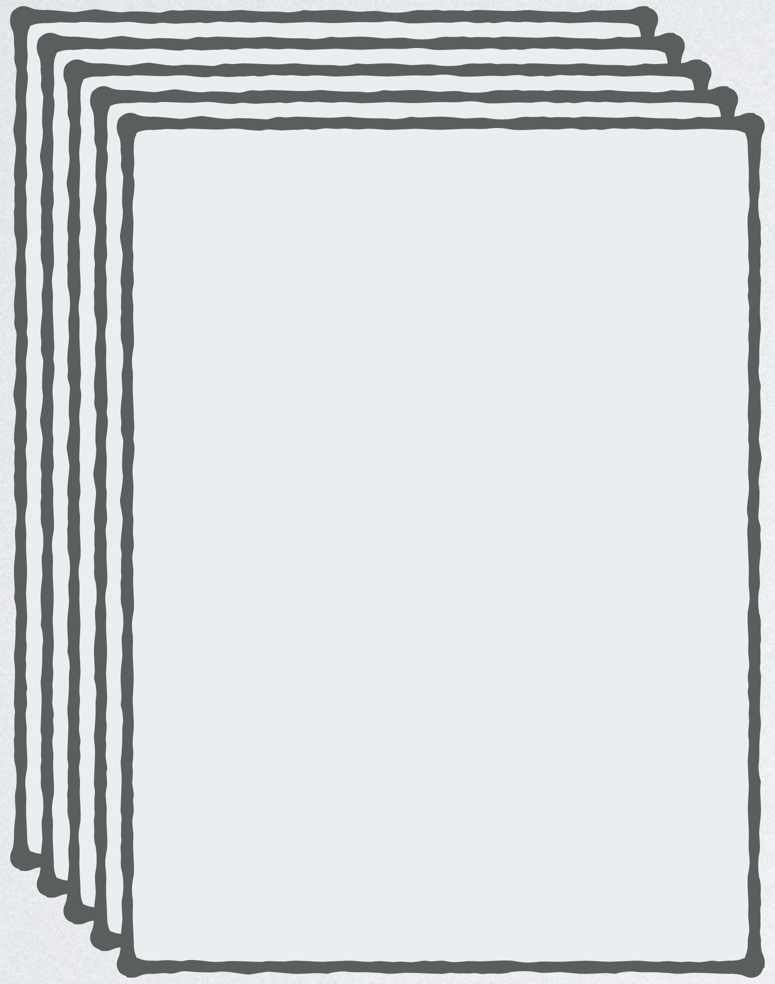
Programs

Performance

Resource Analysis



Resource Analysis



Programs

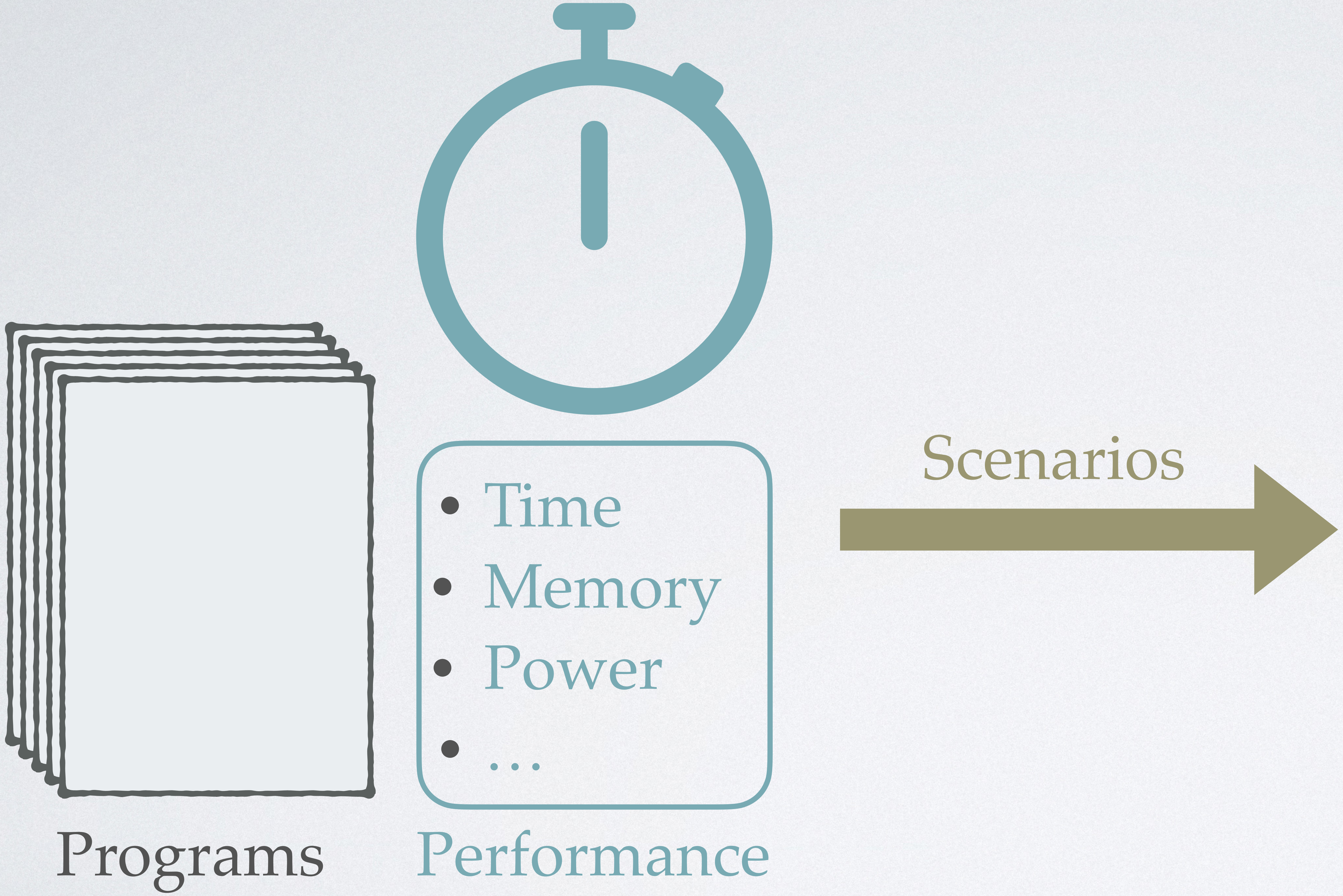
- Time
- Memory
- Power
- …

Performance

Scenarios

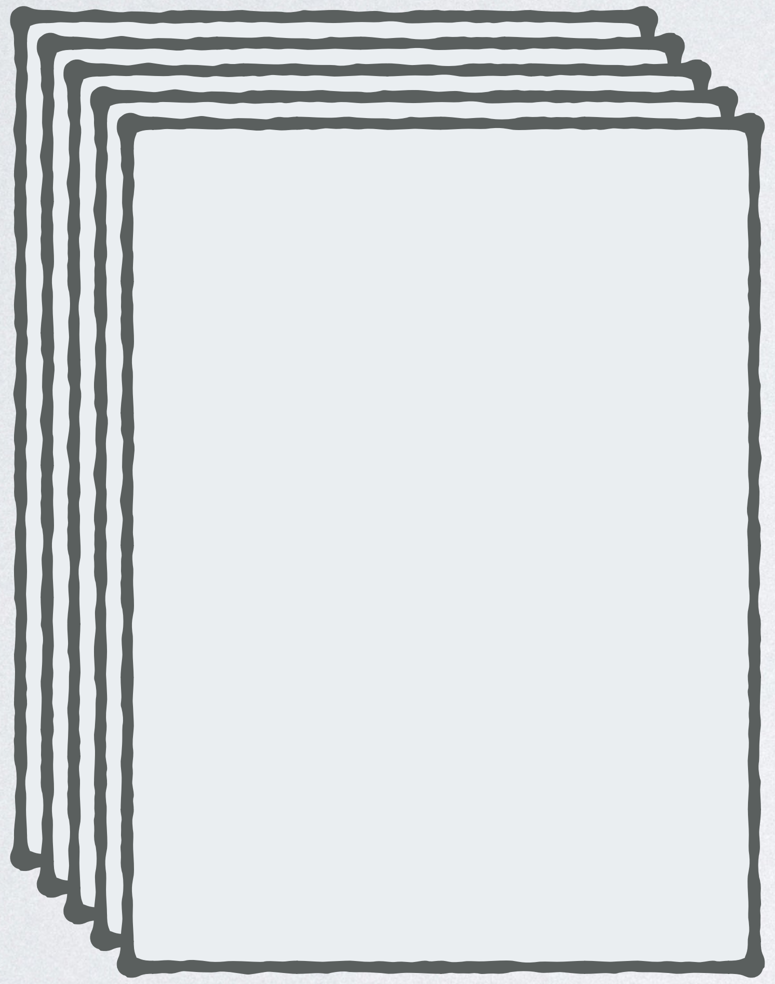
- Identifying bottlenecks

Resource Analysis



- Identifying bottlenecks
- Timing side channels

RESOURCE ANALYSIS



- Time
- Memory
- Power
- …

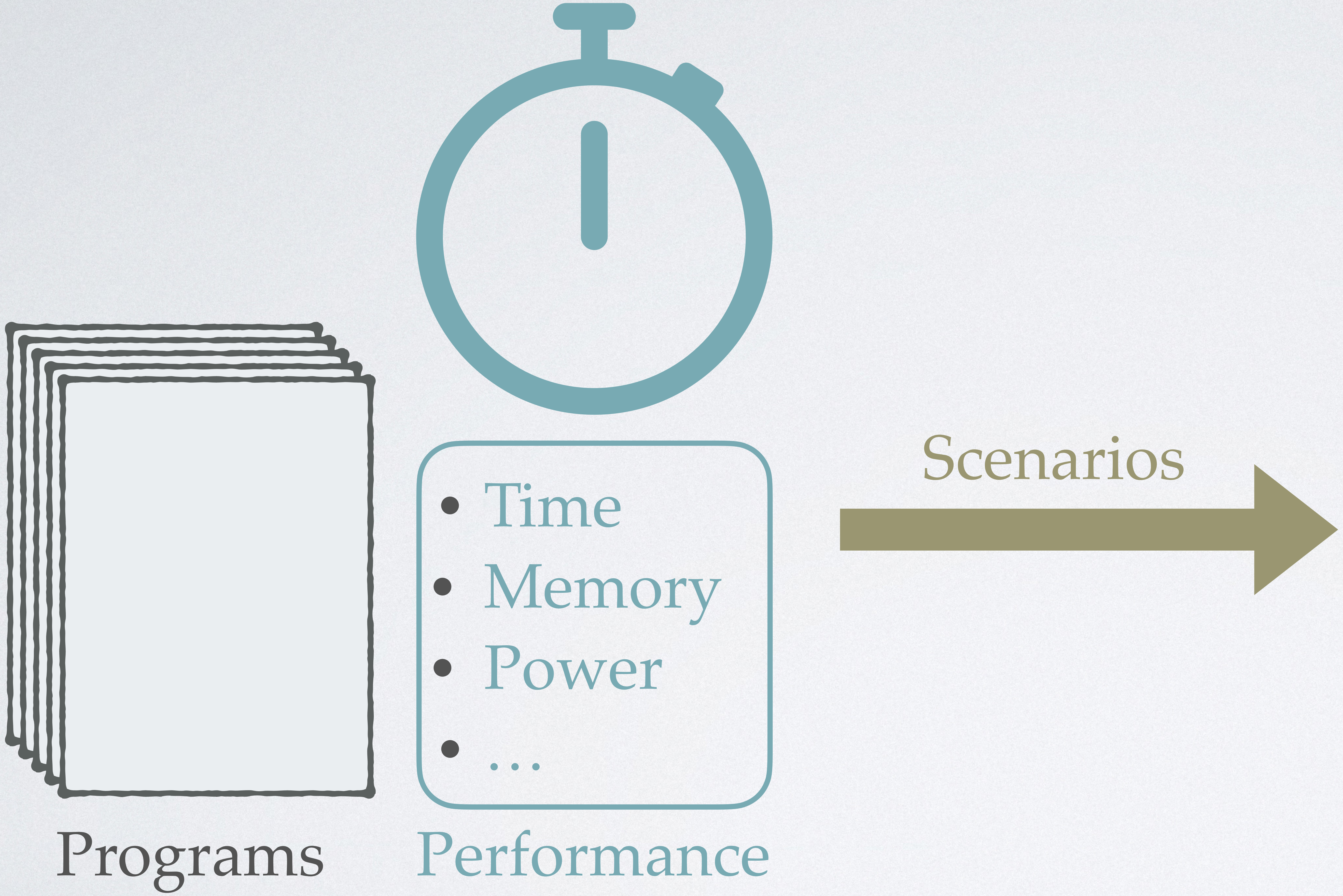
Programs

Performance

Scenarios →

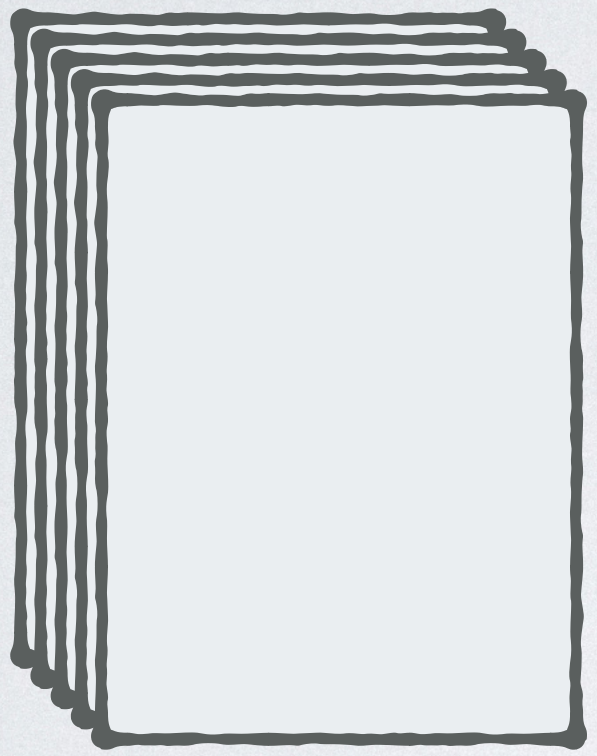
- Identifying bottlenecks
- Timing side channels
- Gas usage in blockchains

RESOURCE ANALYSIS



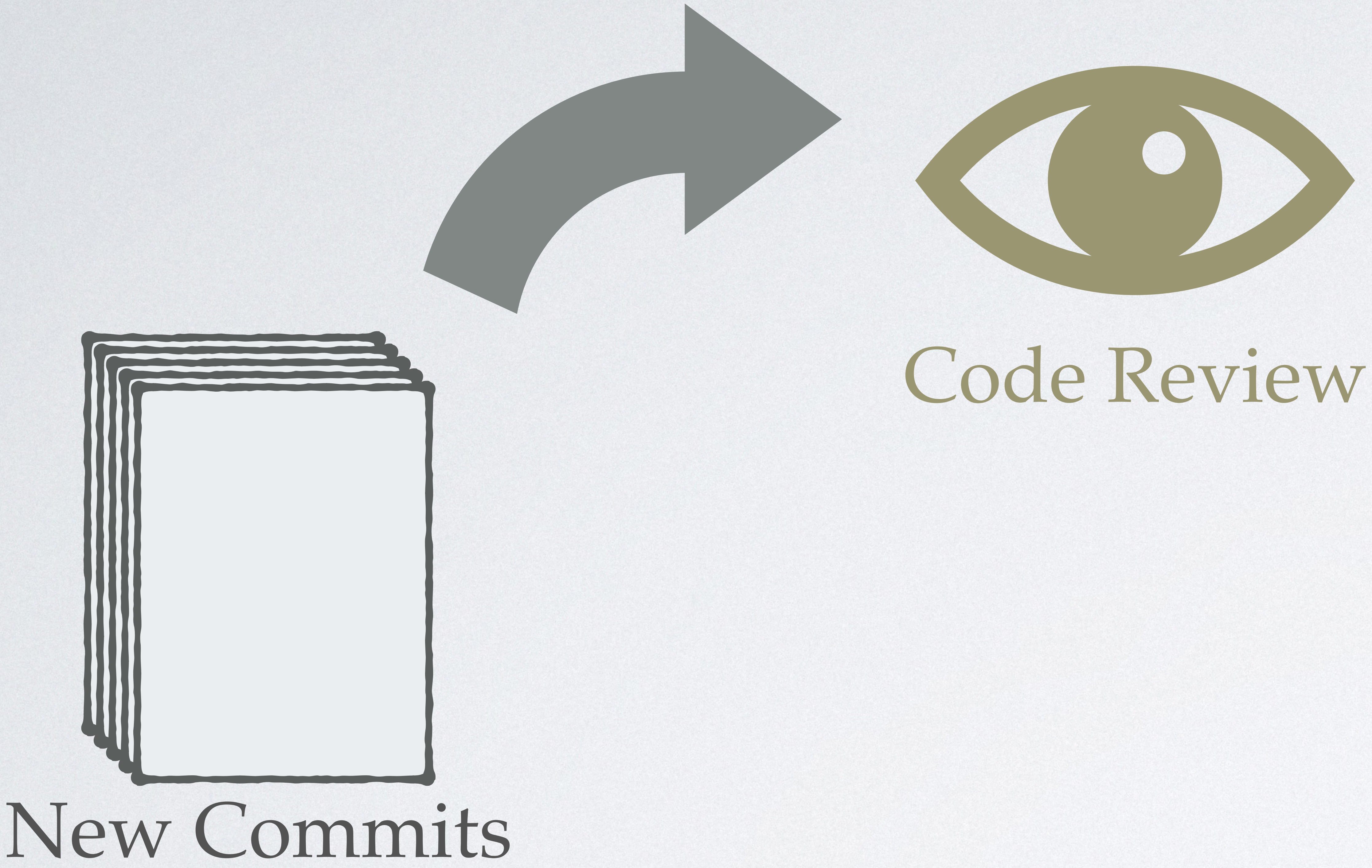
- Identifying bottlenecks
- Timing side channels
- Gas usage in blockchains
- Carbon footprint

STATIC RESOURCE ANALYSIS

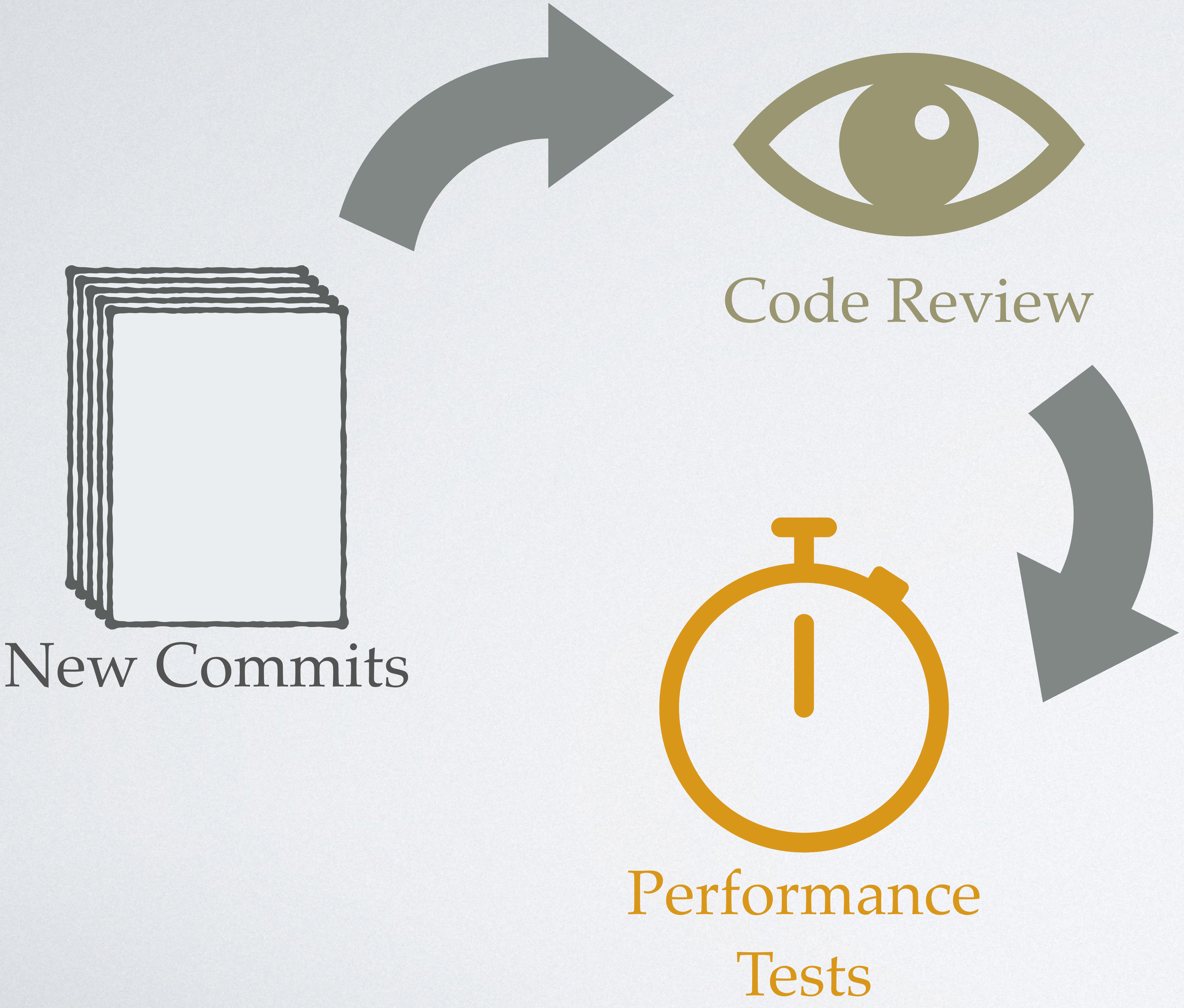


New Commits

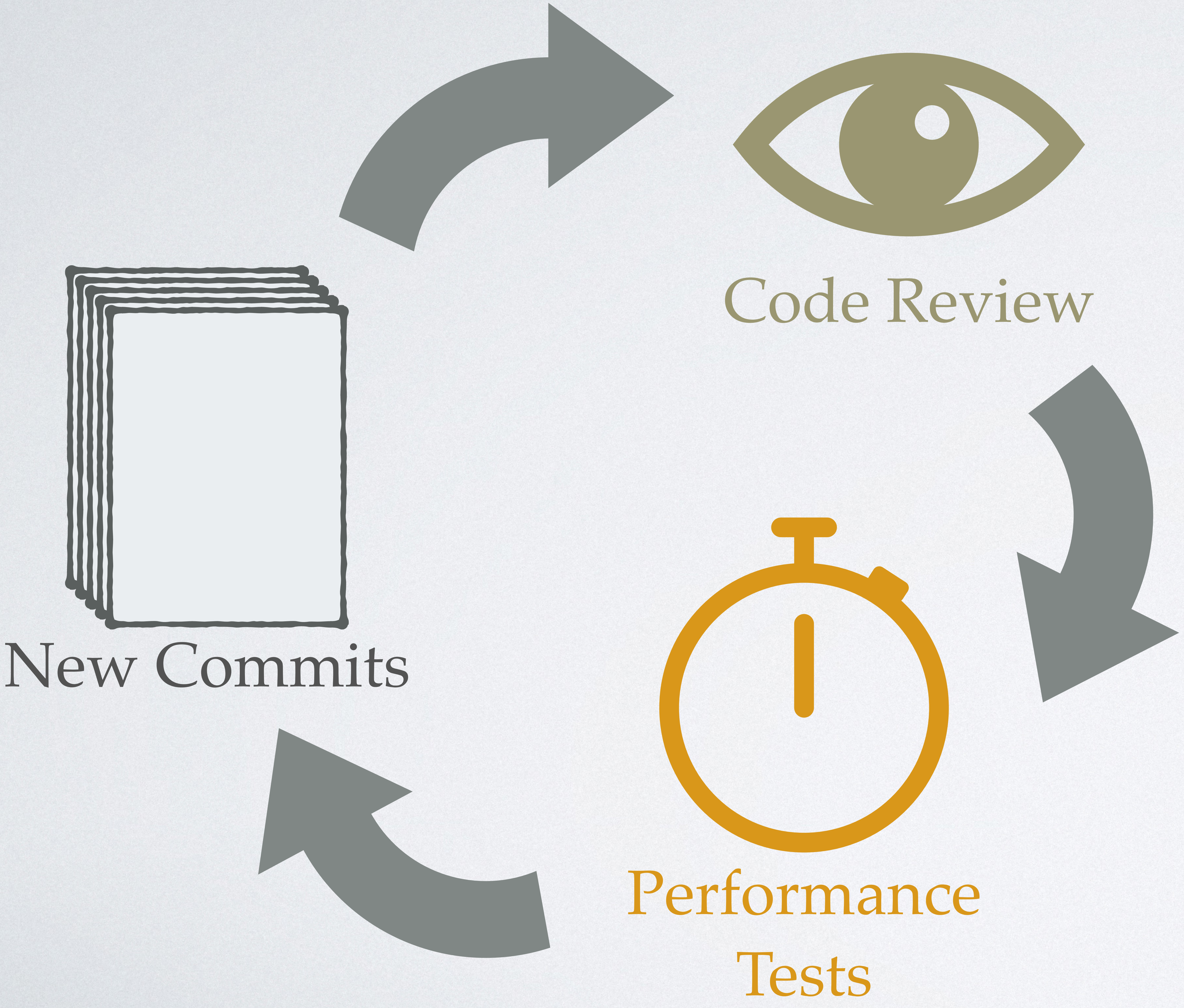
Static Resource Analysis



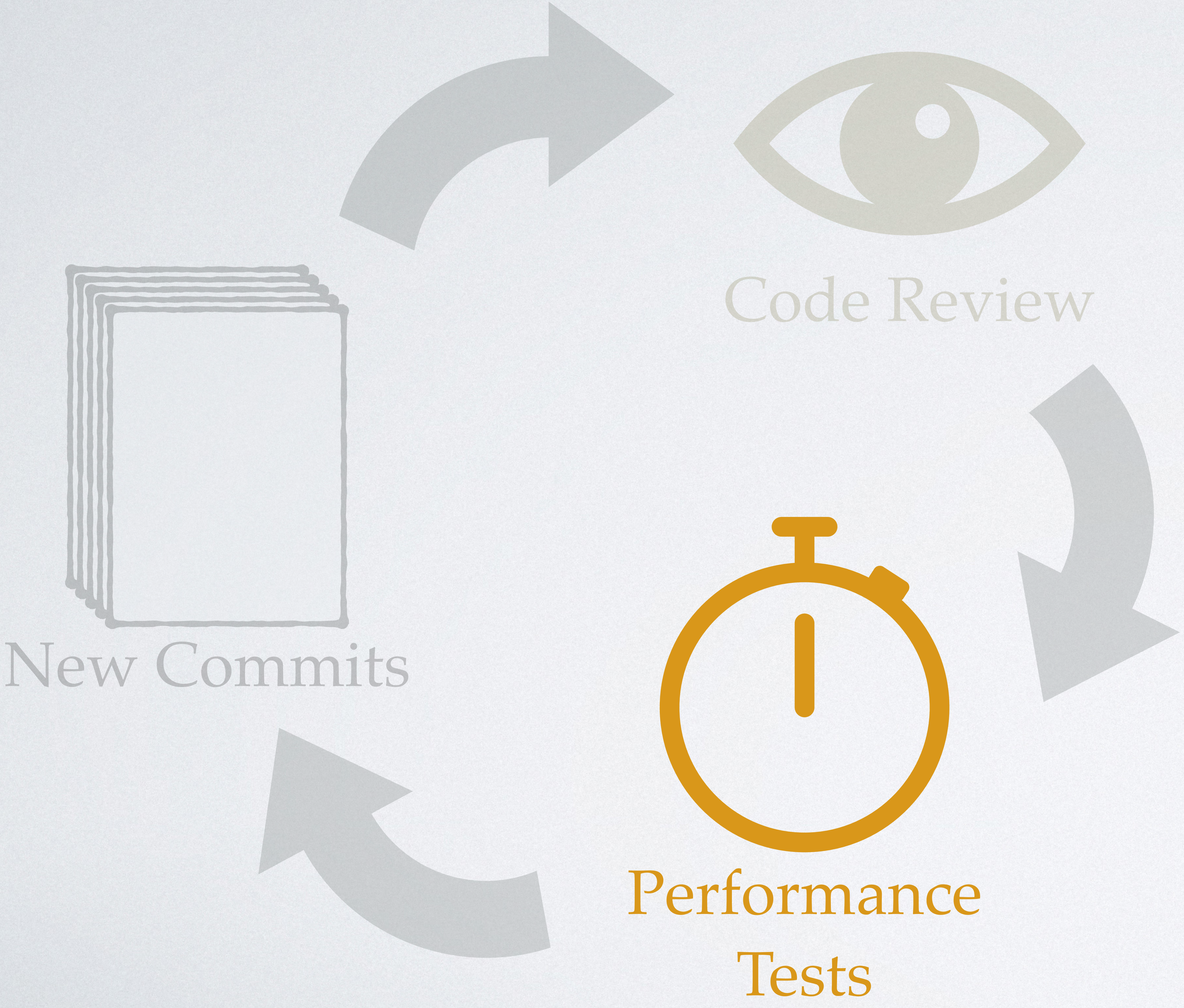
STATIC RESOURCE ANALYSIS



STATIC RESOURCE ANALYSIS



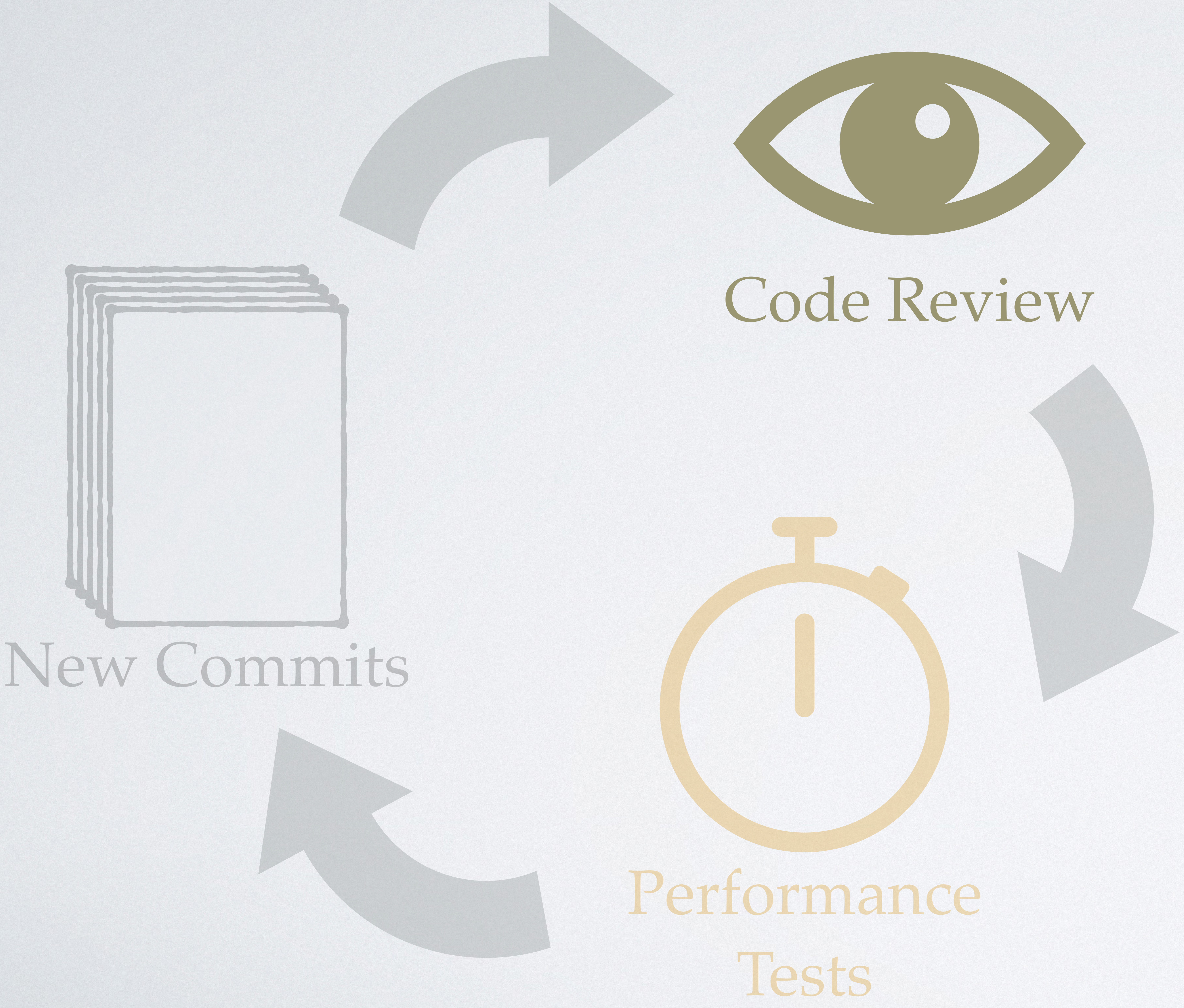
Static Resource Analysis



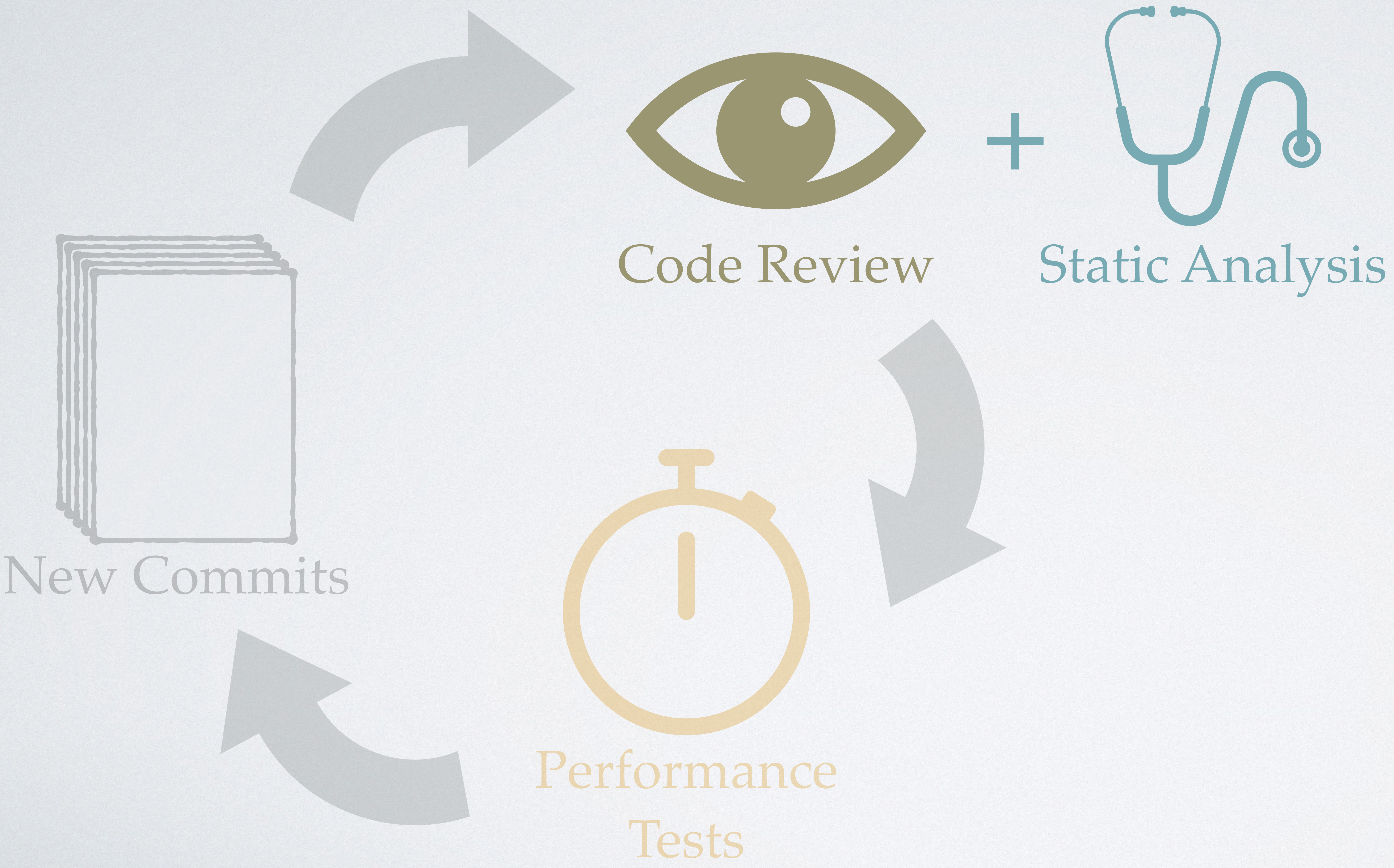
Possible drawbacks:

- Incomplete test coverage
- Time-consuming

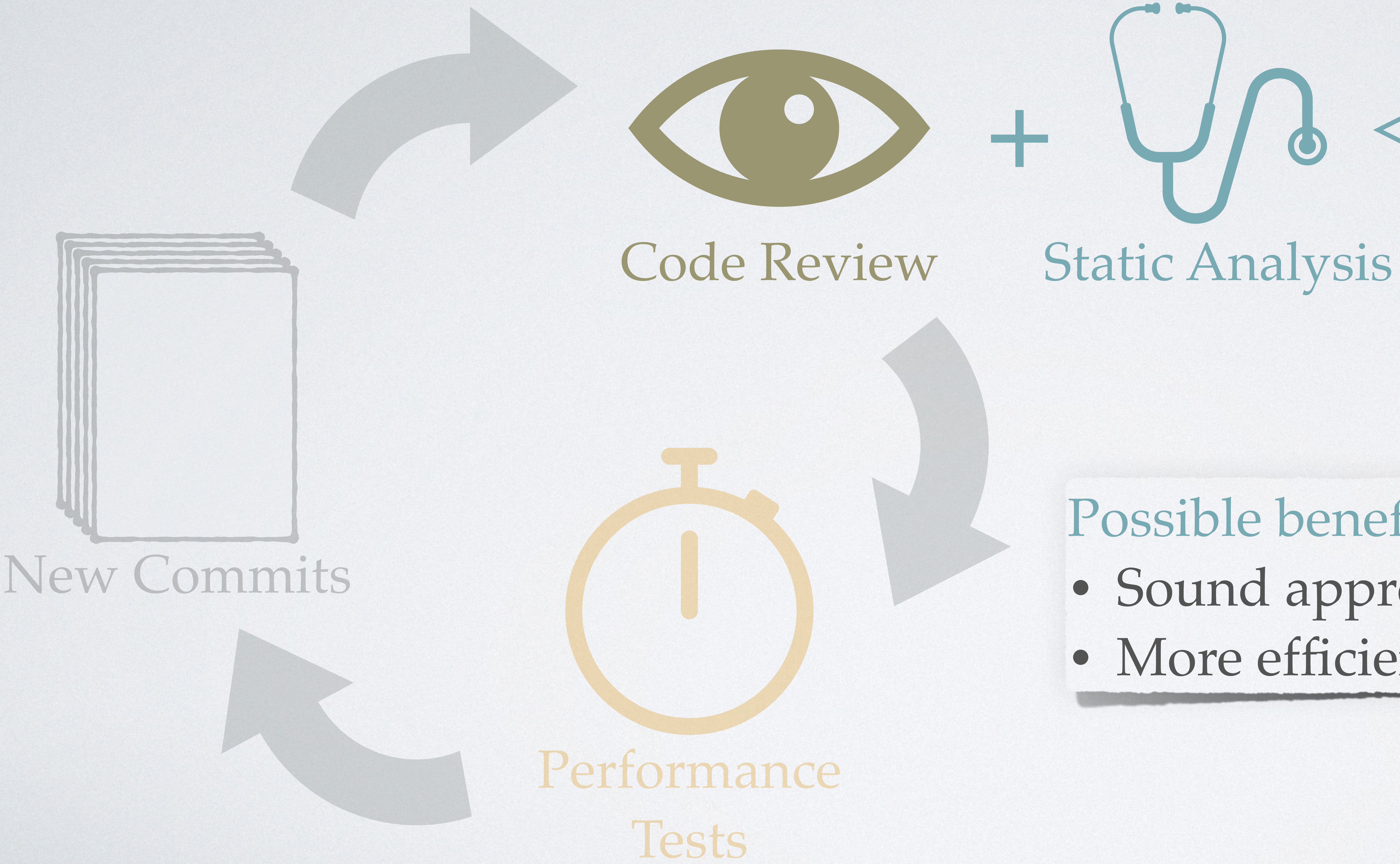
STATIC RESOURCE ANALYSIS



Static Resource Analysis



STATIC RESOURCE ANALYSIS



Analyze resource usage at compile time!

Possible benefits:

- Sound approximations for all inputs
- More efficient if analysis is incremental

Static Resource Analysis in Infer

Examples come from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

Examples come from Infer’s documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

STATIC RESOURCE ANALYSIS IN INFER

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

Examples come from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

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

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

Examples come from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

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

Examples come from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

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

$O(|list|^2)$

Examples come from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

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

$$8\,|list| + 16 = O(|list|) \xrightarrow{\text{Complexity increase!}} O(|list|^2)$$

Examples come from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in RaML

RaML

[RaML17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

STATIC RESOURCE ANALYSIS IN RAML

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

RaML

[RaML17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

STATIC RESOURCE ANALYSIS IN RAML

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

RaML

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

[RaML17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

STATIC RESOURCE ANALYSIS IN RAML

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

RaML

append : resource-annotated type

[RaML17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

Static Resource Analysis in RaML

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

RaML

append : resource-annotated type

Simplified bound:

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

[RaML17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

Static Resource Analysis in RaML

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

RaML

append : resource-annotated type

Simplified bound:

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

This Talk: Type-Based Automatic Amortized Resource Analysis

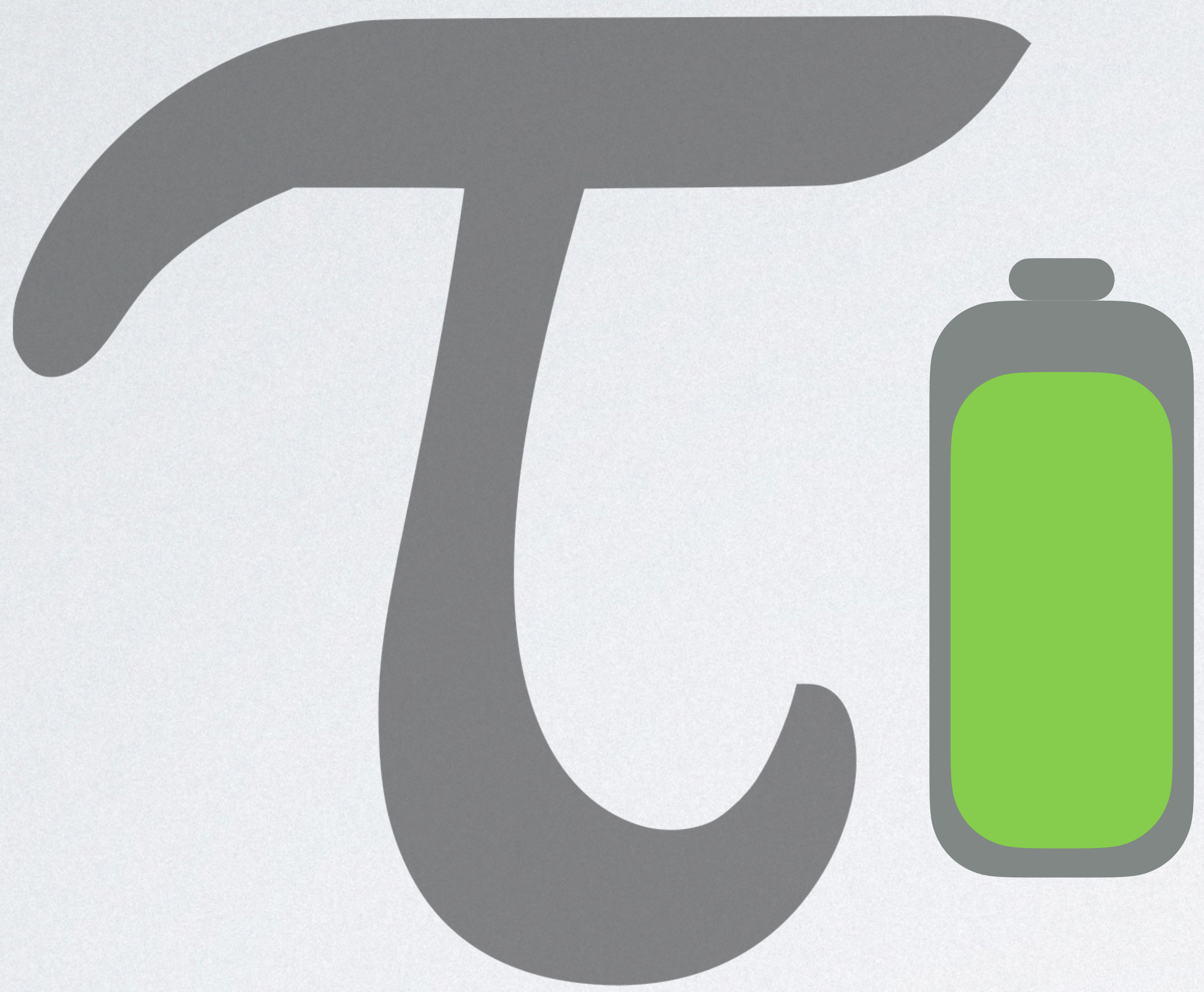
[RaML17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

OUTLINE

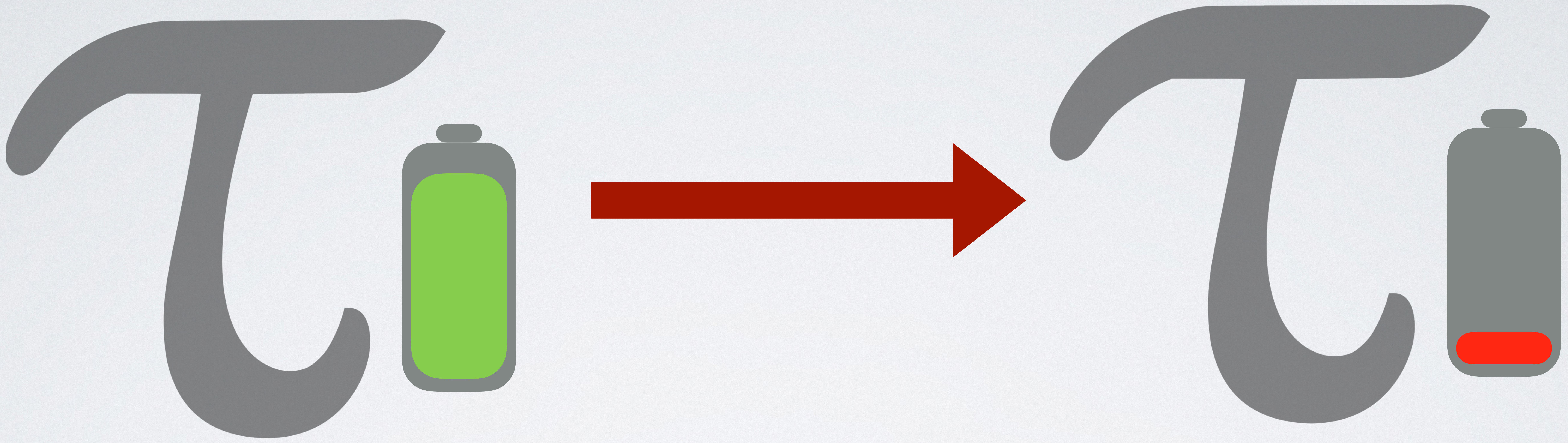
- Automatic Amortized Resource Analysis
- Type-Guided Worst-Case Input Generation
- Resource-Guided Program Synthesis

AUTOMATIC AMORTIZED RESOURCE ANALYSIS

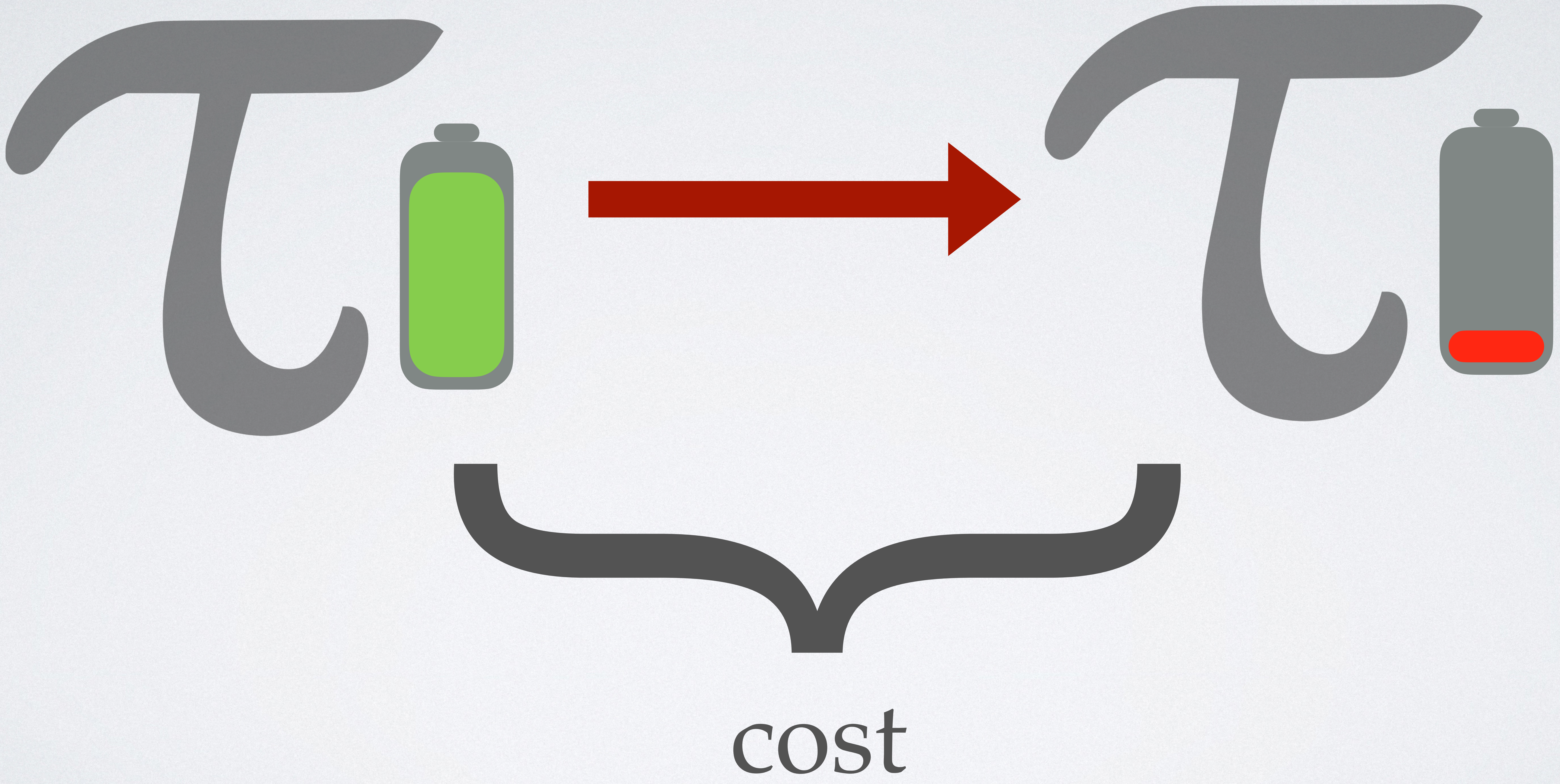
AUTOMATIC AMORTIZED RESOURCE ANALYSIS



Automatic Amortized Resource Analysis

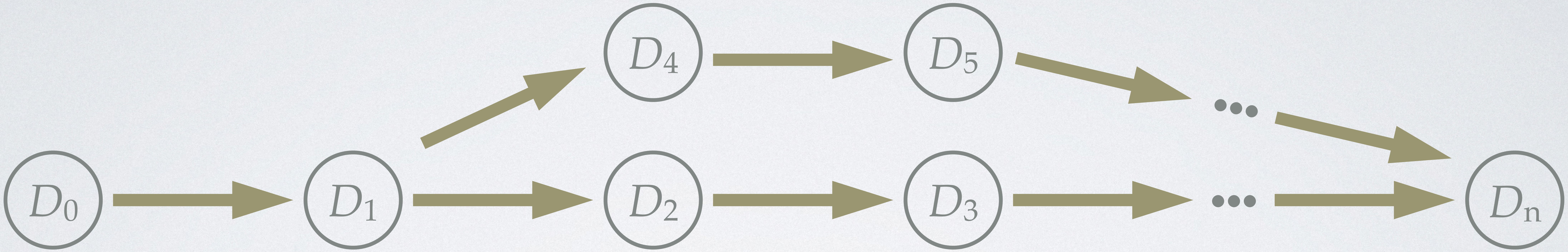


Automatic Amortized Resource Analysis

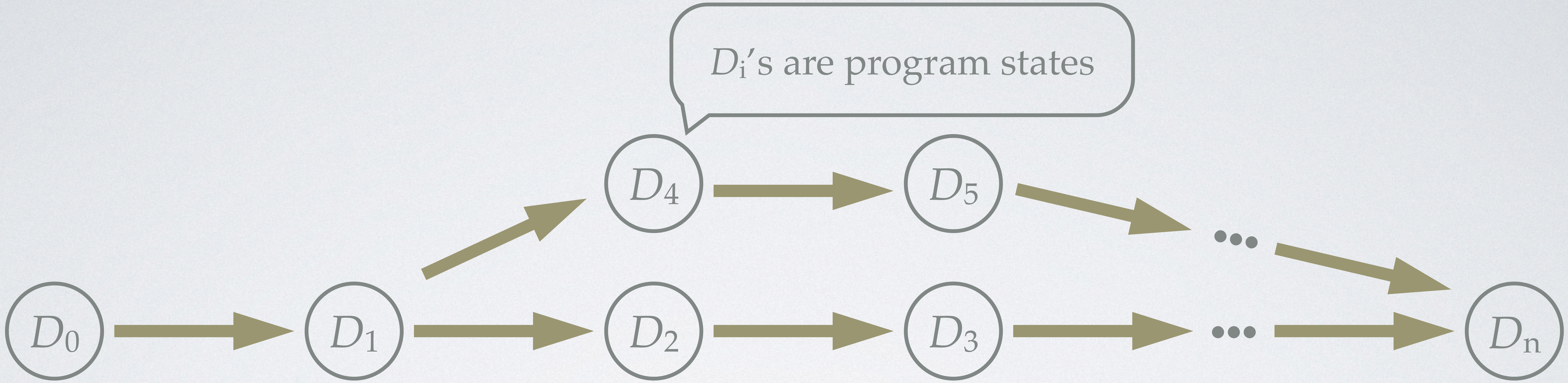


The Potential Method

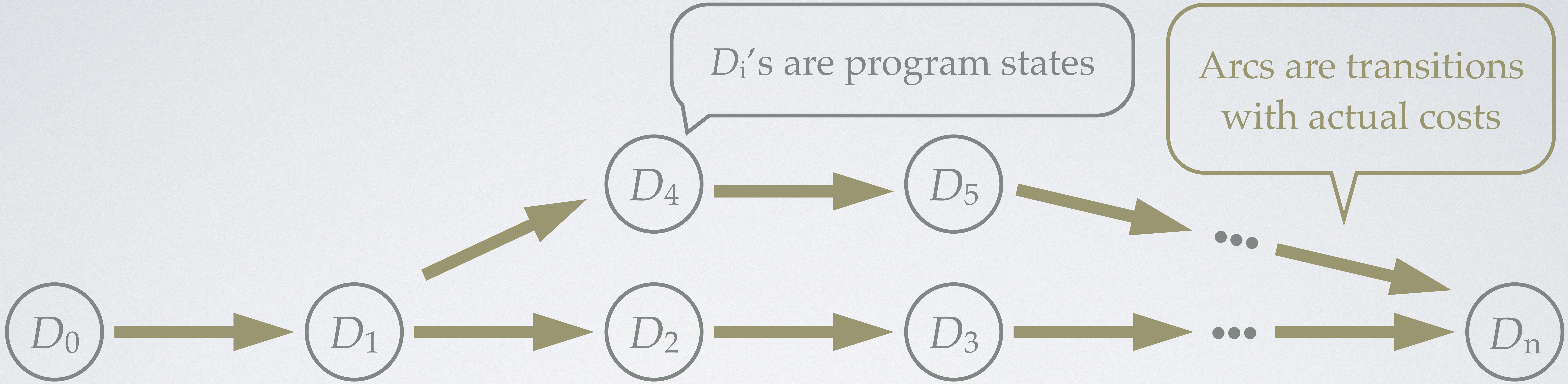
The Potential Method



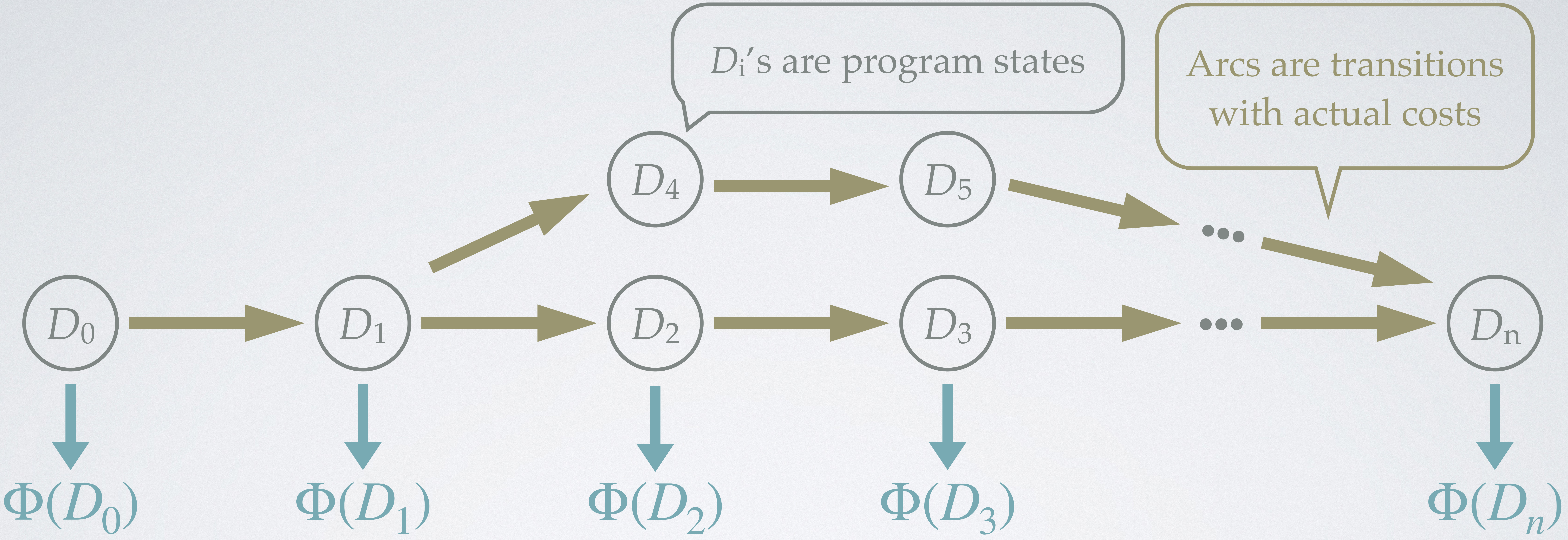
The Potential Method



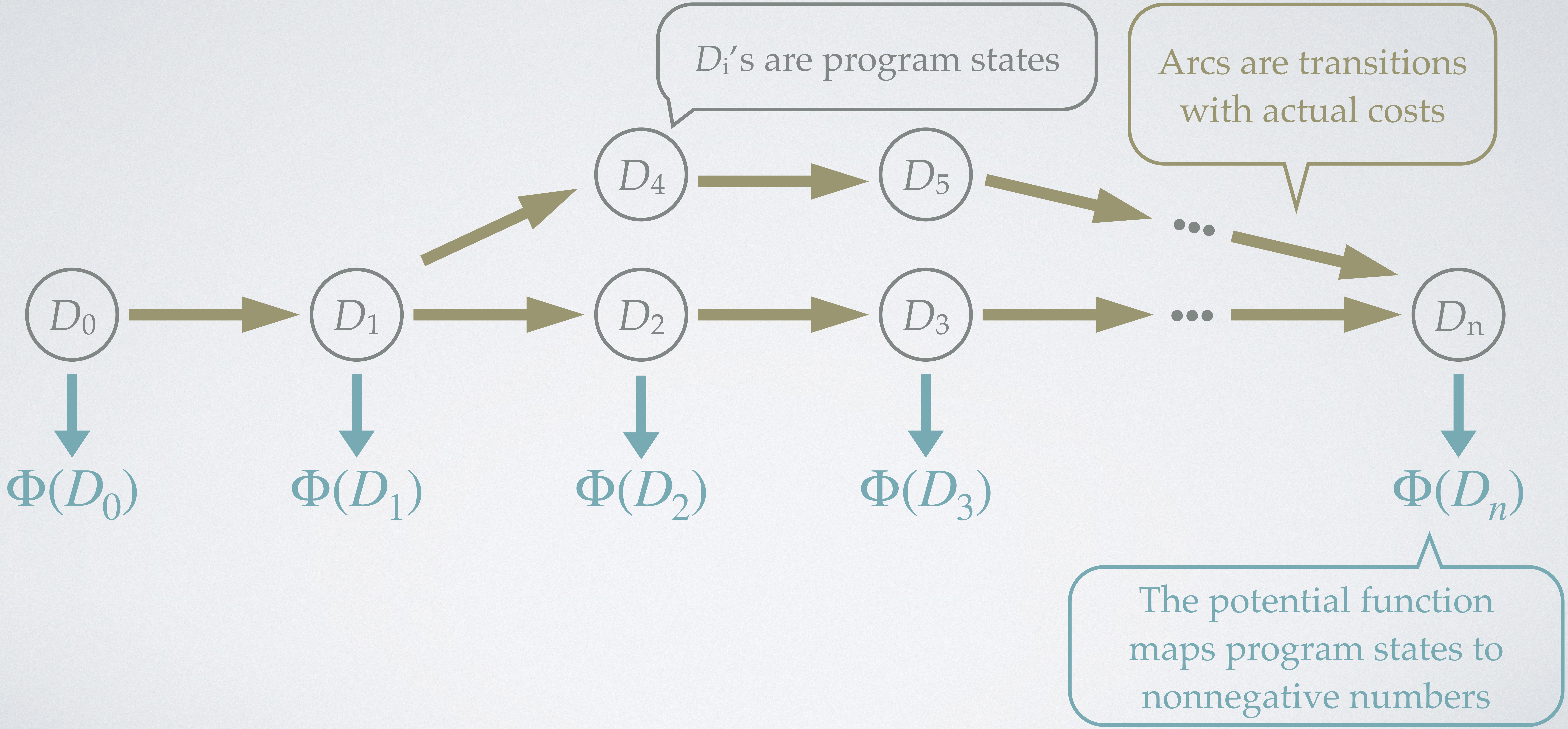
The Potential Method



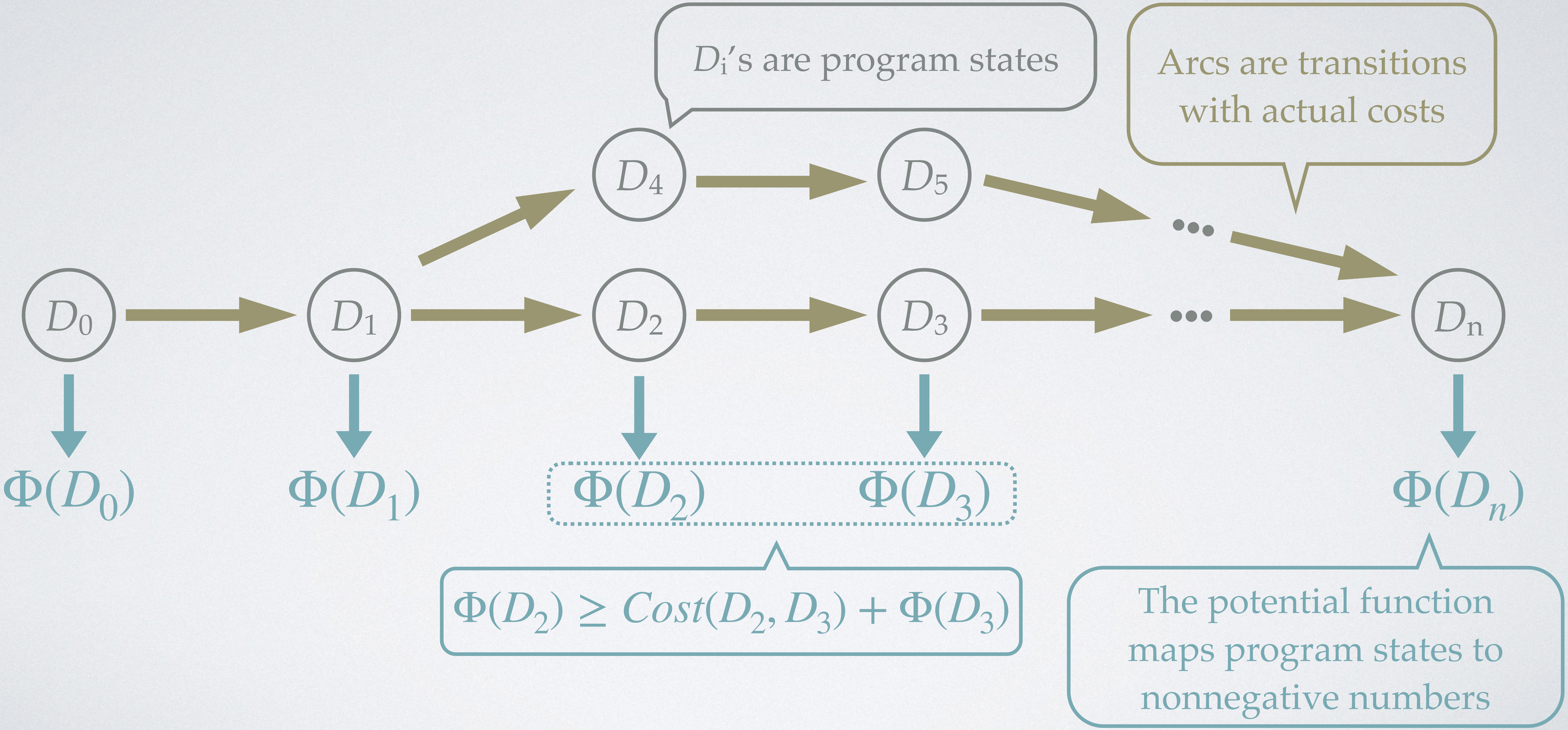
The Potential Method



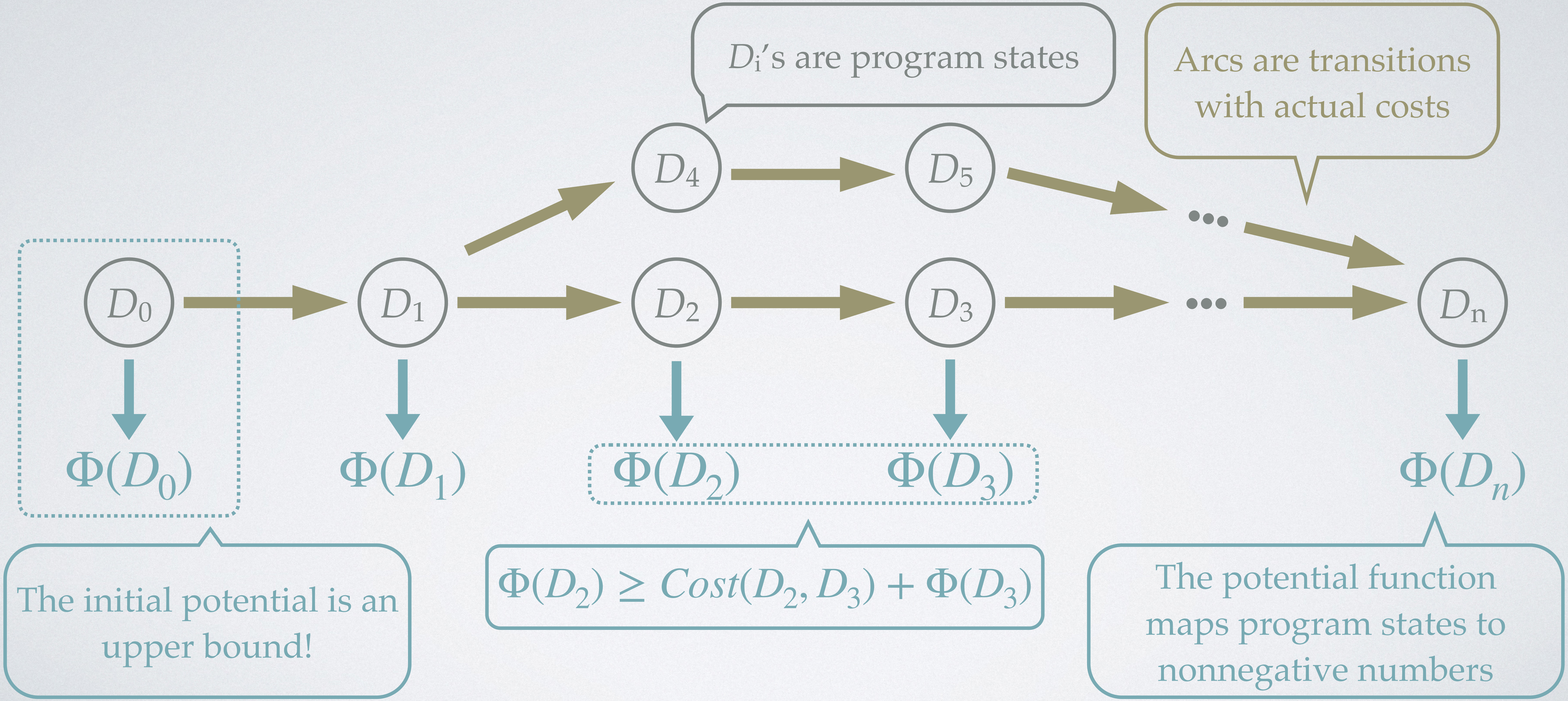
THE POTENTIAL METHOD



The Potential Method



The Potential Method



Potential-Augmented Types

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

Potential-Augmented Types

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

Resource metric:
count recursive calls

POTENTIAL-AUGMENTED TYPES

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

$Cost = |\ell_1|$

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

Resource metric: count recursive calls

Potential-Augmented Types

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

$Cost = |\ell_1|$

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

Resource metric: count recursive calls

POTENTIAL-AUGMENTED TYPES

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

$Cost = |\ell_1|$

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

Resource metric:
count recursive calls

Potential-Augmented Types

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

$Cost = |\ell_1|$

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

Resource metric: count recursive calls

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

POTENTIAL-AUGMENTED TYPES

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
```

Resource metric: count recursive calls

$Cost = |\ell_1|$

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

POTENTIAL-AUGMENTED TYPES

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
```

Resource metric: count recursive calls

$Cost = |\ell_1|$

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

Potential-Augmented Types

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

$Cost = |\ell_1|$

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

Resource metric:
count recursive calls

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 is consumed
[l2: L⁰(a)]; 0 units
// l2 is consumed. Type-checked!
```

Potential-Augmented Types

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

$Cost = |\ell_1|$

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

Resource metric: count recursive calls

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 is consumed
[l2: L⁰(a)]; 0 units
// l2 is consumed. Type-checked!
```

$$\frac{\Gamma; q \vdash e_1 : A \qquad \Gamma, x_h : \tau, x_t : L^p(\tau); q + p \vdash e_2 : A}{\Gamma, x : L^p(\tau); q \vdash \mathrm{mat}_{\mathrm{L}}\{e_1; x_h, x_t.e_2\}(x) : A} \text{ (L:MATL)}$$

POTENTIAL-AUGMENTED TYPES

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

$Cost = |\ell_1|$

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

Resource metric: count recursive calls

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 is consumed
[l2: L⁰(a)]; 0 units
// l2 is consumed. Type-checked!
```

$$\frac{\Gamma; q \vdash e_1 : A \qquad \Gamma, x_h : \tau, x_t : L^p(\tau); q + p \vdash e_2 : A}{\Gamma, x : L^p(\tau); q \vdash \mathsf{mat}_{\mathrm{L}}\{e_1; x_h, x_t.\, e_2\}(x) : A} \text{ (L:MATL)}$$

POTENTIAL-AUGMENTED TYPES

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

Resource metric: count recursive calls

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

$Cost = |\ell_1|$

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 is consumed
[l2: L⁰(a)]; 0 units
// l2 is consumed. Type-checked!
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
```

$$\frac{\Gamma; q \vdash e_1 : A \qquad \Gamma, x_h : \tau, x_t : L^p(\tau); q + p \vdash e_2 : A}{\Gamma, x : L^p(\tau); q \vdash \mathsf{mat}_L\{e_1; x_h, x_t.\, e_2\}(x) : A} \text{(L:MatL)}$$

Potential-Augmented Types

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

$Cost = |\ell_1|$

Resource metric: count recursive calls

```
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 is consumed
[l2: L⁰(a)]; 0 units
// l2 is consumed. Type-checked!
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
[l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
```

$$\frac{\Gamma; q \vdash e_1 : A \qquad \Gamma, x_h : \tau, x_t : L^p(\tau); q + p \vdash e_2 : A}{\Gamma, x : L^p(\tau); q \vdash \mathrm{mat}_{\mathrm{L}}\{e_1; x_h, x_t. e_2\}(x) : A} \text{(L:MatL)}$$

POTENTIAL-AUGMENTED TYPES

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

$Cost = |\ell_1|$

Resource metric: count recursive calls

```
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 is consumed
[l2: L⁰(a)]; 0 units
// l2 is consumed. Type-checked!
[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
```

$$\frac{\Gamma; q \vdash e_1 : A \qquad \Gamma, x_h : \tau, x_t : L^p(\tau); q + p \vdash e_2 : A}{\Gamma, x : L^p(\tau); q \vdash \mathrm{mat}_L\{e_1; x_h, x_t.\, e_2\}(x) : A} \ (\text{L:MATL})$$

POTENTIAL-AUGMENTED TYPES

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

$Cost = |\ell_1|$

Resource metric: count recursive calls

```
let rec append l1 l2 =            [l1: L¹(a), l2: L⁰(a)]; 0 units
  match l1 with                   // l1 is consumed
  | [] ->                         [l2: L⁰(a)]; 0 units
    l2                            // l2 is consumed. Type-checked!
  | x::xs ->                      [l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
    let () = tick(1) in           [l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
    let rest = append xs l2 in    [x: a, rest: L⁰(a)]; 0 units
    x::rest                       // x and rest are consumed. Type-checked!
```

$$\frac{\Gamma; q \vdash e_1 : A \qquad \Gamma, x_h : \tau, x_t : L^p(\tau); q + p \vdash e_2 : A}{\Gamma, x : L^p(\tau); q \vdash \mathrm{mat}_L\{e_1; x_h, x_t.\, e_2\}(x) : A} \text{ (L:MatL)}$$

Potential-Augmented Types

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

$Cost = |\ell_1|$

Resource metric: count recursive calls

```
let rec append l1 l2 =           [l1: L¹(a), l2: L⁰(a)]; 0 units
  match l1 with                  // l1 is consumed
  | [] ->                        [l2: L⁰(a)]; 0 units
    l2                           // l2 is consumed. Type-checked!
  | x::xs ->                     [l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
    let () = tick(1) in          [l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
    let rest = append xs l2 in   [x: a, rest: L⁰(a)]; 0 units
    x::rest                      // x and rest are consumed. Type-checked!
```

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

AUTOMATION VIA LP SOLVING

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

Automation via LP Solving

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
```

AUTOMATION VIA LP SOLVING

p, q, r, s, t are unknown numeric variables

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
```

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

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
```

Linear Constraints

```
p≥0,q≥0,r≥0,s≥0,t≥0
```

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

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
```

Linear Constraints

p≥0, q≥0, r≥0, s≥0, t≥0

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

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 is consumed
```

Linear Constraints

```
p≥0,q≥0,r≥0,s≥0,t≥0
```

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

Linear Constraints

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 is consumed
[l2: Lq(a)]; r units
```

```
p≥0,q≥0,r≥0,s≥0,t≥0
```

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

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 is consumed
[l2: Lq(a)]; r units
// l2 is consumed
```

Linear Constraints

p≥0,q≥0,r≥0,s≥0,t≥0

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

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 is consumed
  | [] ->                         [l2: Lq(a)]; r units
    l2                            // l2 is consumed
  | x::xs ->
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

Linear Constraints

p≥0,q≥0,r≥0,s≥0,t≥0

q≥s,r≥t

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

Linear Constraints

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 is consumed
  | [] ->                       [l2: Lq(a)]; r units
    l2                          // l2 is consumed
  | x::xs ->                    [l2: Lq(a), x: a, xs: Lp(a)]; r+p units
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

```
p≥0,q≥0,r≥0,s≥0,t≥0

q≥s,r≥t
```

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

Linear Constraints

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 is consumed
  | [] ->                         [l2: Lq(a)]; r units
    l2                            // l2 is consumed
  | 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::rest
```

```
p≥0,q≥0,r≥0,s≥0,t≥0

q≥s,r≥t
```

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

Linear Constraints

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       p≥0,q≥0,r≥0,s≥0,t≥0
  match l1 with                 // l1 is consumed
  | [] ->                       [l2: Lq(a)]; r units
    l2                          // l2 is consumed                      q≥s,r≥t
  | 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   r+p-1≥0
    let rest = append xs l2 in
    x::rest
```

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

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

```
let rec append l1 l2 =            [l1: Lp(a), l2: Lq(a)]; r units
  match l1 with                   // l1 is consumed
  | [] ->                         [l2: Lq(a)]; r units
    l2                            // l2 is consumed
  | 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
```

Linear Constraints

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

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

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 is consumed
  | [] ->                         [l2: Lq(a)]; r units
    l2                            // l2 is consumed
  | 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
```

Linear Constraints

```
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
```

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

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

Linear Constraints

```
let rec append l1 l2 =          [l1: Lp(a), l2: Lq(a)]; r units        p≥0,q≥0,r≥0,s≥0,t≥0
  match l1 with                 // l1 is consumed
  | [] ->                       [l2: Lq(a)]; r units
    l2                          // l2 is consumed                       q≥s,r≥t
  | 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   r+p-1≥0
    let rest = append xs l2 in  [x: a, rest: Ls(a)]; p-1+t units        p≥p,q≥q,r+p-1≥r
    x::rest                     // x and rest are consumed
```

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

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

Linear Constraints

```
let rec append l1 l2 =           [l1: Lp(a), l2: Lq(a)]; r units           p≥0,q≥0,r≥0,s≥0,t≥0
  match l1 with                  // l1 is consumed
  | [] ->                        [l2: Lq(a)]; r units
    l2                           // l2 is consumed                         q≥s,r≥t
  | 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 r+p-1≥0
    let rest = append xs l2 in   [x: a, rest: Ls(a)]; p-1+t units           p≥p,q≥q,r+p-1≥r
    x::rest                      // x and rest are consumed                p-1+t≥s+t
```

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

Linear Constraints

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          p≥0,q≥0,r≥0,s≥0,t≥0
  match l1 with                  // l1 is consumed
  | [] ->                        [l2: Lq(a)]; r units
    l2                           // l2 is consumed                         q≥s,r≥t
  | 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 r+p-1≥0
    let rest = append xs l2 in   [x: a, rest: Ls(a)]; p-1+t units          p≥p,q≥q,r+p-1≥r
    x::rest                      // x and rest are consumed                p-1+t≥s+t
```

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

AUTOMATION VIA LP SOLVING

p,q,r,s,t are unknown numeric variables

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

Linear Constraints

```
let rec append l1 l2 =              [l1: Lp(a), l2: Lq(a)]; r units         p≥0,q≥0,r≥0,s≥0,t≥0
  match l1 with                     // l1 is consumed
  | [] ->                           [l2: Lq(a)]; r units
    l2                              // l2 is consumed                       q≥s,r≥t
  | 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  r+p-1≥0
    let rest = append xs l2 in      [x: a, rest: Ls(a)]; p-1+t units        p≥p,q≥q,r+p-1≥r
    x::rest                         // x and rest are consumed              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

Automation via LP Solving

p,q,r,s,t are unknown numeric variables

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

Linear Constraints

```
let rec append l1 l2 =          [l1: Lp(a), l2: Lq(a)]; r units          p≥0,q≥0,r≥0,s≥0,t≥0
  match l1 with                 // l1 is consumed
  | [] ->                       [l2: Lq(a)]; r units
    l2                          // l2 is consumed                          q≥s,r≥t
  | 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  r+p-1≥0
    let rest = append xs l2 in  [x: a, rest: Ls(a)]; p-1+t units           p≥p,q≥q,r+p-1≥r
    x::rest                     // x and rest are consumed                 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

The Frontier of AARA

[RaML17]	Multivariate polynomial bounds, amortized complexity (binary counters, …)
[Atkey10]	Imperative programs, heaps, separation logic
[JHL+10]	Higher-order functions
[HM18]	Logarithmic amortized complexity (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*.

Can we use the type information from AARA to guide other tasks?

Type-Based Resource-Guided Search

Type-Based Resource-Guided Search

- **Search** algorithms are used in many PL-related tasks.

Type-Based Resource-Guided Search

- **Search** algorithms are used in many PL-related tasks.
- **Symbolic Execution**: **search** for an execution path that satisfies constraints.

Type-Based Resource-Guided Search

- **Search** algorithms are used in many PL-related tasks.
- **Symbolic Execution**: **search** for an execution path that satisfies constraints.
- **Program Synthesis**: **search** for a program that satisfies specifications.

Type-Based Resource-Guided Search

- Search algorithms are used in many PL-related tasks.
- **Symbolic Execution**: search for an execution path that satisfies constraints.
- **Program Synthesis**: search for a program that satisfies specifications.

Type-Based Resource-Guided Search

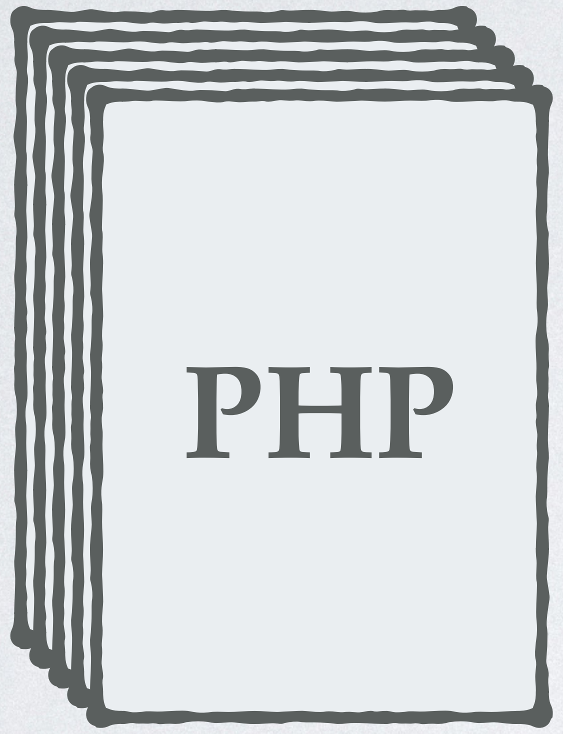
- Search algorithms are used in many PL-related tasks.
- **Symbolic Execution**: search for an execution path that satisfies constraints.
- **Program Synthesis**: search for a program that satisfies specifications.

- **Idea**: Resource information can be used to prune the search space.

OUTLINE

- [x] Automatic Amortized Resource Analysis
- [ ] Type-Guided **Worst-Case Input** Generation
- [ ] Resource-Guided Program Synthesis

Example of Worst-Case Analysis

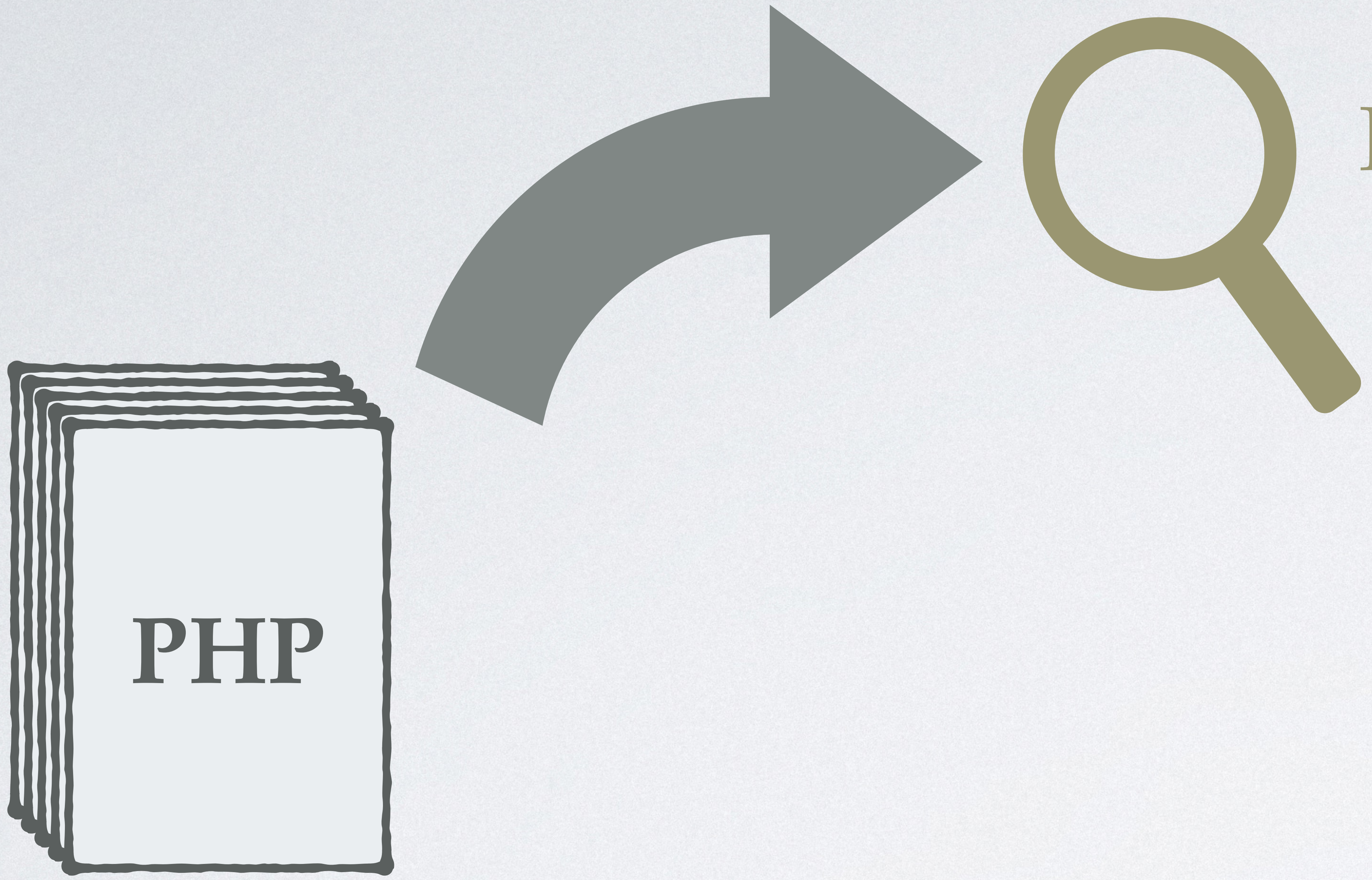


[1] CVE - CVE-2011-4885. Available on: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885.

[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.

EXAMPLE OF WORST-CASE ANALYSIS



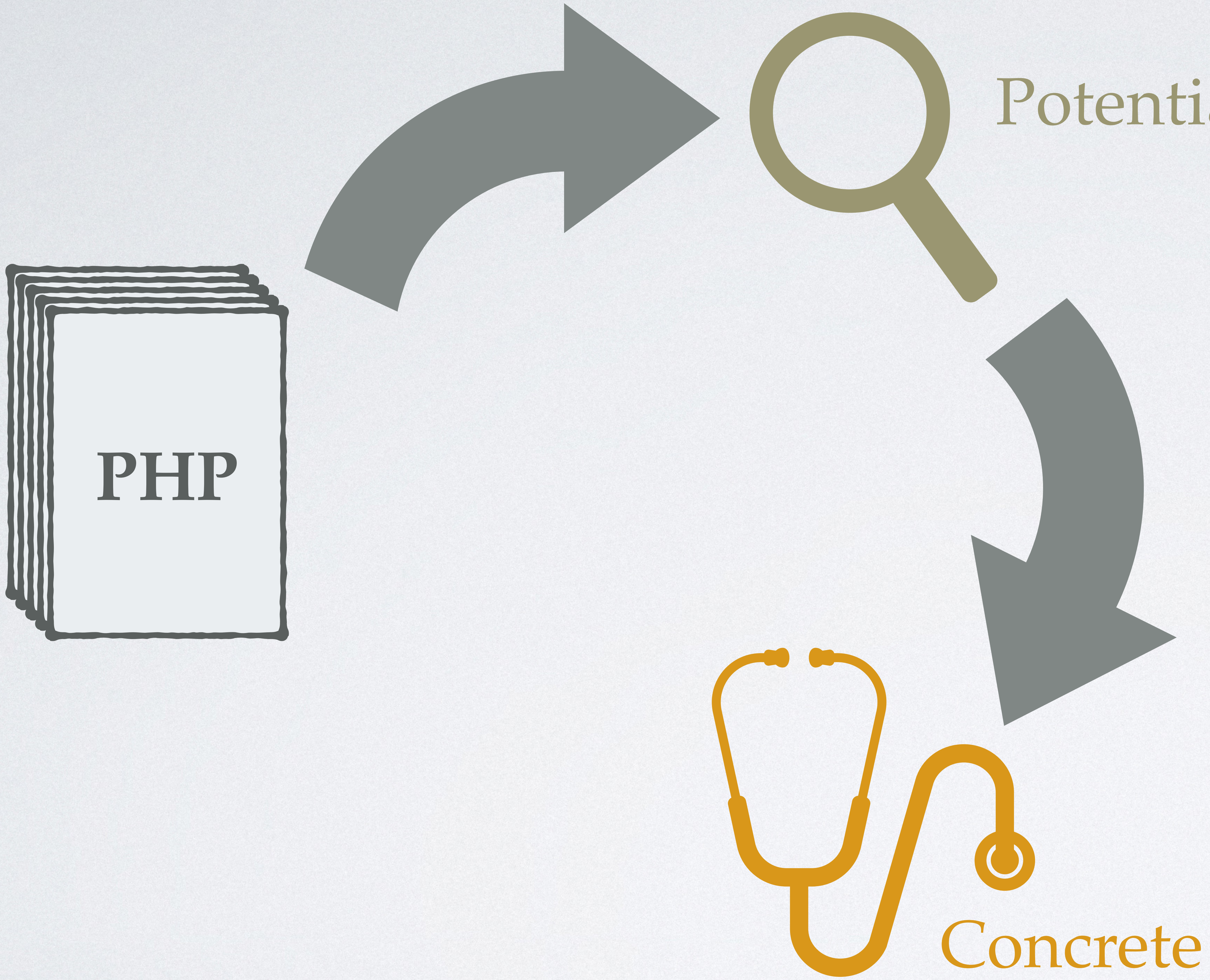
Potential Denial-of-Service attack[1]

[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.

EXAMPLE OF WORST-CASE ANALYSIS



Potential Denial-of-Service attack[1]

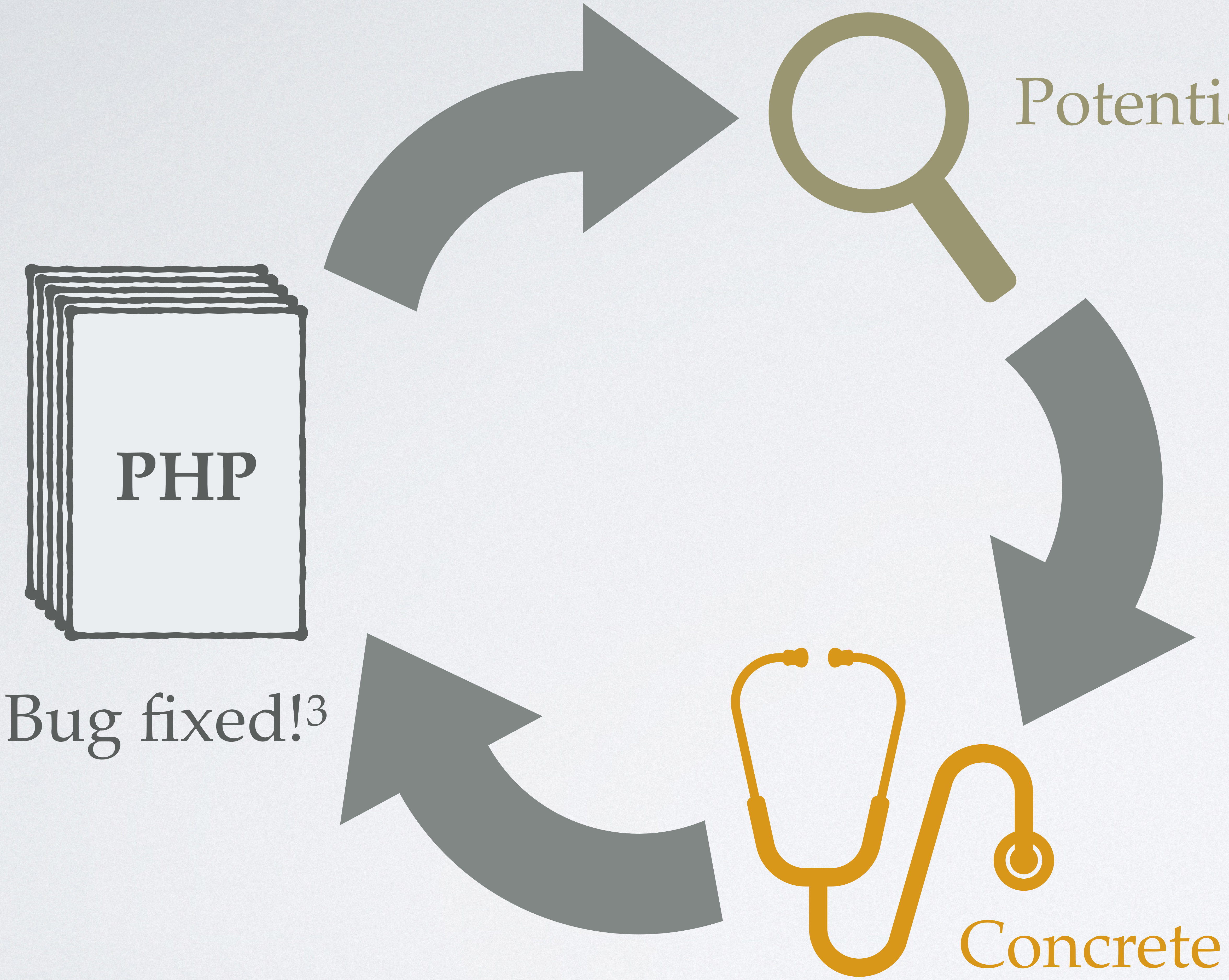
Concrete exploits (by hash collisions)[2]

[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.

Example of 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.

Example of Worst-Case Analysis

Worst-case inputs are instrumental to understand and fix 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.

Existing Approaches

EXISTING APPROACHES

Dynamic

- Fuzz testing
- Symbolic execution
- Dynamic worst-case analysis
- …

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

Existing Approaches

Dynamic

- Fuzz testing
- Symbolic execution
- Dynamic worst-case analysis
- …

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

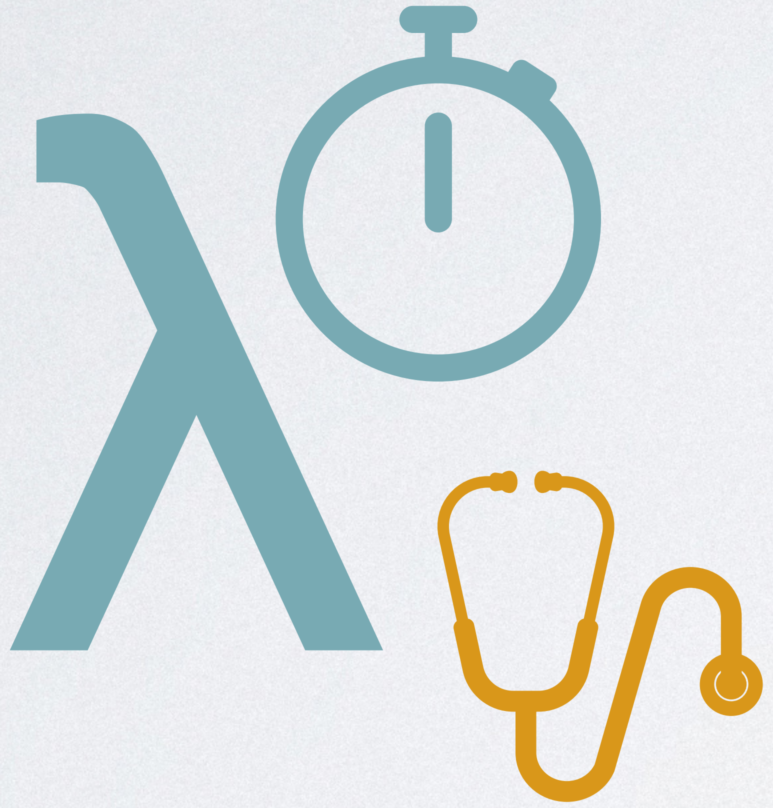
Static

- Type systems
- Abstract interpretation
- …

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

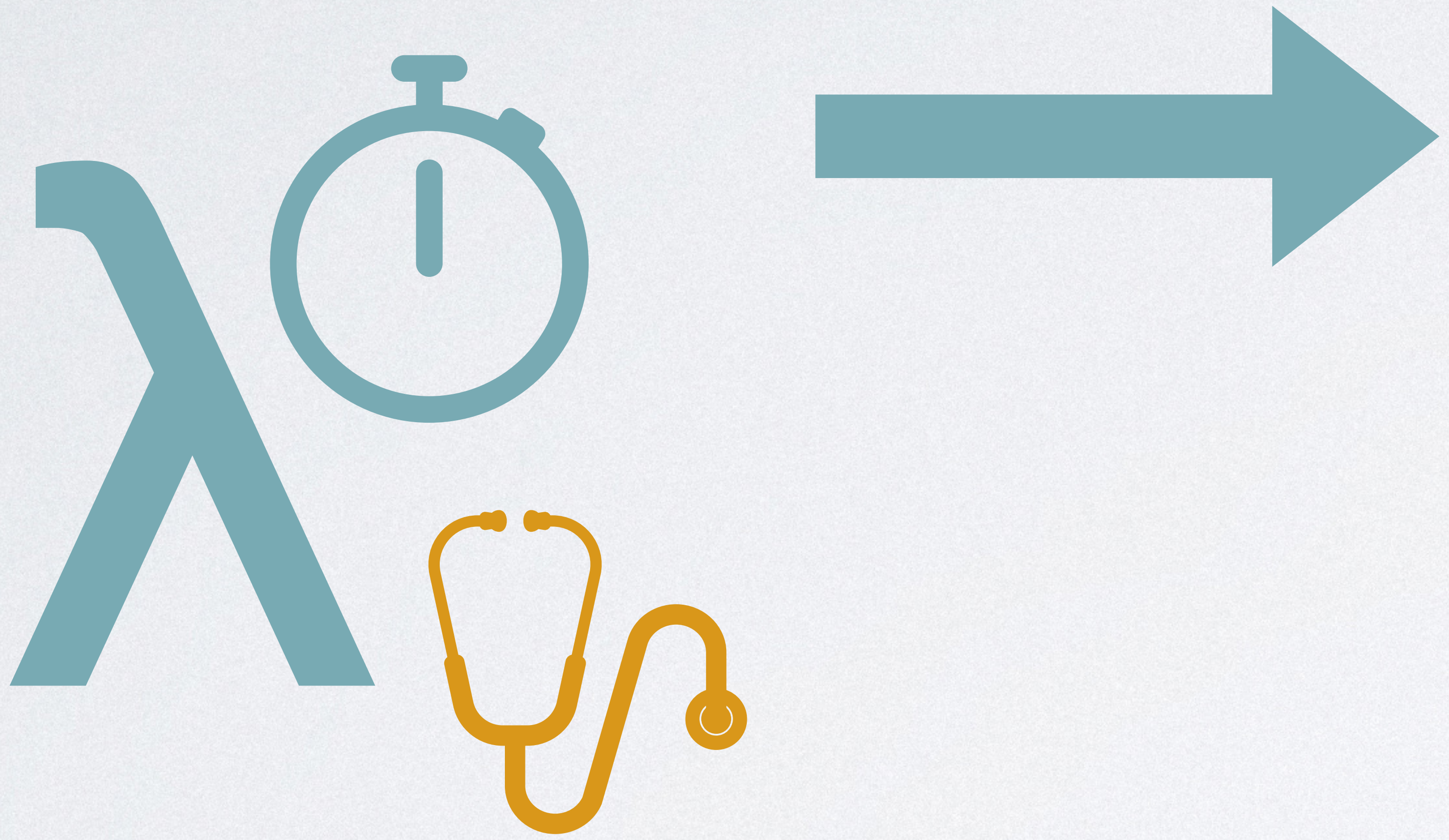
Type-Guided Worst-Case Input Generation

D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *POPL'19*.



TYPE-GUIDED WORST-CASE INPUT GENERATION

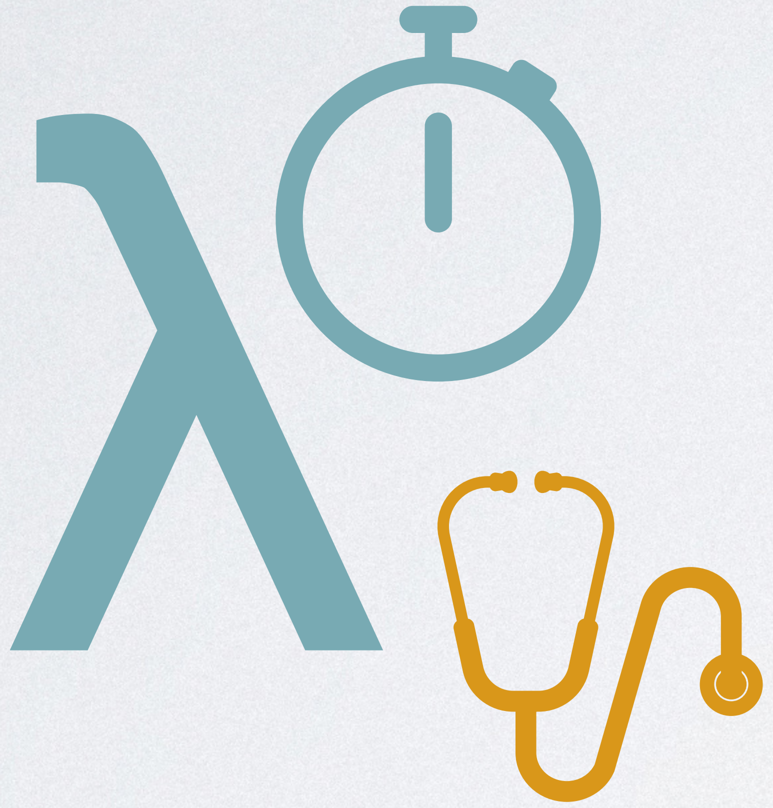
D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *POPL'19*.



Resource Aware ML (RaML)

Type-Guided Worst-Case Input Generation

D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *POPL'19*.



Resource Aware ML (RaML)

Type-Guided Worst-Case Input Generation

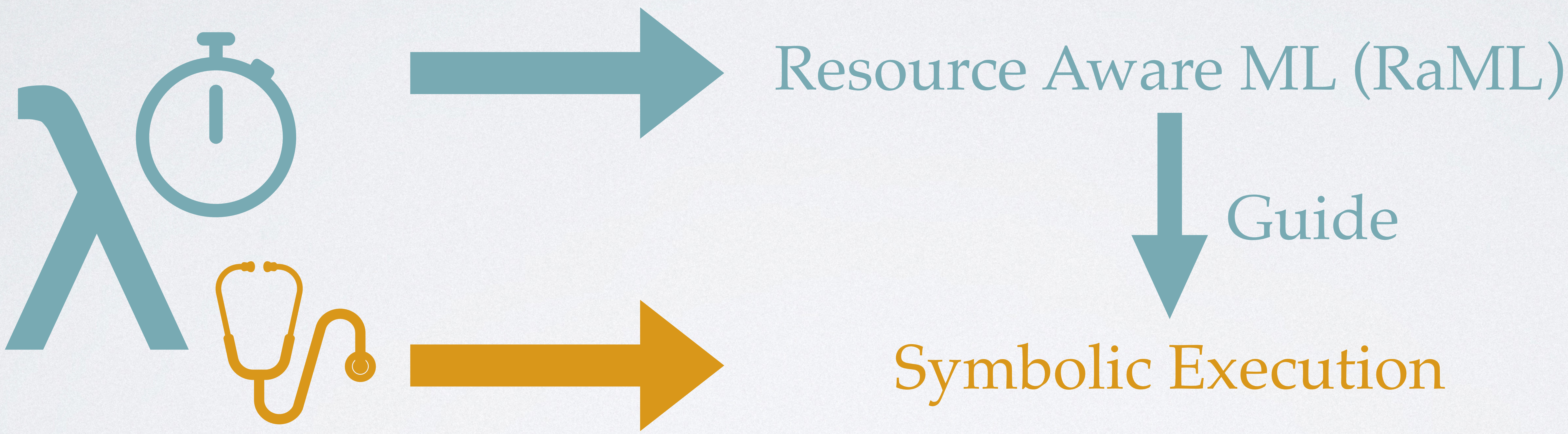
D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *POPL'19*.

Resource Aware ML (RaML)

Symbolic Execution

Type-Guided Worst-Case Input Generation

D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *POPL'19*.



Symbolic Execution

- **Idea**: **search** all execution paths, **record** path constraints, and **compute** resource usage.

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

Symbolic Execution

- **Idea**: **search** all execution paths, **record** path constraints, and **compute** resource usage.

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

symbolic environment

Symbolic Execution

- **Idea**: **search** all execution paths, **record** path constraints, and **compute** resource usage.

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

symbolic environment

expression

Symbolic Execution

- **Idea**: <u>**search**</u> all execution paths, <u>**record**</u> path constraints, and <u>**compute**</u> resource usage.

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

symbolic environment

expression

path constraints

Symbolic Execution

- **Idea**: **search** all execution paths, **record** path constraints, and **compute** resource usage.

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

symbolic environment

expression

path constraints

symbolic evaluation result

Symbolic Execution

- **Idea**: **search** all execution paths, **record** path constraints, and **compute** resource usage.

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

symbolic environment

expression

path constraints

symbolic evaluation result

- Symbolic execution rules for conditional expressions:

SYMBOLIC EXECUTION

- **Idea**: **search** all execution paths, **record** path constraints, and **compute** resource usage.

$$\underbrace{\gamma}_{\text{symbolic environment}} \vdash \underbrace{e}_{\text{expression}} \Rightarrow \underbrace{\langle \underbrace{\psi}_{\text{path constraints}}, S \rangle}_{\text{symbolic evaluation result}}$$

- Symbolic execution rules for conditional expressions:

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

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

Symbolic Execution

```
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'
```

Symbolic Execution

```
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'
```

Worst case: always enter the then-branch

Symbolic Execution

```
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'
```

Worst case: always enter the then-branch

- An example of worst-case execution paths for input lists of length 4:

$$\ell \mapsto [\mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3, \mathsf{int}^4] \vdash$$
$$\mathsf{lpairs}\ \ell \Rightarrow \langle (\mathsf{int}^1 < \mathsf{int}^2) \wedge (\mathsf{int}^3 < \mathsf{int}^4),$$
$$[(\mathsf{int}^1, \mathsf{int}^2), (\mathsf{int}^3, \mathsf{int}^4)] \rangle$$

Symbolic Execution

```
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'
```

Worst case: always enter the then-branch

- An example of worst-case execution paths for input lists of length 4:

$$\ell \mapsto [\mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3, \mathsf{int}^4] \vdash$$
$$\mathsf{lpairs}\ \ell \Rightarrow \langle (\mathsf{int}^1 < \mathsf{int}^2) \wedge (\mathsf{int}^3 < \mathsf{int}^4), [(\mathsf{int}^1, \mathsf{int}^2), (\mathsf{int}^3, \mathsf{int}^4)] \rangle$$

- Invoke an SMT solver to find a model, e.g., [0,1,0,1].

Type-Guided Symbolic Execution

- **Nondeterminism** leads to state explosion:

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

Type-Guided Symbolic Execution

- **Nondeterminism** leads to state explosion:

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

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

Use the information about **potentials** obtained from **resource aware type checking** to **prune the search space** of symbolic execution.

Type-Guided Symbolic Execution

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

```
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'
```

Type-Guided Symbolic Execution

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

```
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'
```

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

Type-Guided Symbolic Execution

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

```
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'
```

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

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

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

TYPE-GUIDED SYMBOLIC EXECUTION

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

```
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'
```

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

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

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

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

TYPE-GUIDED SYMBOLIC EXECUTION

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

```
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'
```

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

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

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

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

$\text{Cost} = 2$

Type-Guided Symbolic Execution

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

```
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'
```

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

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

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

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

$\text{Cost} = 2$

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

Type-Guided Symbolic Execution

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

```
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'
```

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

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

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

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

$\text{Cost} = 2$

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

TYPE-GUIDED SYMBOLIC EXECUTION

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

```
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'
```

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

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

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

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

$\text{Cost} = 2$

Waste!

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

Type-Guided Symbolic Execution

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

```
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'
```

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

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

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

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

Cost = 2

Waste!

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

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

Type-Guided Symbolic Execution

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

```
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'
```

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

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

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

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

$\text{Cost} = 2$

Waste!

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

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

Prune the search space!

Theoretical Results

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

Speed up Input Generation

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

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

Speed up Input Generation

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

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

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

Speed up Input Generation

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

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

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

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

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

Still Sound!

Speed up Input Generation

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

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

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

Still Sound!

- **Generalization**: enforce all the calls with the ***same*** shape of inputs execute the ***same*** path in the function body.

Example: Quicksort

Terminalizer

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Dis-iMac:raml di$
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Dis-iMac:raml di$
```

Example: Quicksort

Terminalizer

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Dis-iMac:raml di$
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Dis-iMac:raml di$
```

OUTLINE

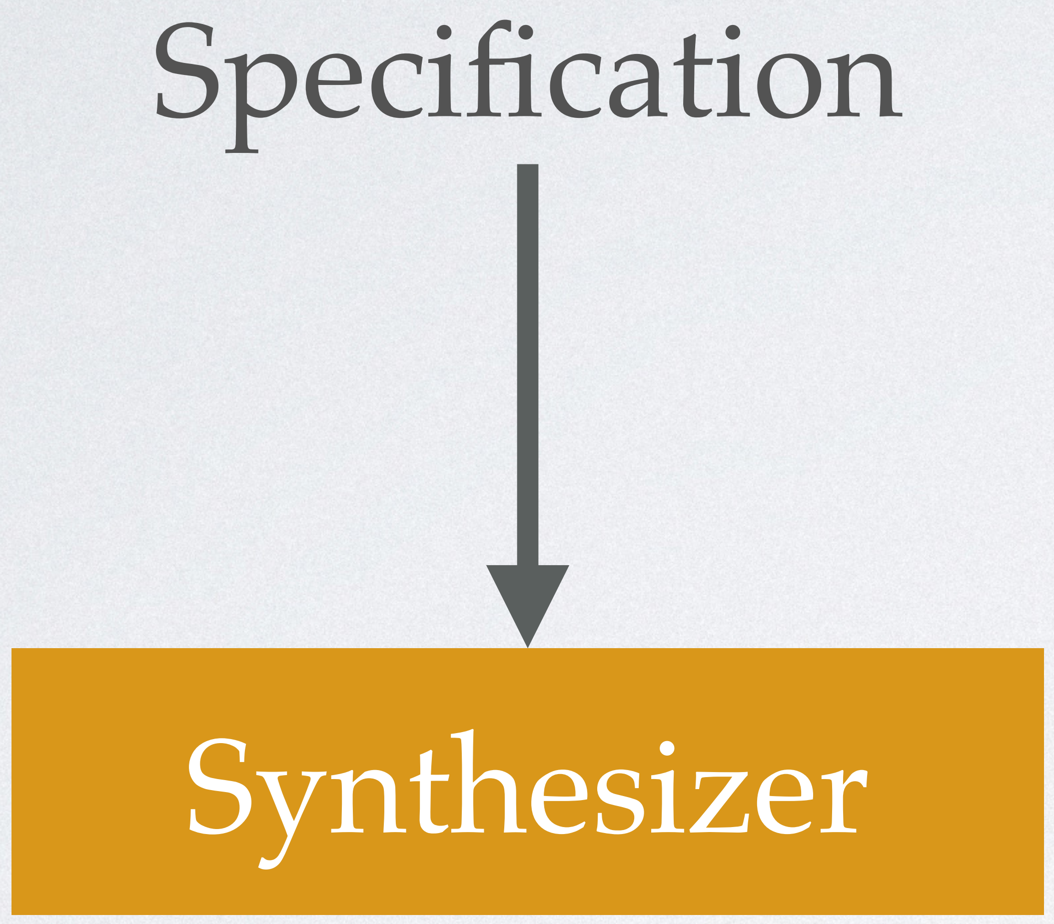
- ☑ Automatic Amortized Resource Analysis
- ☑ Type-Guided Worst-Case Input Generation
- ☐ Resource-Guided **Program Synthesis**

Type-Directed Synthesis

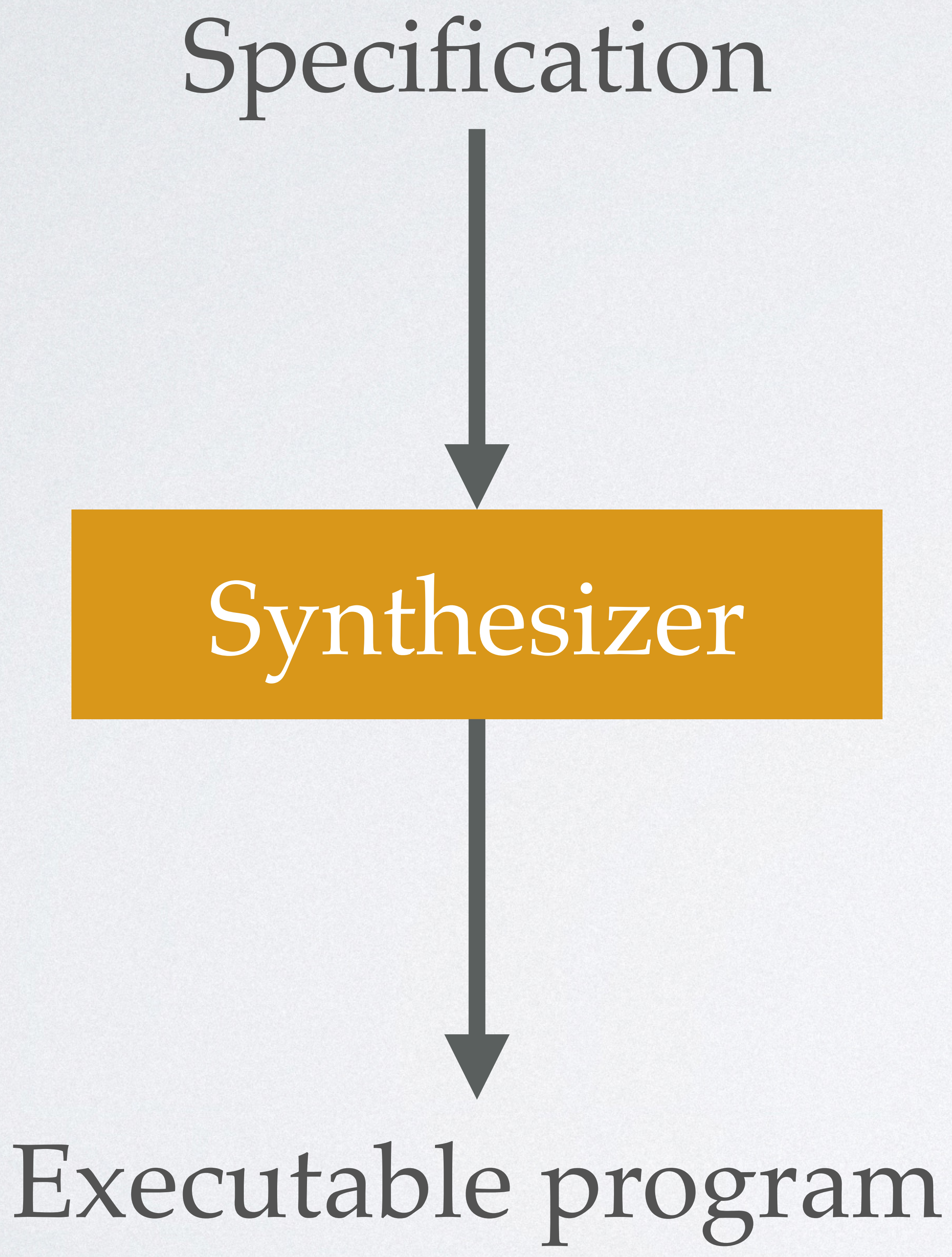
Type-Directed Synthesis

Specification

TYPE-DIRECTED SYNTHESIS



Type-Directed Synthesis



TYPE-DIRECTED SYNTHESIS

Specification

Synthesizer

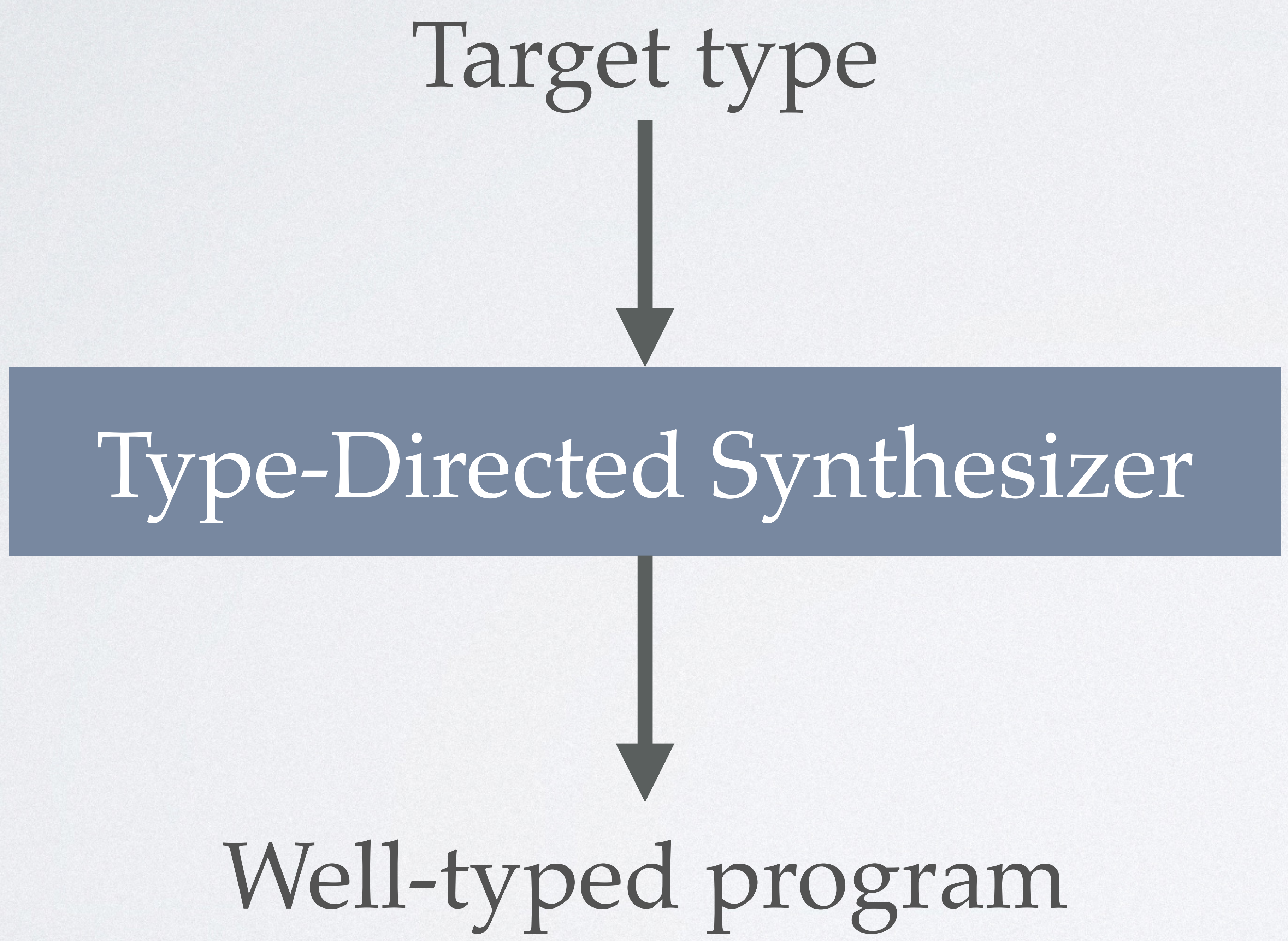
Executable program

Target type

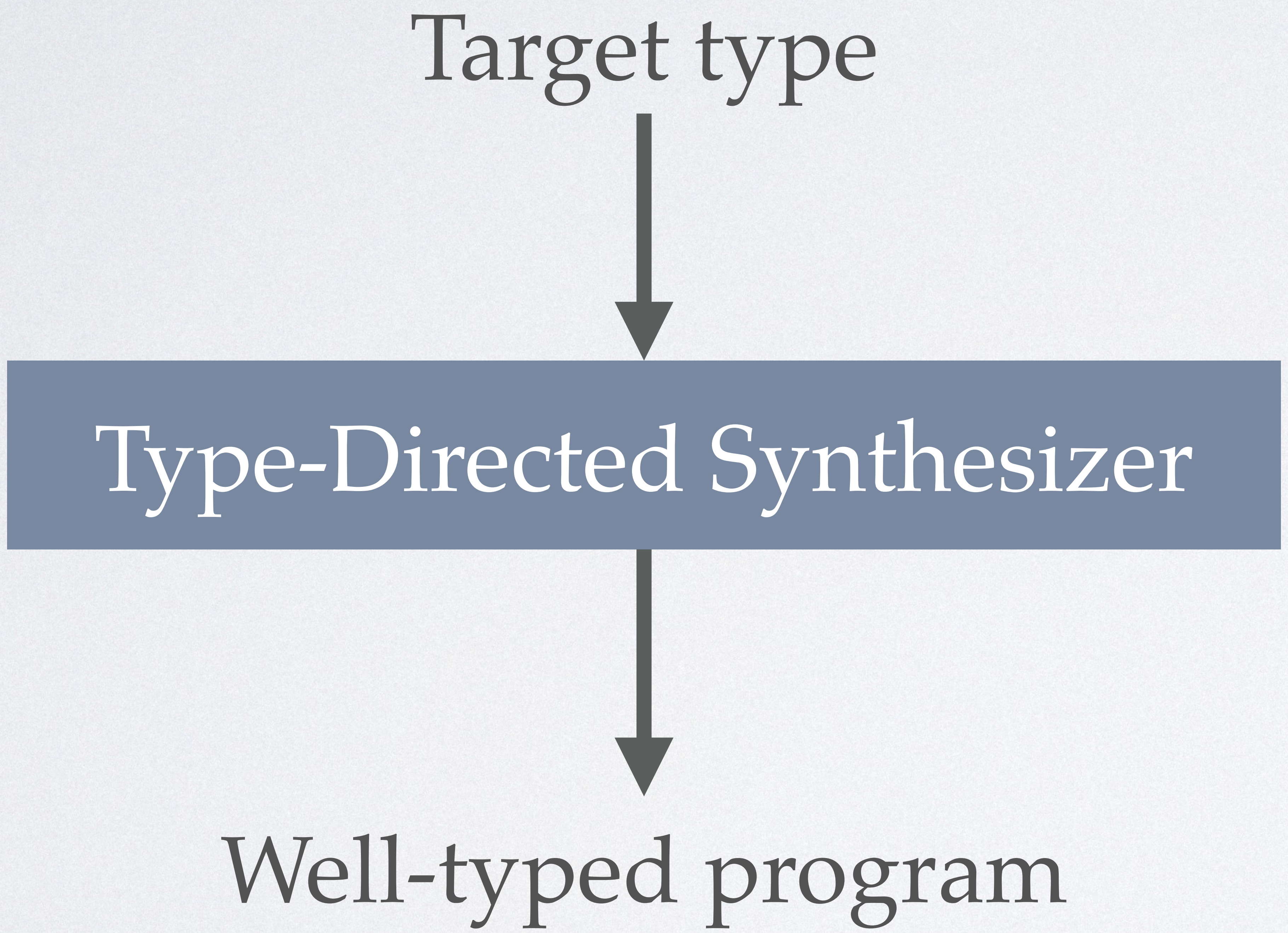
Type-Directed Synthesizer

Well-typed program

Type-Directed Synthesis



Type-Directed Synthesis




```
id : a -> a
```

Type-Directed Synthesis

Target type

↓

Type-Directed Synthesizer

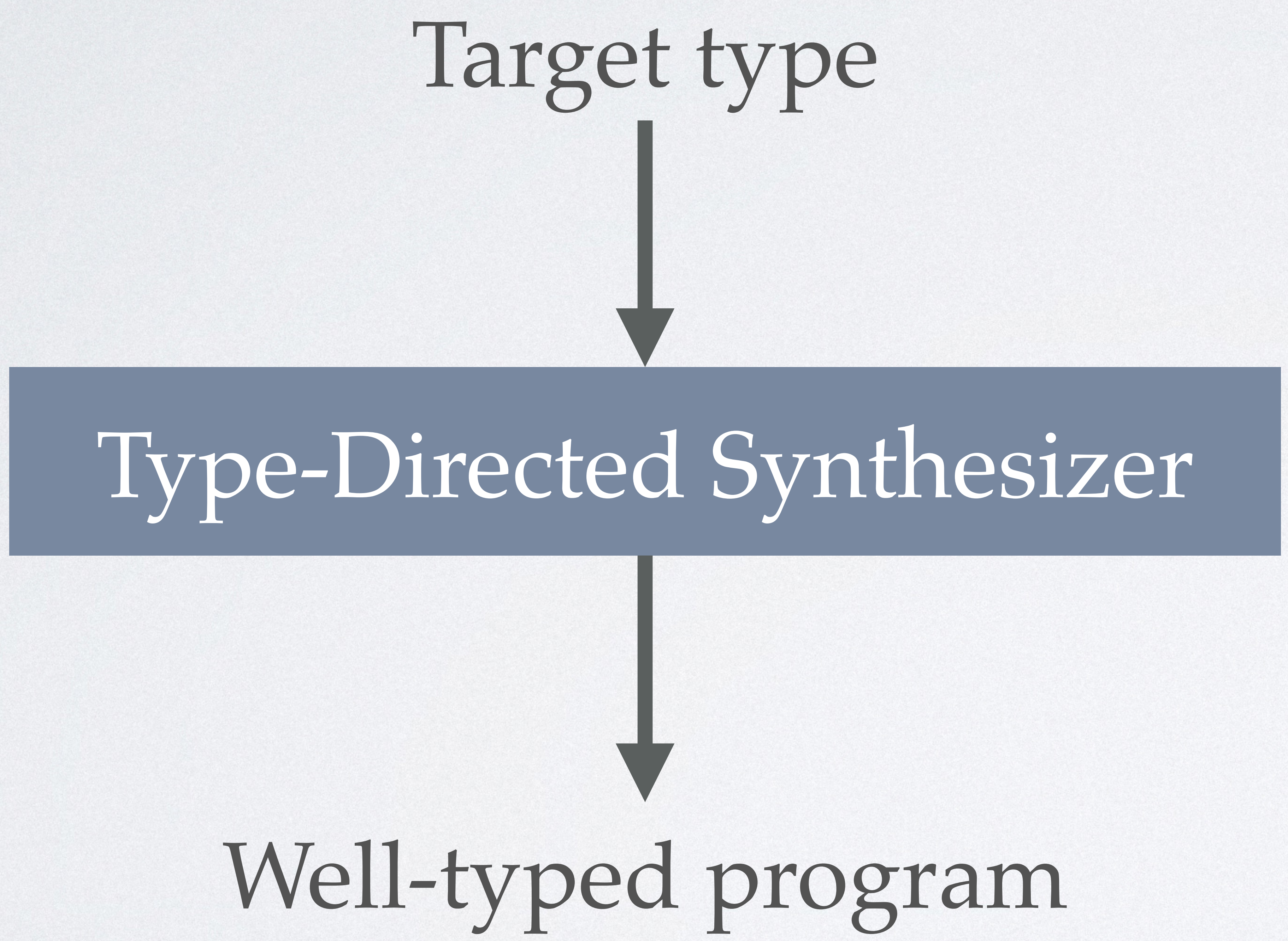
↓

Well-typed program

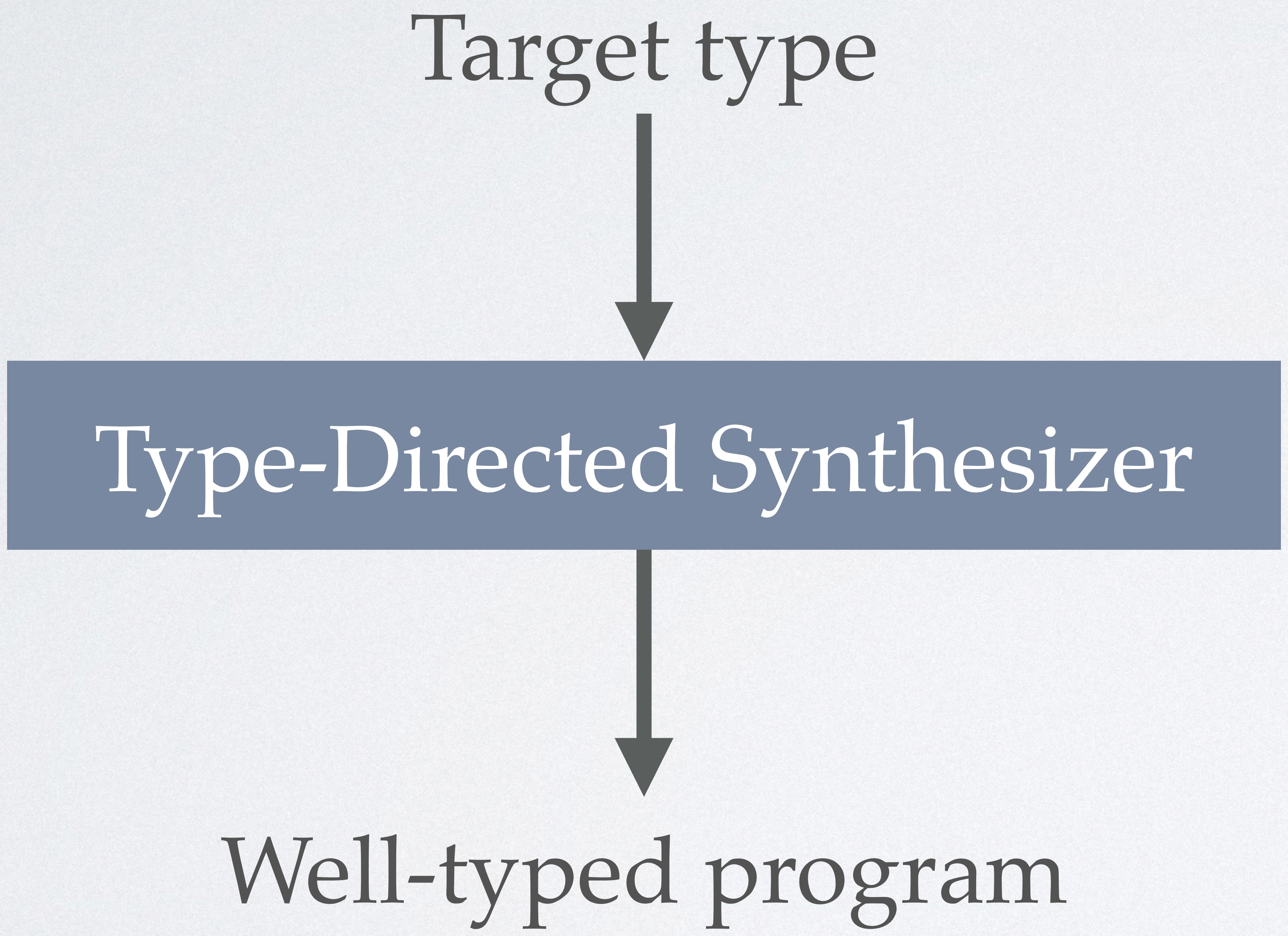
```
id : a -> a
```

```
let id x = x
```

Type-Directed Synthesis

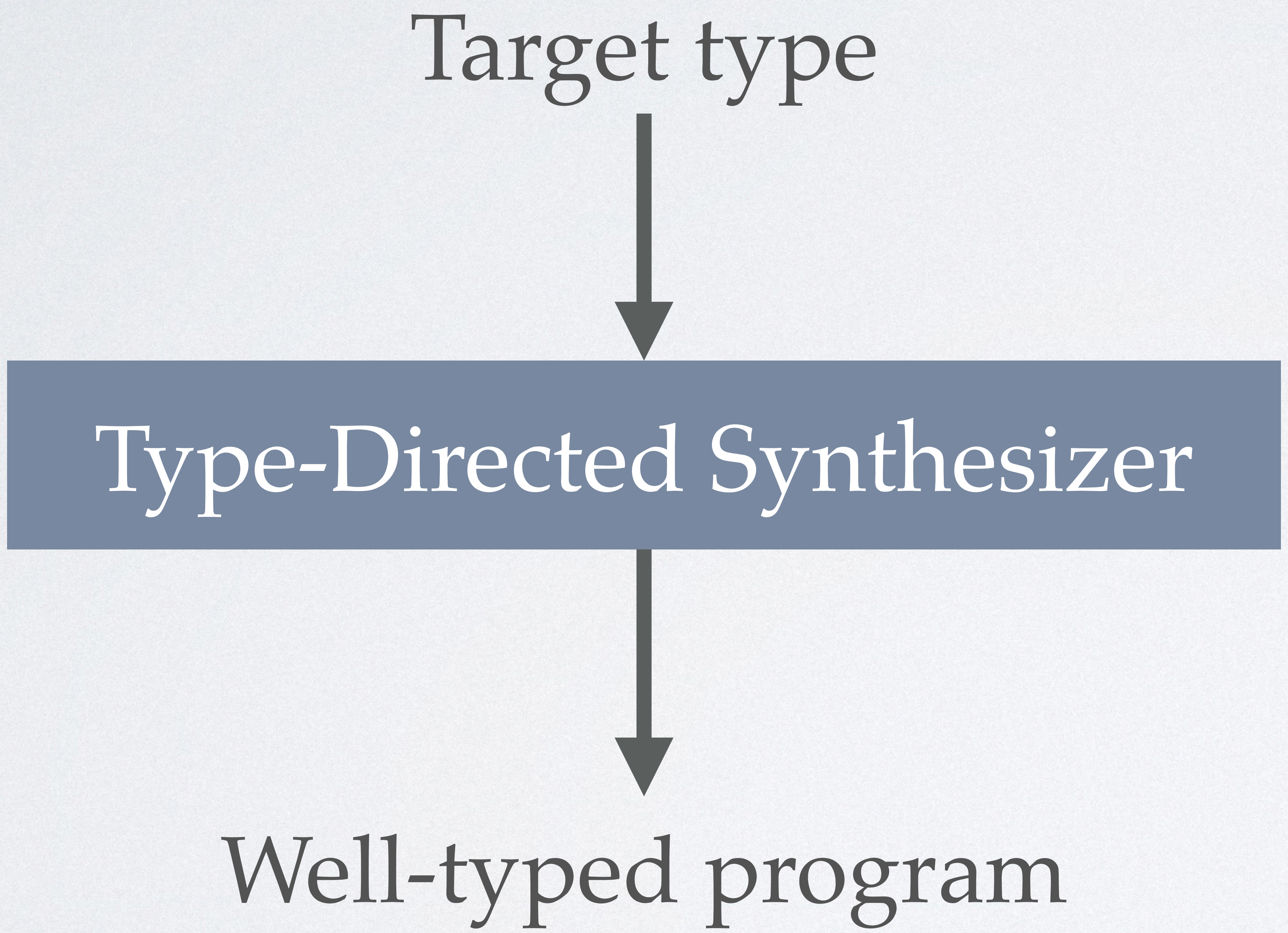


Type-Directed Synthesis




```
rep : int -> a -> List a
```

TYPE-DIRECTED SYNTHESIS




```
rep : int -> a -> List a
```

```
let rep n x = []
```

TYPE-DIRECTED SYNTHESIS

Target type

Type-Directed Synthesizer

Well-typed program

```
rep : int -> a -> List a
```

does not implement the replicate function

```
let rep n x = []
```

LIQUID TYPES

[RKJ08] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.

Liquid Types

`{ v: B | ψ }`

A value `v` of type `B` that satisfies `ψ`

[RKJ08] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.

LIQUID TYPES

`{ v: B | ψ }` A value `v` of type `B` that satisfies `ψ`

`{ Int | v ≥ 0 }` A non-negative integer

[RKJ08] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.

Liquid Types

```
{ v: B | ψ }
```

A value `v` of type `B` that satisfies `ψ`

```
{ Int | v ≥ 0 }
```

A non-negative integer

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

A function that returns a list whose length is one plus the length of its input

[RKJ08] P. M. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI'08*.

Synthesis with Liquid Types

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

Synthesis with Liquid Types

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

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

Synthesis with Liquid Types

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

Reduce the synthesis problem to
finding an inhabitant of the target type

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

SYNTHESIS WITH LIQUID TYPES

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

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

Reduce the synthesis problem to finding an inhabitant of the target type

Use **type rules** to **reject** incomplete programs during the **search**

Synthesis with Liquid Types

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

Synthesis with Liquid Types

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

Type-Directed Synthesizer

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

SYNTHESIS WITH LIQUID TYPES

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

Type-Directed Synthesizer

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

Quadratic Complexity!
(#function calls)

ReSyn: Liquid Types + Linear Potentials

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

Potential: numeric

$$\{\ v : B \mid \psi\ \}^{\phi}$$

Refinement: boolean

ReSyn: Liquid Types + Linear Potentials

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

Potential: numeric

$$\{\ v\!:\ B\ |\ \psi\ \}^{\phi}$$

Refinement: boolean

$$\{\ \texttt{Int}\ |\ v \geq 0\ \}^{5\cdot v}$$

A non-negative integer carrying potential equal to 5 times of its value

ReSyn: Liquid Types + Linear Potentials

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

Potential: numeric

$$\{\ v\text{:}\ B\ |\ \psi\ \}^{\phi}$$

Refinement: boolean

$\{\ \mathtt{Int}\ |\ v \geq 0\ \}^{5 \cdot v}$

A non-negative integer carrying potential equal to 5 times of its value

$\mathtt{List}\ a^{\mathtt{ite}(v \geq 0, 1, 0)}$

A list of numbers carrying potential equal to #non-negative elements in it

ReSyn: Liquid Types + Linear Potentials

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.

Potential: numeric

$$\{\ v\!:\ B \mid \underbrace{\psi}_{\text{Refinement: boolean}}\ \}^{\phi}$$

Refinement: boolean

Type-checking is reduced to constraint solving in Presburger arithmetic.

`{ Int | v ≥ 0 }`$^{5\cdot v}$

A non-negative integer carrying potential equal to 5 times of its value

`List a`$^{\mathtt{ite(v\geq 0,1,0)}}$

A list of numbers carrying potential equal to #non-negative elements in it

Resource-Guided Synthesis

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

member : (z: a) -> (zs: SList a¹) ->
{ Bool | v = (z ∈ elems(zs) }
```

Resource-Guided Synthesis

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

member : (z: a) -> (zs: SList a¹) ->
{ Bool | v = (z ∈ elems(zs) }
```

RESOURCE-GUIDED SYNTHESIS

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

member : (z: a) -> (zs: SList a¹) ->
{ Bool | v = (z ∈ elems(zs) }
```

each element in the list carries **one unit** of potential,
thus the complexity must be **linear** in the list length

Resource-Guided Synthesis

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

RESOURCE-GUIDED SYNTHESIS

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

let rec common xs ys =
  ??
```

RESOURCE-GUIDED SYNTHESIS

```
  common : (xs: SList a¹) -> (ys: SList a¹) ->
{ SList 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 ??
```

RESOURCE-GUIDED SYNTHESIS

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


 let rec common xs ys =
   match xs with
   | [] -> []
   | x::xt ->
     if not (member x ys)    ys: List aᵖ <: List a¹
     then common xt ys       ys: List aᑫ <: List a¹
     else ??
```

Resource-Guided Synthesis

```
common : (xs: SList a^1) -> (ys: SList a^1) ->
{ SList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)    ys: List a^p <: List a^1      [p ≥ 1]
    then common xt ys       ys: List a^q <: List a^1      [q ≥ 1]
    else ??
```

Resource-Guided Synthesis

```
common : (xs: SList a¹) -> (ys: SList a¹) ->
{ SList 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 ??
```

Potential Sharing

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

RESOURCE-GUIDED SYNTHESIS

```
 common : (xs: SList a¹) -> (ys: SList a¹) ->
{ SList 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 ??
```

Potential Sharing

$[1 \geq p+q]$

ys: List a^p <: List a^1 $[p \geq 1]$

ys: List a^q <: List a^1 $[q \geq 1]$

Resource-Guided Synthesis

```
common : (xs: SList a¹) -> (ys: SList a¹) ->
{ SList 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 ??
```

Potential Sharing

ys: List a^p <: List a^1

ys: List a^q <: List a^1

[1 ≥ p+q]

[p ≥ 1]

[q ≥ 1]

Infeasible!

Resource-Guided Synthesis

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

```
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)
```

Example: List Append

ReSyn - resource-guided program synthesis

Triple | List Compress | List Intersect | List Insert | List Range | More ReSyn

```

-- | Let's ask ReSyn to generate a function
-- that returns a list three times the size of the input list `xs`,
-- and it only allowed two linear traversals over `xs`.
-- Try uncommenting different versions of the append
-- and observe how ReSyn associates the calls to append differently
-- to make sure they only perform two traversals of `xs`.
-- Also, try adding `-r=False` in the command line to disable resource analysis,
-- and observe that calls are always associated to the right.
triple :: xs: List {a| |2} -> {List a | len _v == 3 * (len xs)}
triple = ??

-- | A regular version of append.
-- | Its type signature requires 1 unit of potential for every element of `xs`,
-- | indicating that it performs a linear traversal over its first argument.
-- append :: xs: List {a| |1} -> ys: List a -> {List a | len _v == len xs + len ys}

-- | An append-and-swap function.
-- | Its type signature requires 1 unit of potential for every element of `ys`,
-- | indicating that it performs a linear traversal over its second(!) argument.
```

Run... -m 0

```
triple = \xs . appendSwap (appendSwap xs
                              xs) xs
```

EXAMPLE: LIST APPEND

ReSyn - resource-guided program synthesis

Triple | List Compress | List Intersect | List Insert | List Range | More ReSyn

```
-- | Let's ask ReSyn to generate a function
-- that returns a list three times the size of the input list `xs`,
-- and it only allowed two linear traversals over `xs`.
-- Try uncommenting different versions of the append
-- and observe how ReSyn associates the calls to append differently
-- to make sure they only perform two traversals of `xs`.
-- Also, try adding `-r=False` in the command line to disable resource analysis,
-- and observe that calls are always associated to the right.
triple :: xs: List {a| |2} -> {List a | len _v == 3 * (len xs)}
triple = ??

-- | A regular version of append.
-- | Its type signature requires 1 unit of potential for every element of `xs`,
-- | indicating that it performs a linear traversal over its first argument.
-- append :: xs: List {a| |1} -> ys: List a -> {List a | len _v == len xs + len ys}

-- | An append-and-swap function.
-- | Its type signature requires 1 unit of potential for every element of `ys`,
-- | indicating that it performs a linear traversal over its second(!) argument.
```

Run... -m 0

```
triple = \xs . appendSwap (appendSwap xs
                          xs) xs
```

OUTLINE

- ☑ Automatic Amortized Resource Analysis
- ☑ Type-Guided Worst-Case Input Generation
- ☑ Resource-Guided Program Synthesis