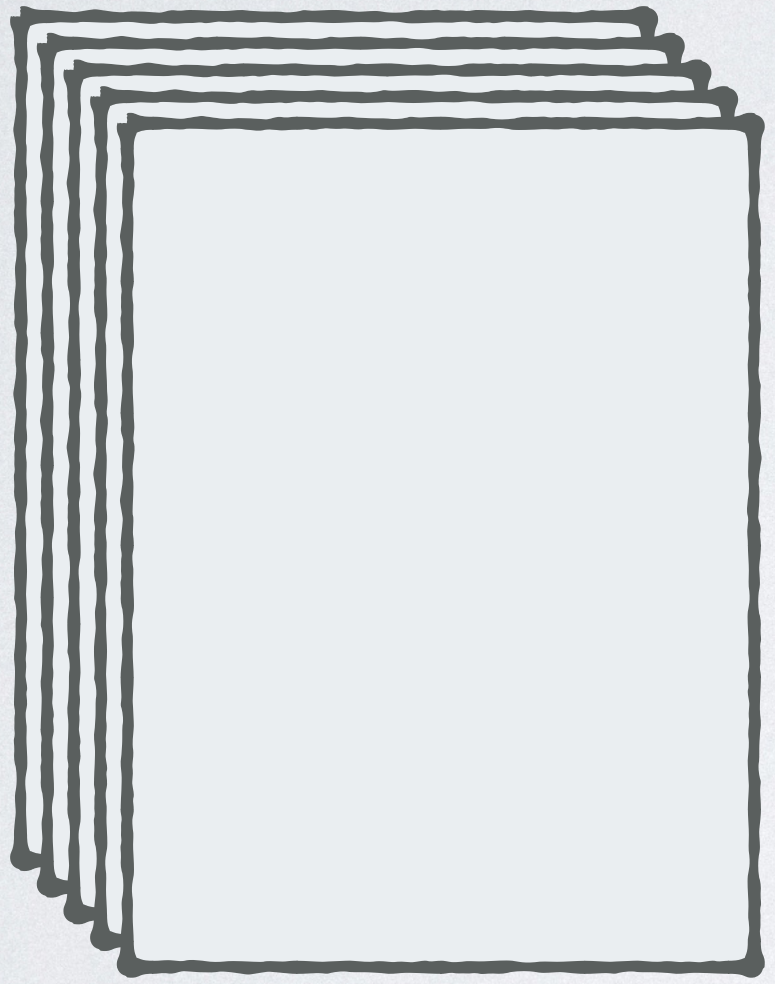
Type-Guided Worst-Case Input Generation

Di Wang, Jan Hoffmann

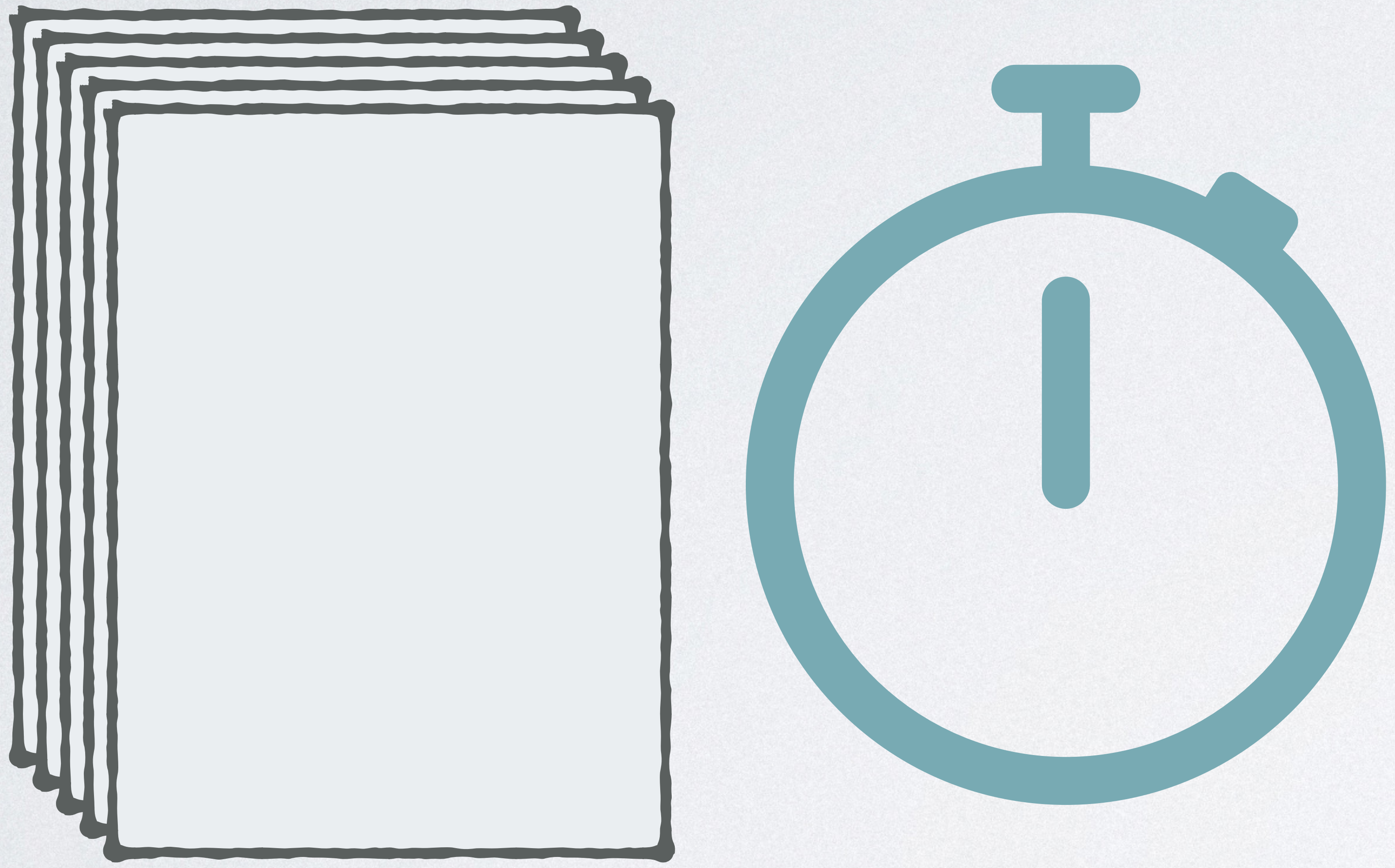
Carnegie Mellon University

Resource Analysis



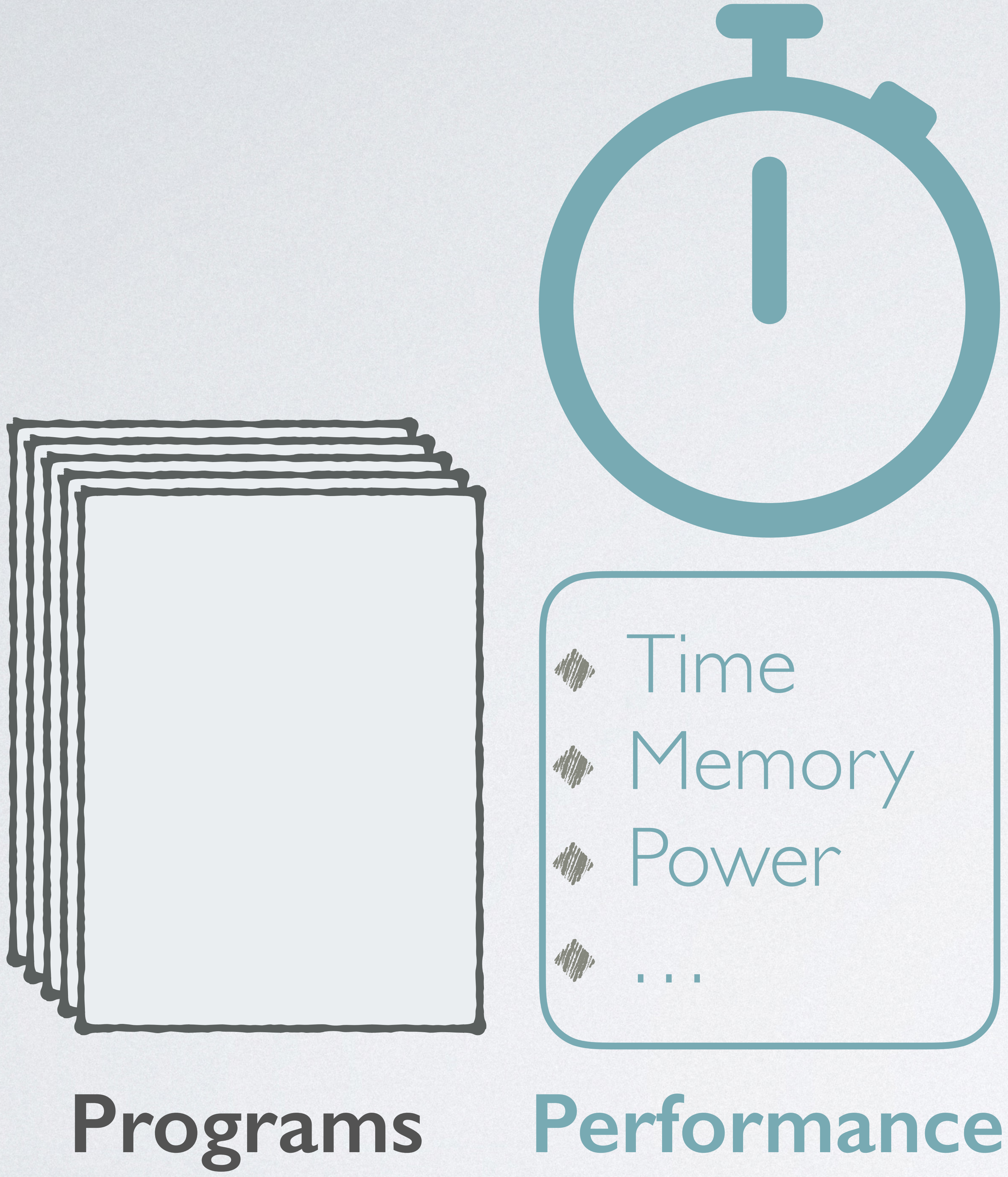
Programs

Resource Analysis



Programs **Performance**

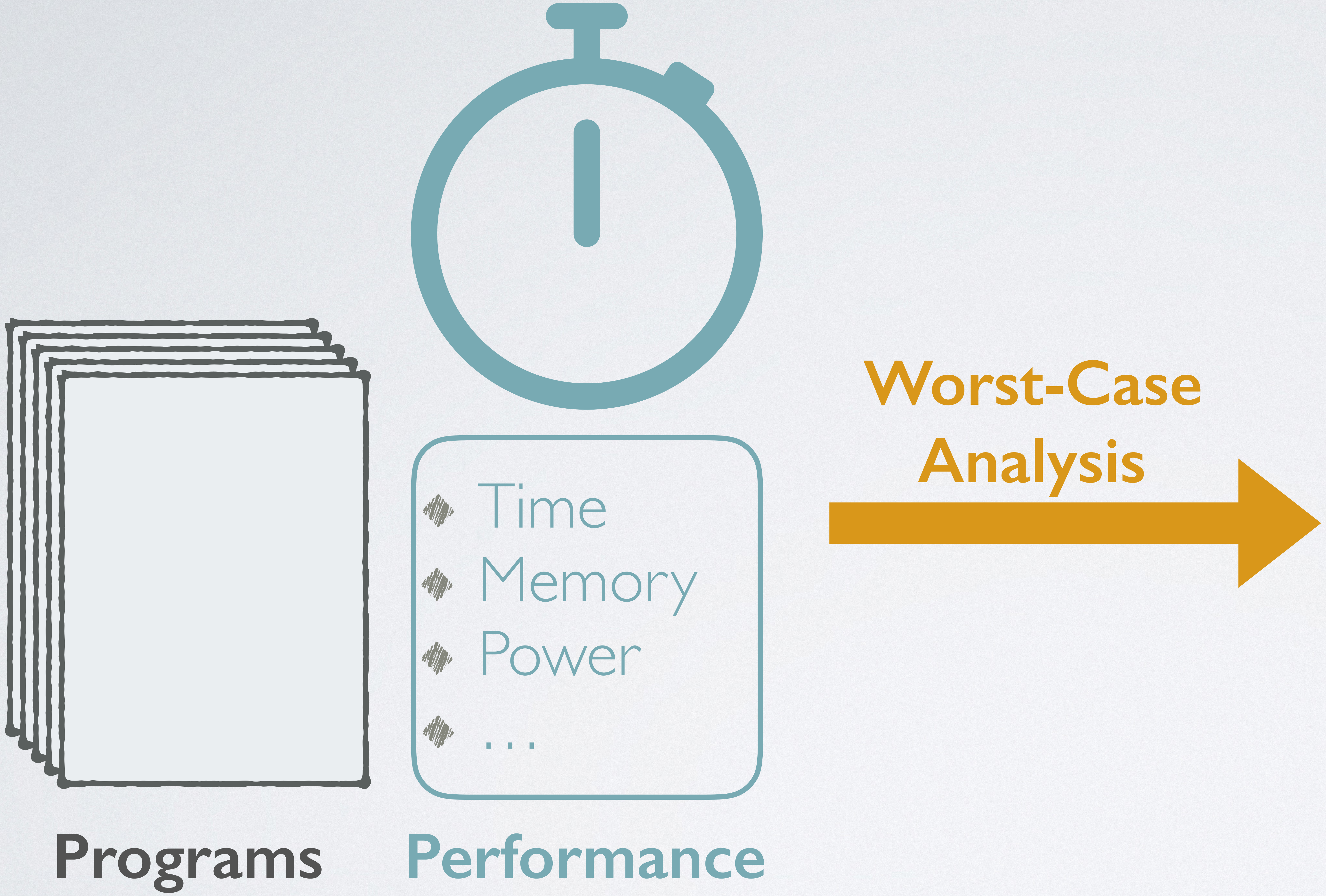
Resource Analysis



- Time
- Memory
- Power
- …

Programs **Performance**

Resource Analysis



- Time
- Memory
- Power
- …

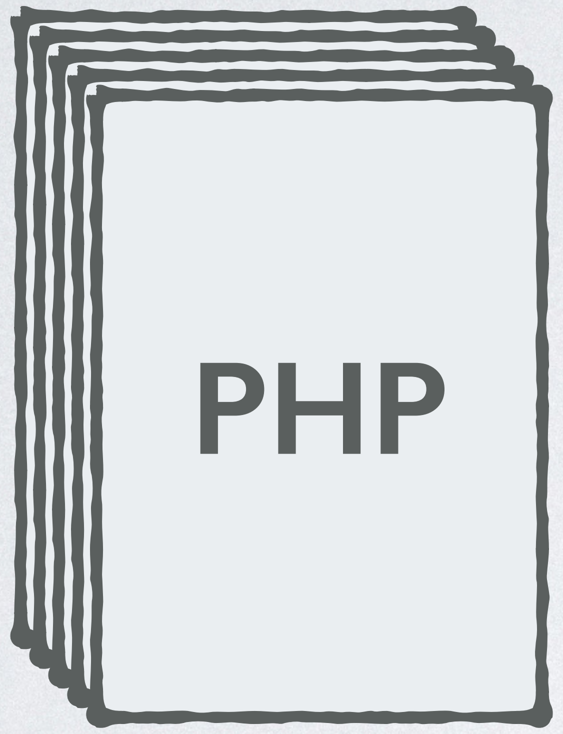
Programs

Performance

Worst-Case Analysis

- Performance bottlenecks
- Algorithmic complexity vulnerabilities
- Timing side channels

Example of Worst-Case Analysis

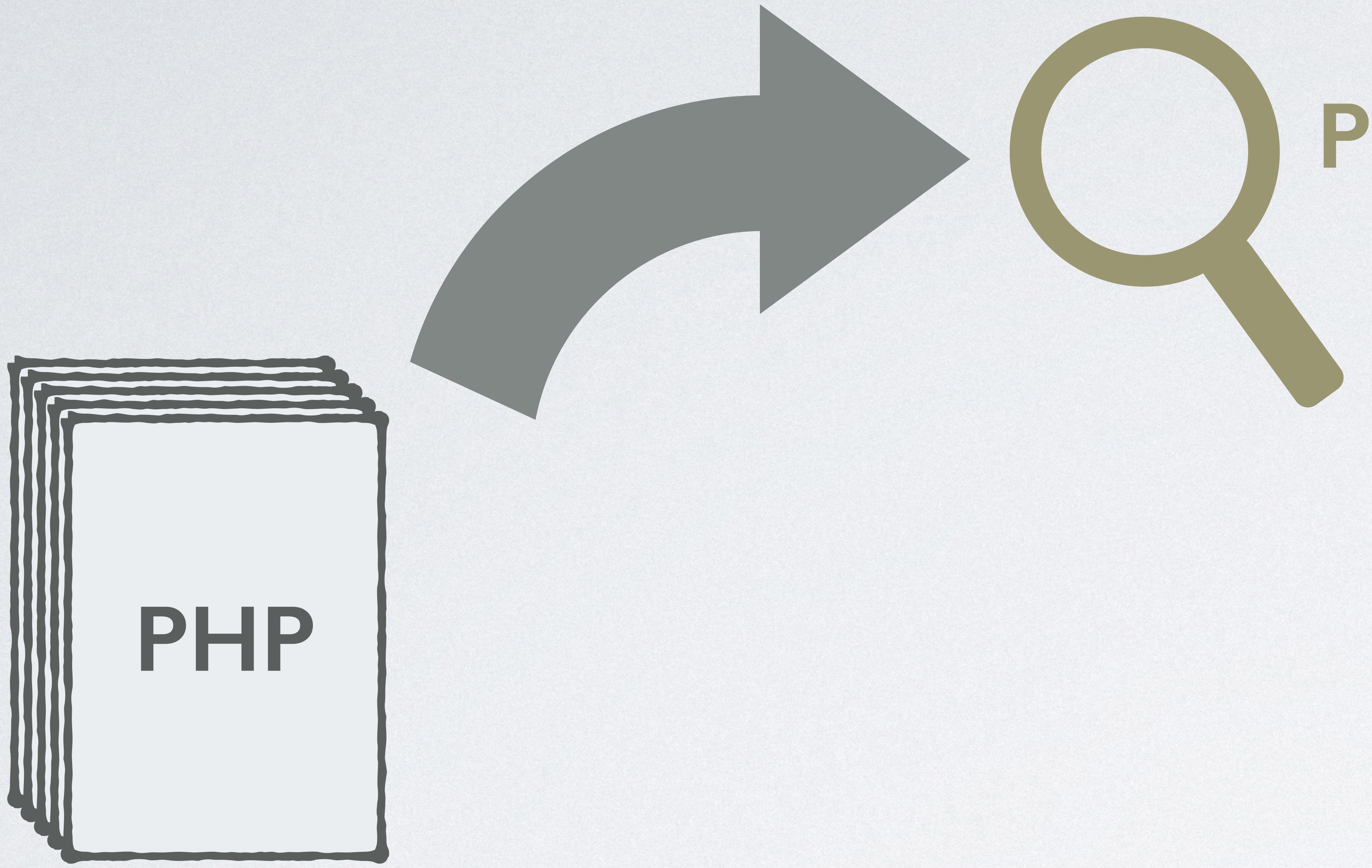


[1] CVE - CVE-2011-4885. Available on: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885.

[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



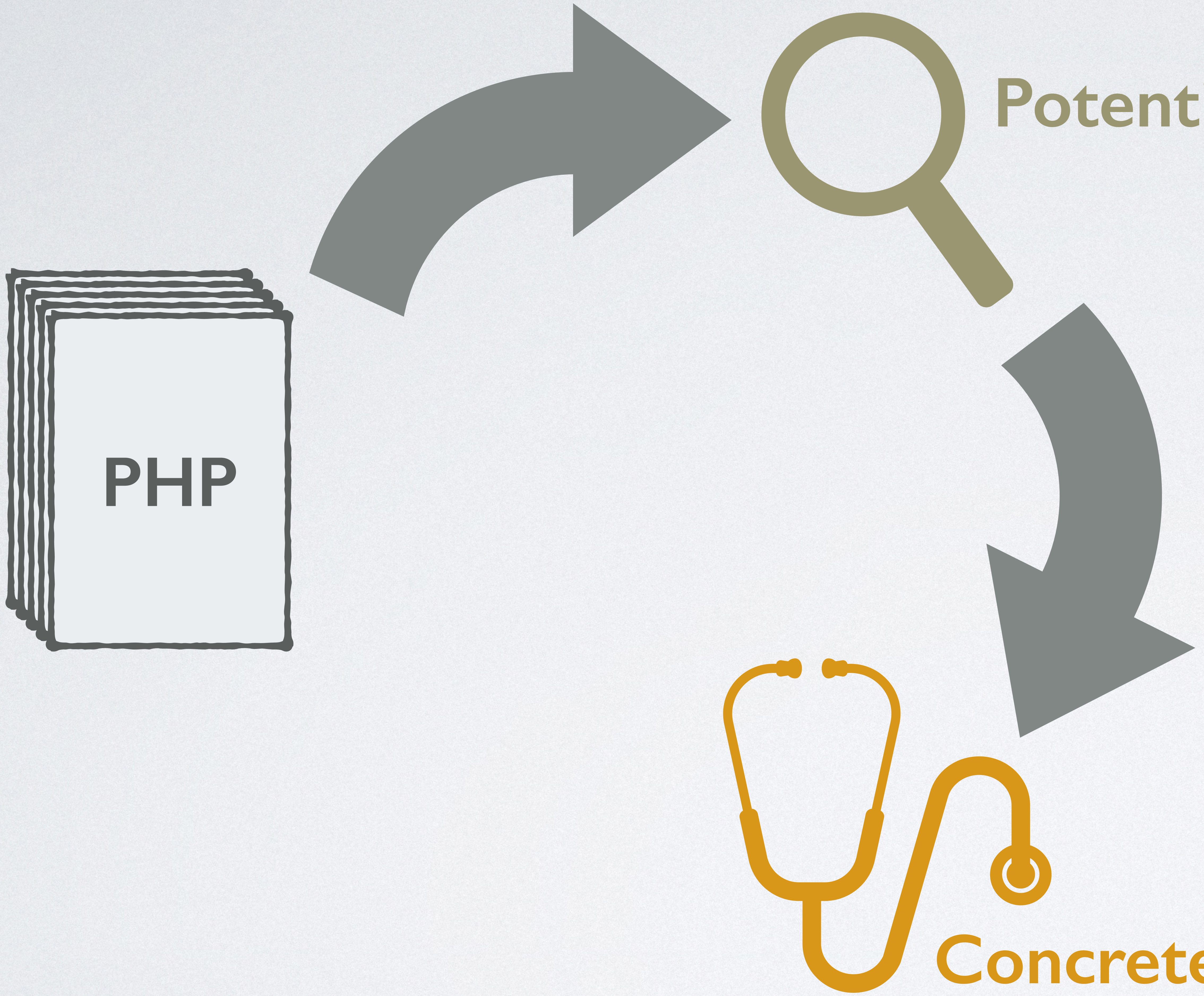
Potential Denial-of-Service attack[1]

[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



PHP

Potential Denial-of-Service attack[1]

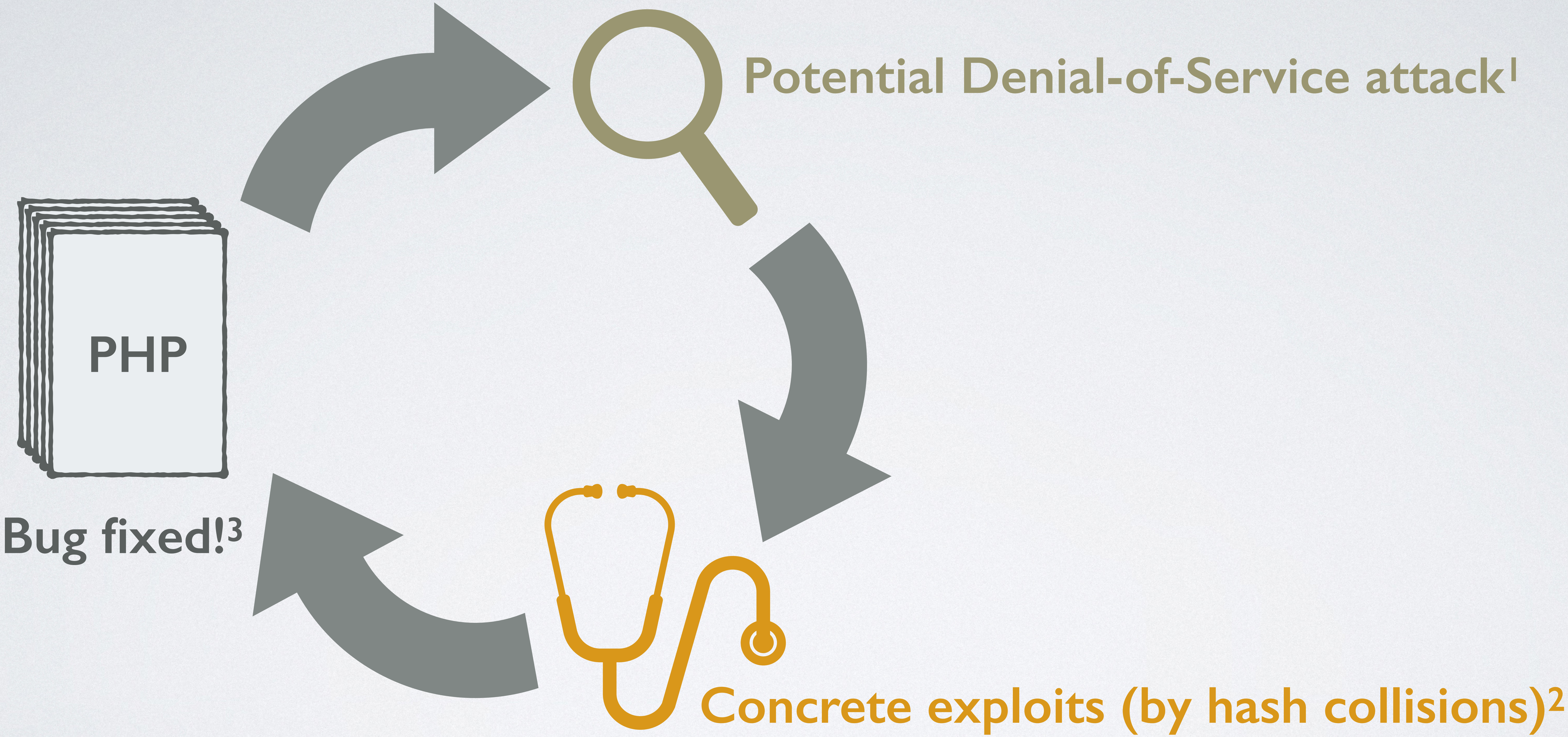
Concrete exploits (by hash collisions)[2]

[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]

PHP

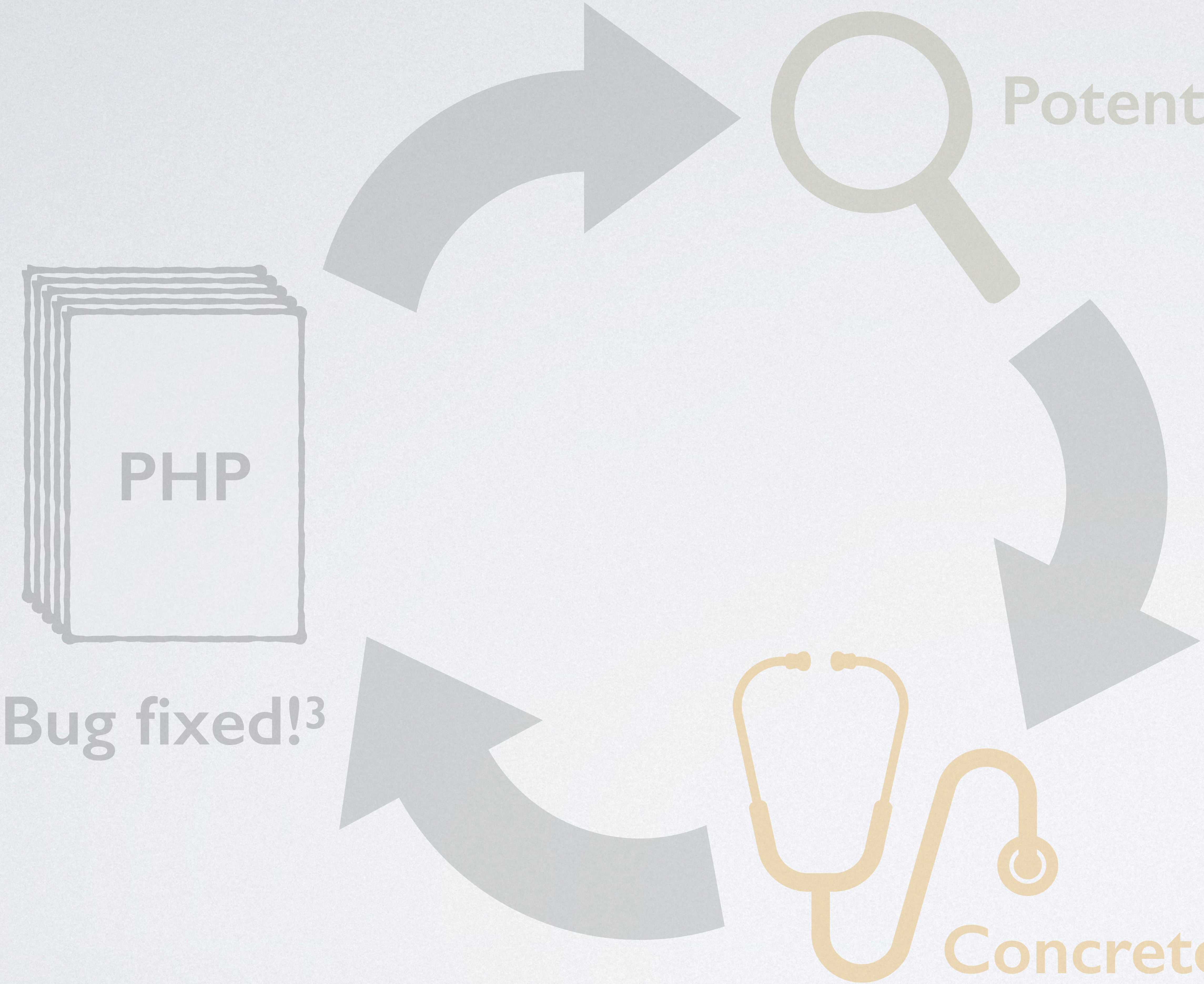
Bug fixed![3]

[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



Potential Denial-of-Service attack[1]

Concrete exploits (by hash collisions)[2]

PHP

Bug fixed![3]

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.

Contributions

- A **type-guided** **worst-case input** generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Contributions

- A **type-guided** **worst-case input** generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

CONTRIBUTIONS

- A **type-guided** **worst-case input** generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Resource Aware ML (RaML)

CONTRIBUTIONS

- A **type-guided** **worst-case input** generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Resource Aware ML (RaML)

CONTRIBUTIONS

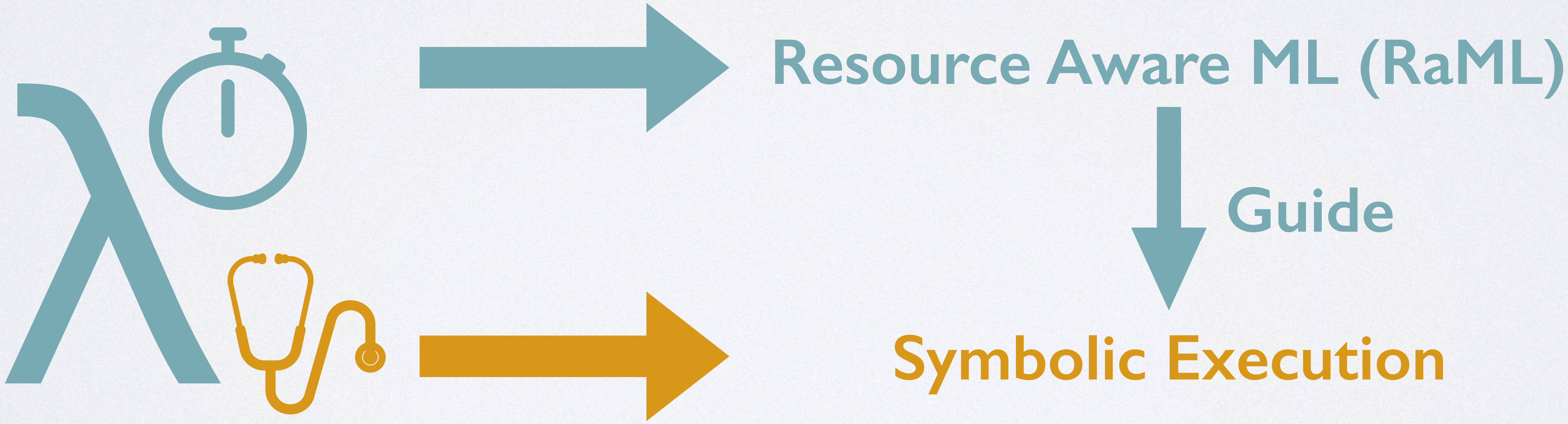
- A type-guided worst-case input generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability

Resource Aware ML (RaML)

Symbolic Execution

CONTRIBUTIONS

- A **type-guided** **worst-case input** generation algorithm
- Proof of soundness and relative completeness
- Heuristics to improve scalability



Resource Aware ML (RaML)

Guide

Symbolic Execution

Overview

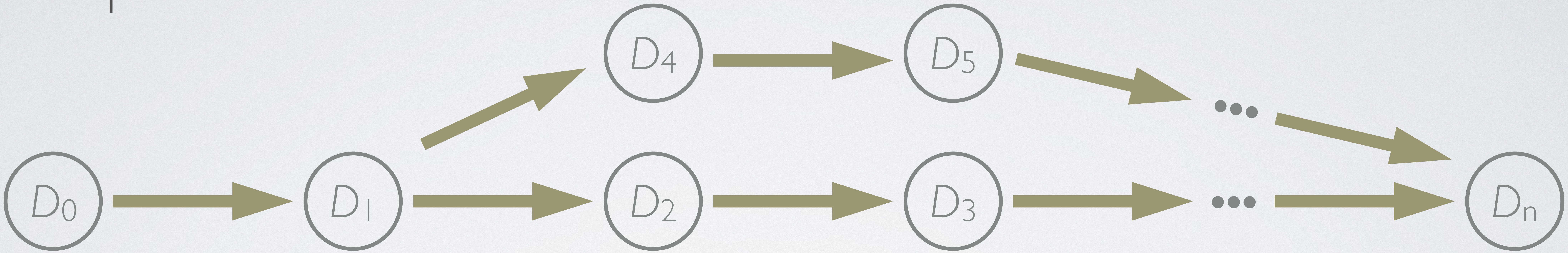
- ☑ Motivation
- □ Resource Aware ML (RaML)
- □ Type-Guided Worst-Case Input Generation
- □ Evaluation

Amortized Resource Analysis

- The potential method

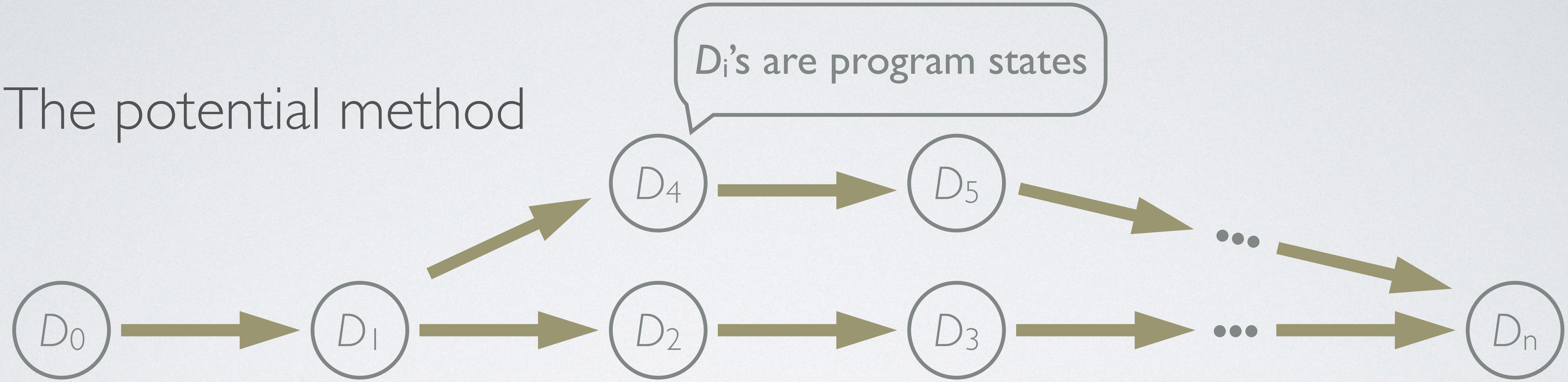
Amortized Resource Analysis

- The potential method



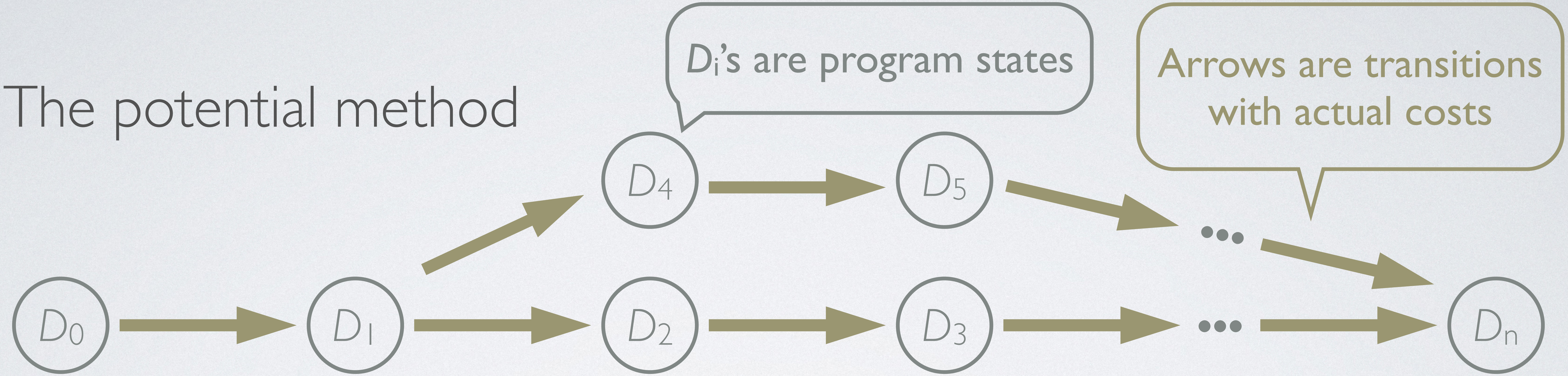
Amortized Resource Analysis

- The potential method



Amortized Resource Analysis

- The potential method

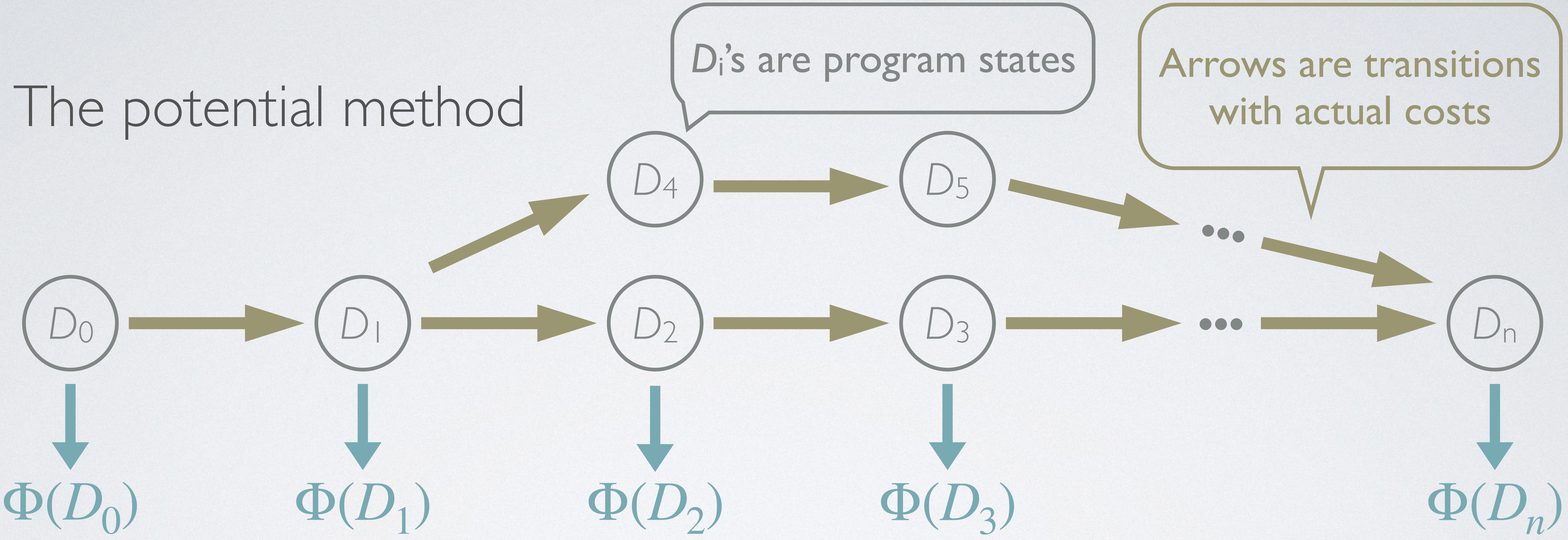


D_i's are program states

Arrows are transitions with actual costs

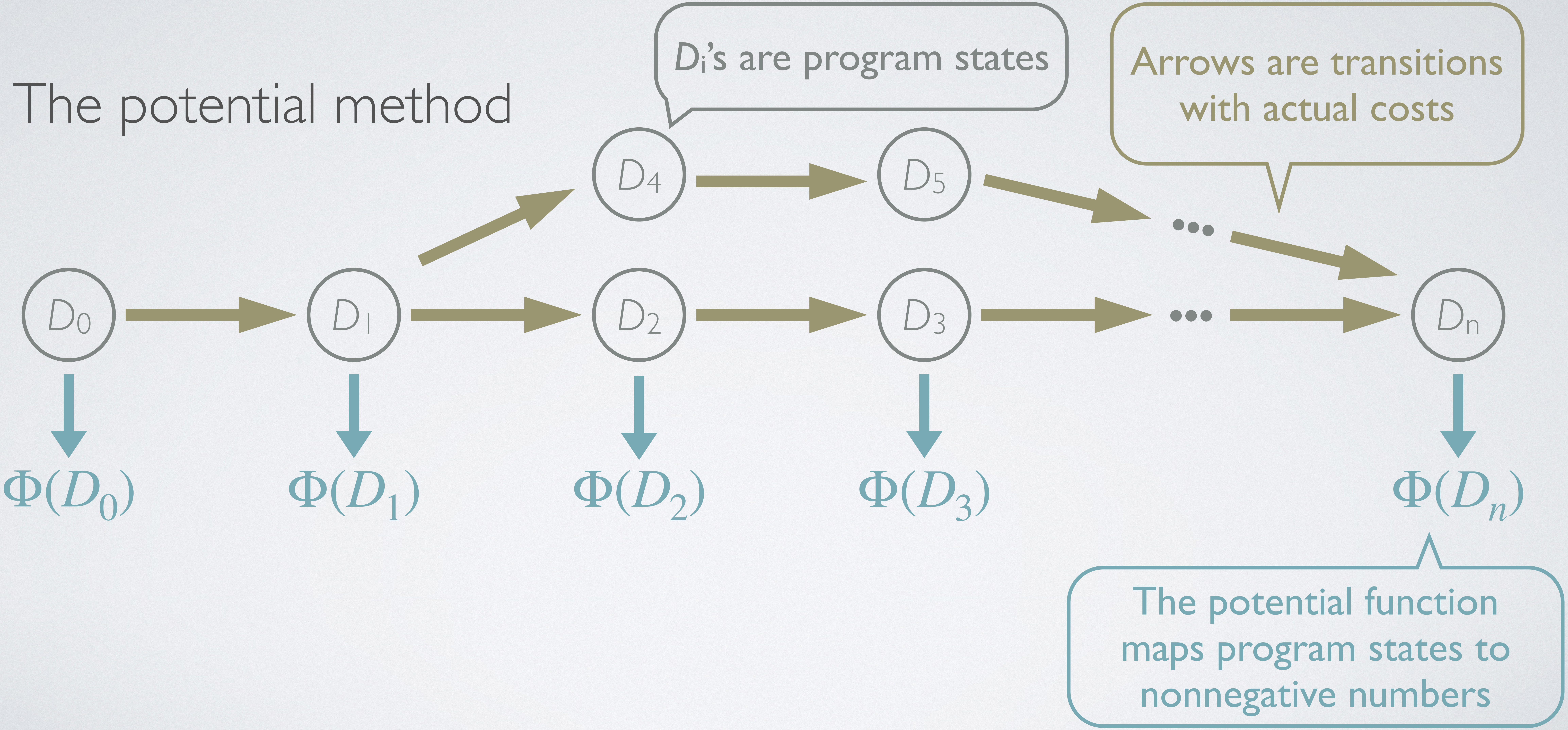
AMORTIZED RESOURCE ANALYSIS

- The potential method



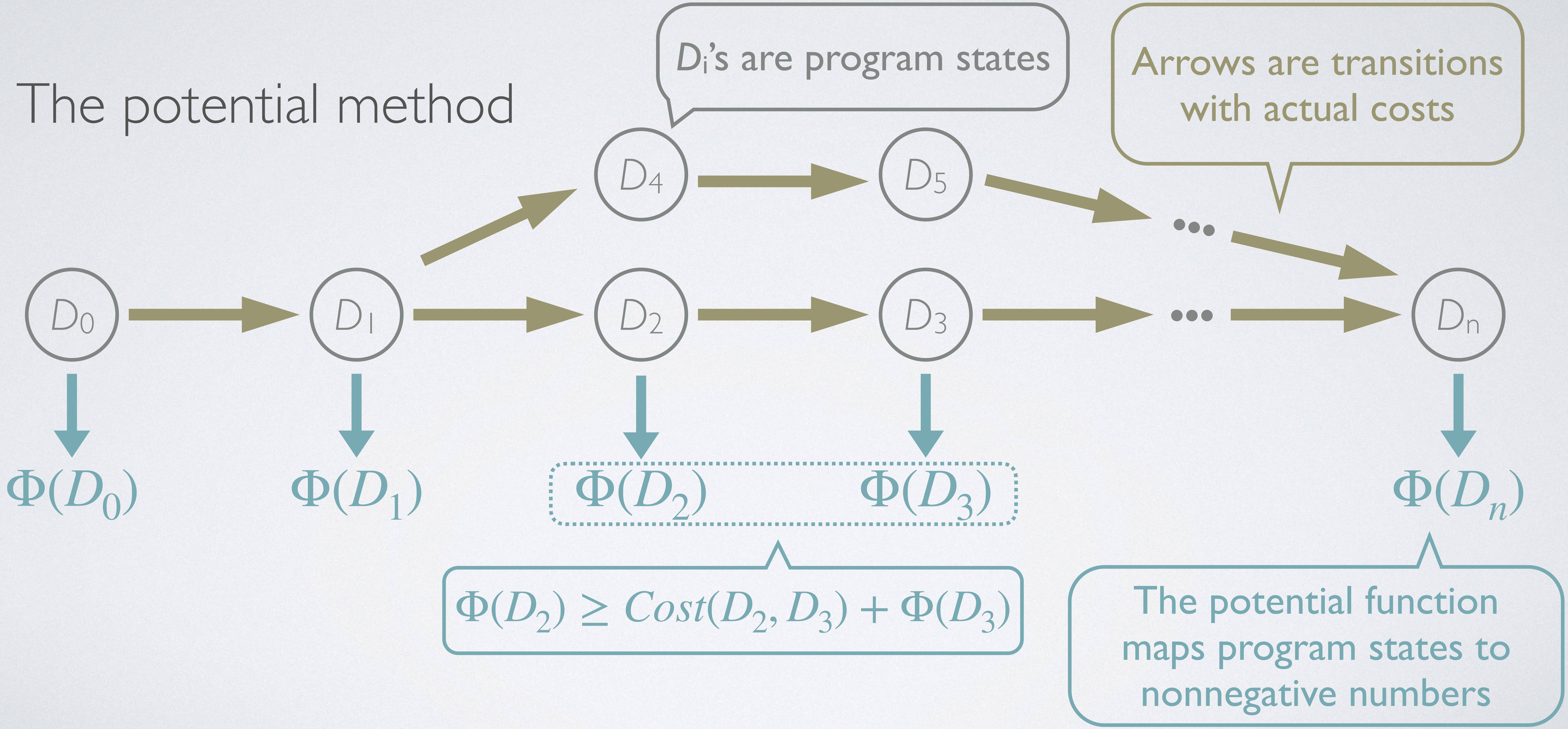
Amortized Resource Analysis

- The potential method


D_i's are program states
Arrows are transitions with actual costs
D_0
D_1
D_2
D_3
D_4
D_5
...
...
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$
The potential function maps program states to nonnegative numbers

Amortized Resource Analysis

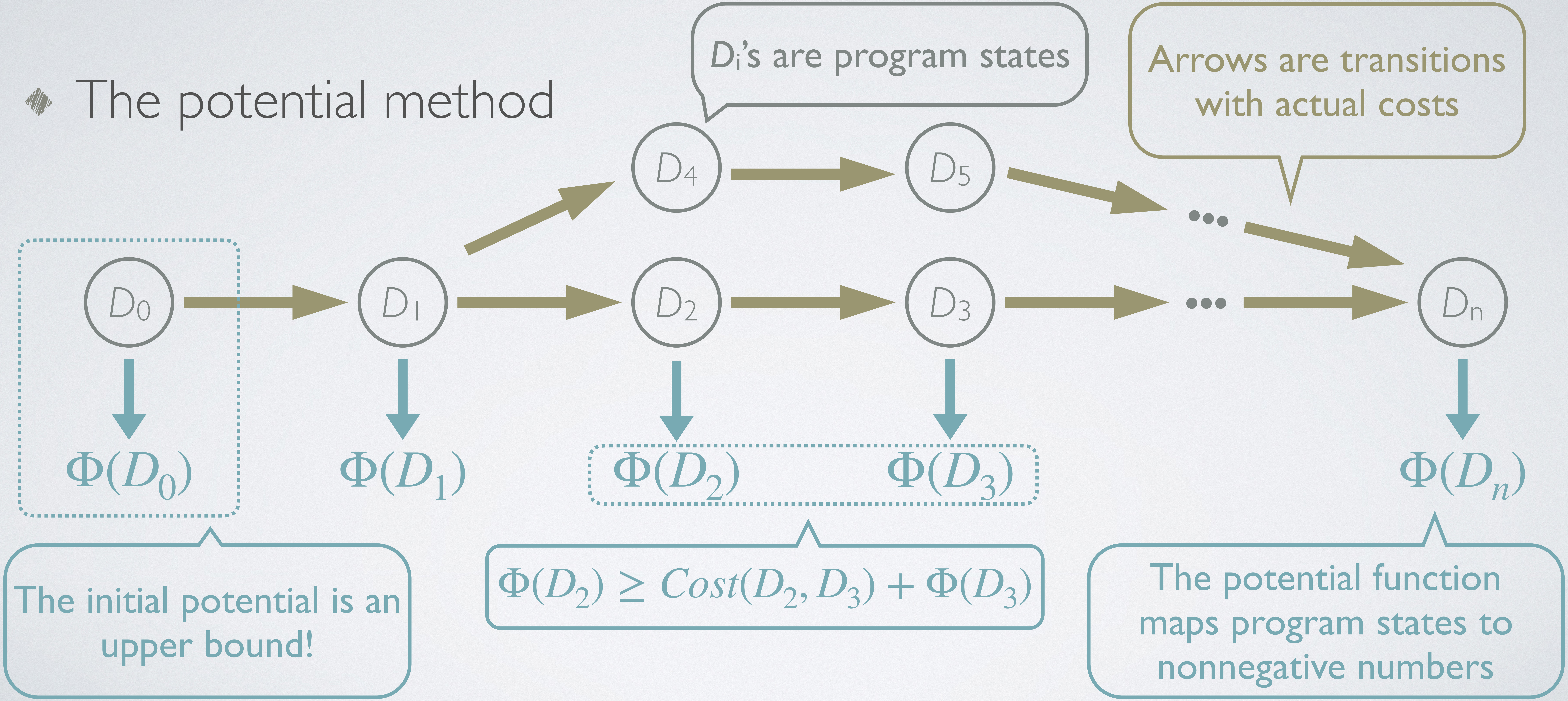
- The potential method



D_i's are program states
Arrows are transitions with actual costs
D_0
D_1
D_2
D_3
D_4
D_5
…
…
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$
$\Phi(D_2) \geq Cost(D_2, D_3) + \Phi(D_3)$
The potential function maps program states to nonnegative numbers

Amortized Resource Analysis

- The potential method



D_i's are program states
Arrows are transitions with actual costs
D_0
D_1
D_2
D_3
D_4
D_5
...
...
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$
The initial potential is an upper bound!
$\Phi(D_2) \geq Cost(D_2, D_3) + \Phi(D_3)$
The potential function maps program states to nonnegative numbers

Type-Based Analysis

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

- The **potential** at a program point is defined by a **static** annotation of data structures.

- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

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

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.

- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

$Cost = 2 \cdot |\ell| + 2$

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

Each of [], ::, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

TYPE-BASED ANALYSIS

$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

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

$\Phi_0 = 2 \cdot |\ell| + 2$

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

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

$$\Phi_0 = 2 \cdot |\ell| + 2$$

Cost = 2

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.

- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

$\Phi_1 = 2 \cdot |xs| + 4$

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

$$\boxed{L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})}$$

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

$$\Phi_2 = 2 \cdot |xs'| + 6$$

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

Type-Based Analysis

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

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

$\Phi_2 = 2 \cdot |xs'| + 6$

Cost = 4

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

TYPE-BASED ANALYSIS

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

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

$\Phi_2 = 2 \cdot |xs'| + 6$

$\Phi_3 = 2 \cdot |xs'| + 2$

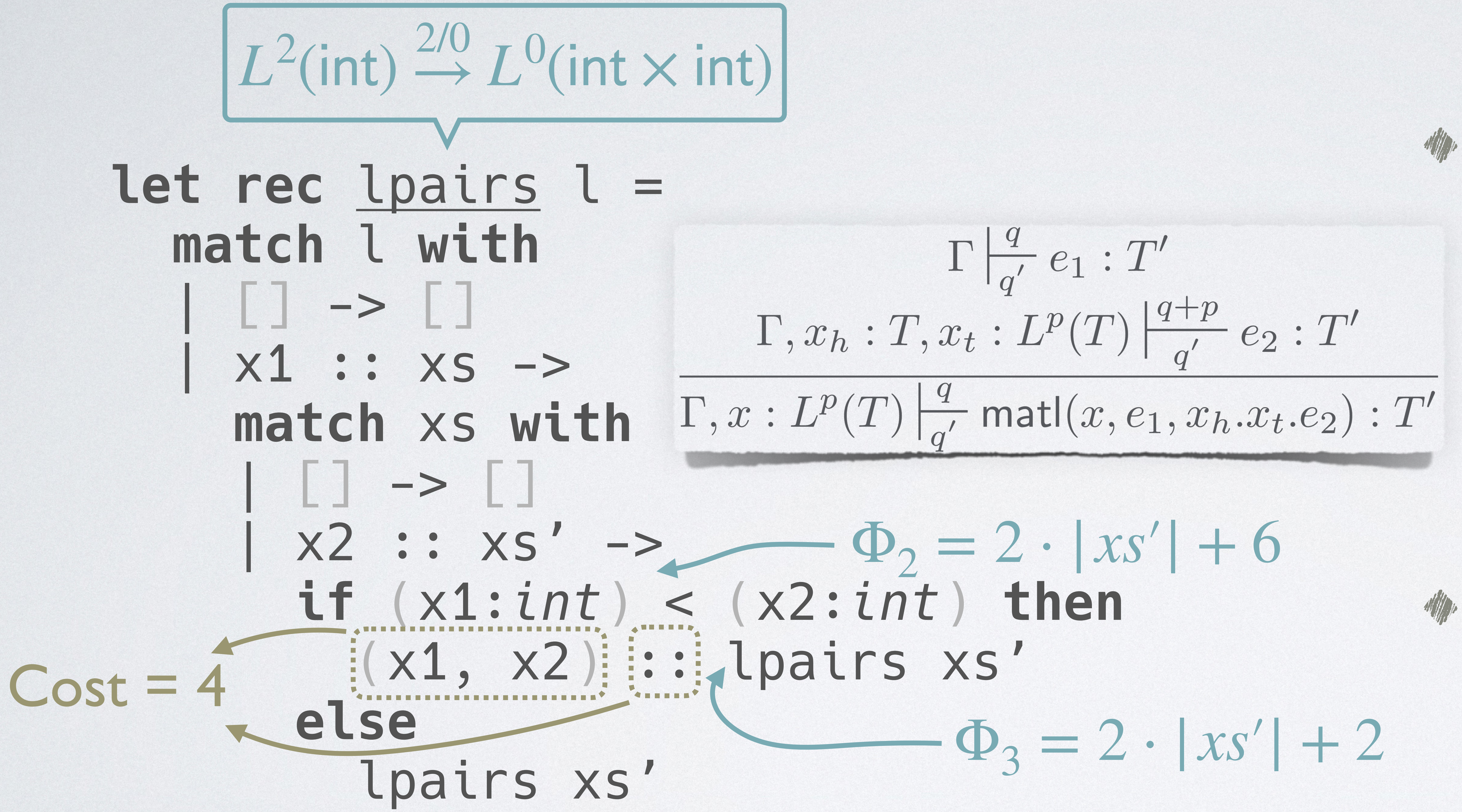
Cost = 4

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.

- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

TYPE-BASED ANALYSIS



$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

$$\frac{\begin{array}{c}\Gamma \overset{q}{\underset{q'}{\vdash}} e_1 : T' \\ \Gamma, x_h : T, x_t : L^p(T) \overset{q+p}{\underset{q'}{\vdash}} e_2 : T'\end{array}}{\Gamma, x : L^p(T) \overset{q}{\underset{q'}{\vdash}} \mathsf{matl}(x, e_1, x_h.x_t.e_2) : T'}$$

$\Phi_2 = 2 \cdot |xs'| + 6$

Cost = 4

$\Phi_3 = 2 \cdot |xs'| + 2$

Each of [ ], : :, (,) consumes 2 memory cells.

- The **potential** at a program point is defined by a **static** annotation of data structures.
- A list of length ***n*** annotated with a nonnegative number ***q*** has ***q·n*** units of potential.

OVERVIEW

- ☑ Motivation
- ☑ Resource Aware ML (RaML)
- □ Type-Guided Worst-Case Input Generation
- □ Evaluation

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

$$\underset{\text{symbolic environment}}{\gamma} \vdash \underset{\substack{\downarrow \\ \text{expression}}}{e} \Rightarrow \langle \psi, S \rangle$$

Symbolic Execution

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

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

symbolic environment

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 evaluation result

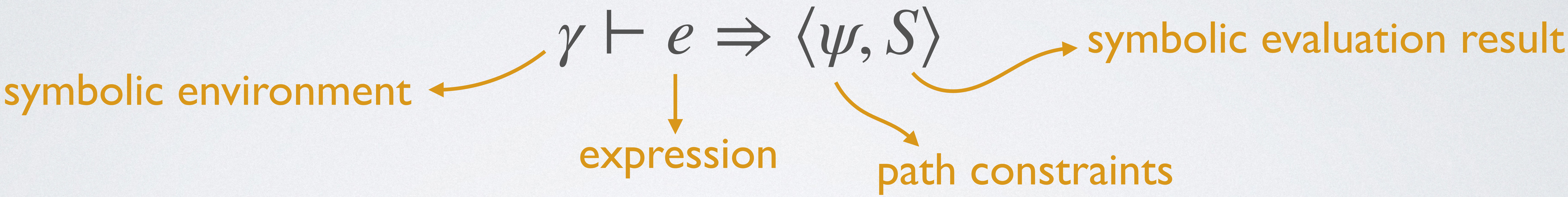
symbolic environment

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

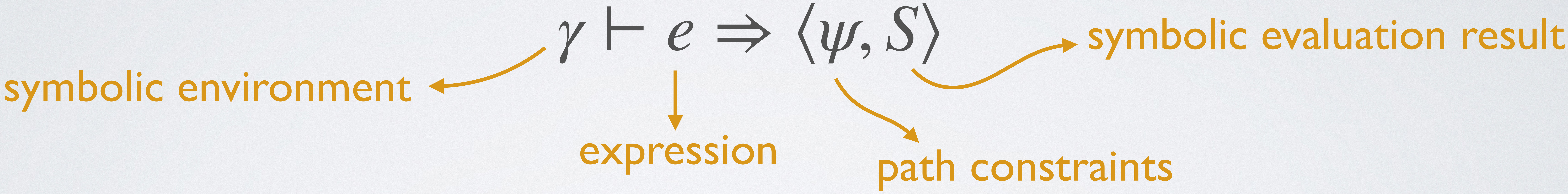
path constraints

symbolic evaluation result

- Symbolic execution rules for conditional expressions

Symbolic Execution

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



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

- Symbolic execution rules for conditional expressions

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

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

Symbolic Execution

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

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

$$\begin{aligned}&\ell \mapsto [\mathrm{int}^1, \mathrm{int}^2, \mathrm{int}^3, \mathrm{int}^4] \vdash \\ &\quad \mathrm{lpairs}\ \ell \Rightarrow \langle (\mathrm{int}^1 < \mathrm{int}^2) \wedge (\mathrm{int}^3 < \mathrm{int}^4), \\ &\qquad\qquad [(\mathrm{int}^1, \mathrm{int}^2), (\mathrm{int}^3, \mathrm{int}^4)] \rangle\end{aligned}$$

Symbolic Execution

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

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

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

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

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

Type-Guided Symbolic Execution

- **Nondeterminism** leads to state explosion

Then

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

Else

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

Type-Guided Symbolic Execution

- **Nondeterminism** leads to state explosion

Then

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

Else

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

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

Type-Guided Symbolic Execution

$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

Type-Guided Symbolic Execution

$$\boxed{L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})}$$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1 :: xs ->
    match xs with
    | [] -> []
    | x2 :: xs’ ->
      if (x1:int) < (x2:int) then
        (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

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1 :: xs ->
    match xs with
    | [] -> []
    | x2 :: xs’ ->
      if (x1:int) < (x2:int) then
        (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

$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

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

$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

Type-Guided Symbolic Execution

$$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$$

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

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

$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost = 4

Type-Guided Symbolic Execution

$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

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

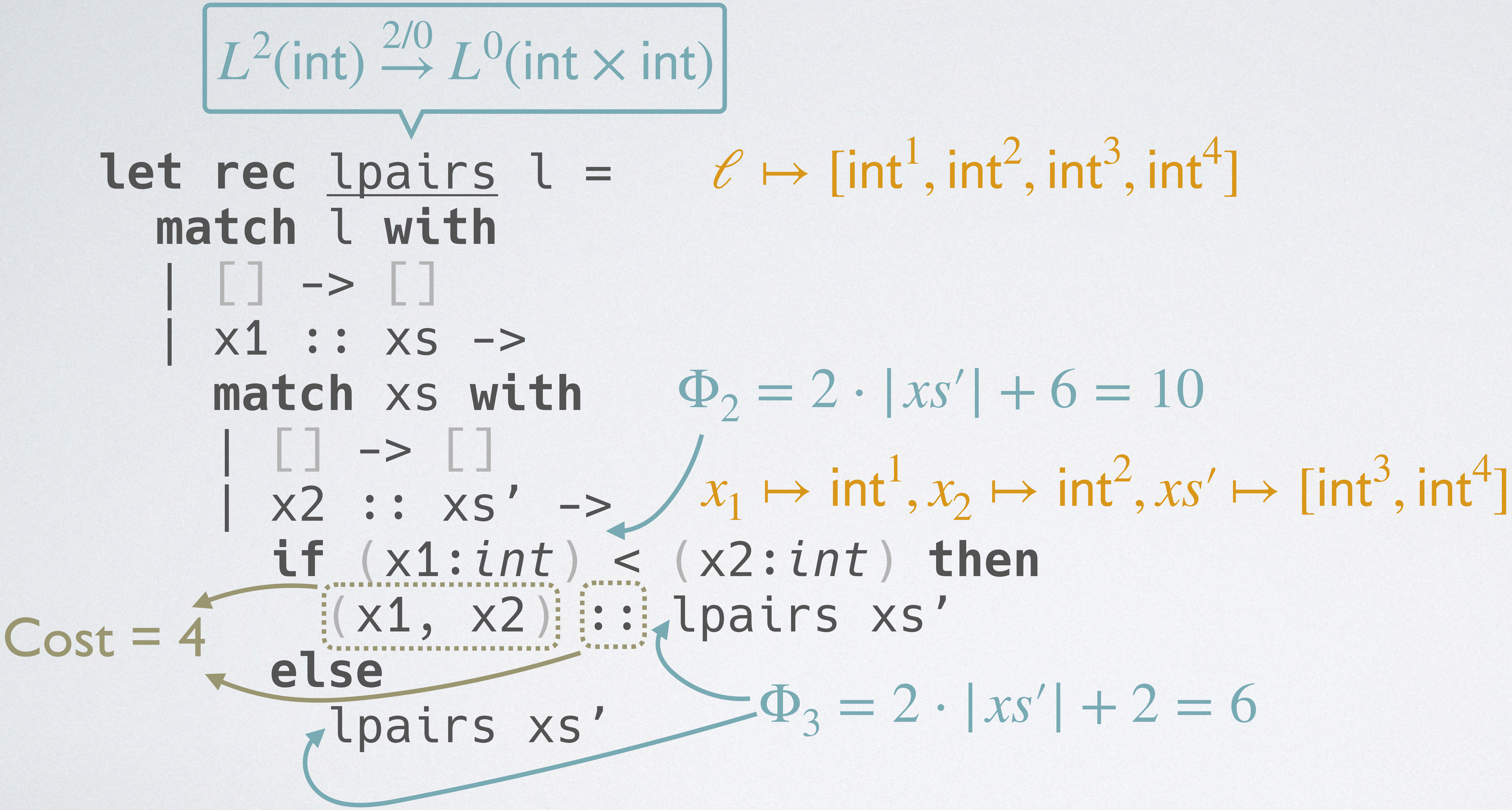
$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost = 4

$\Phi_3 = 2 \cdot |xs'| + 2 = 6$

Type-Guided Symbolic Execution



$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

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

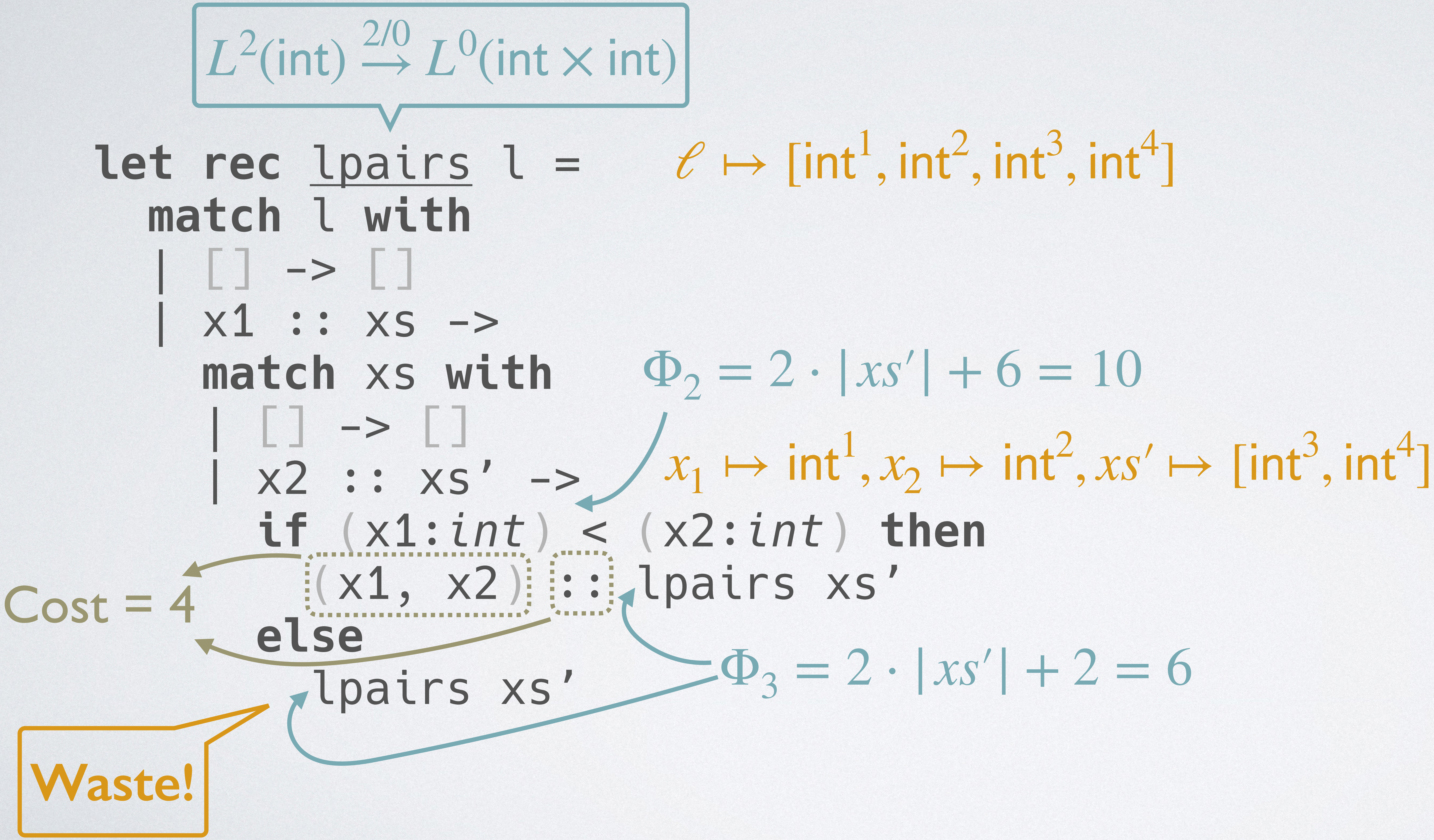
$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost = 4

$\Phi_3 = 2 \cdot |xs'| + 2 = 6$

Type-Guided Symbolic Execution



$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

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

$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost = 4

$\Phi_3 = 2 \cdot |xs'| + 2 = 6$

Waste!

Type-Guided Symbolic Execution

$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

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

$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

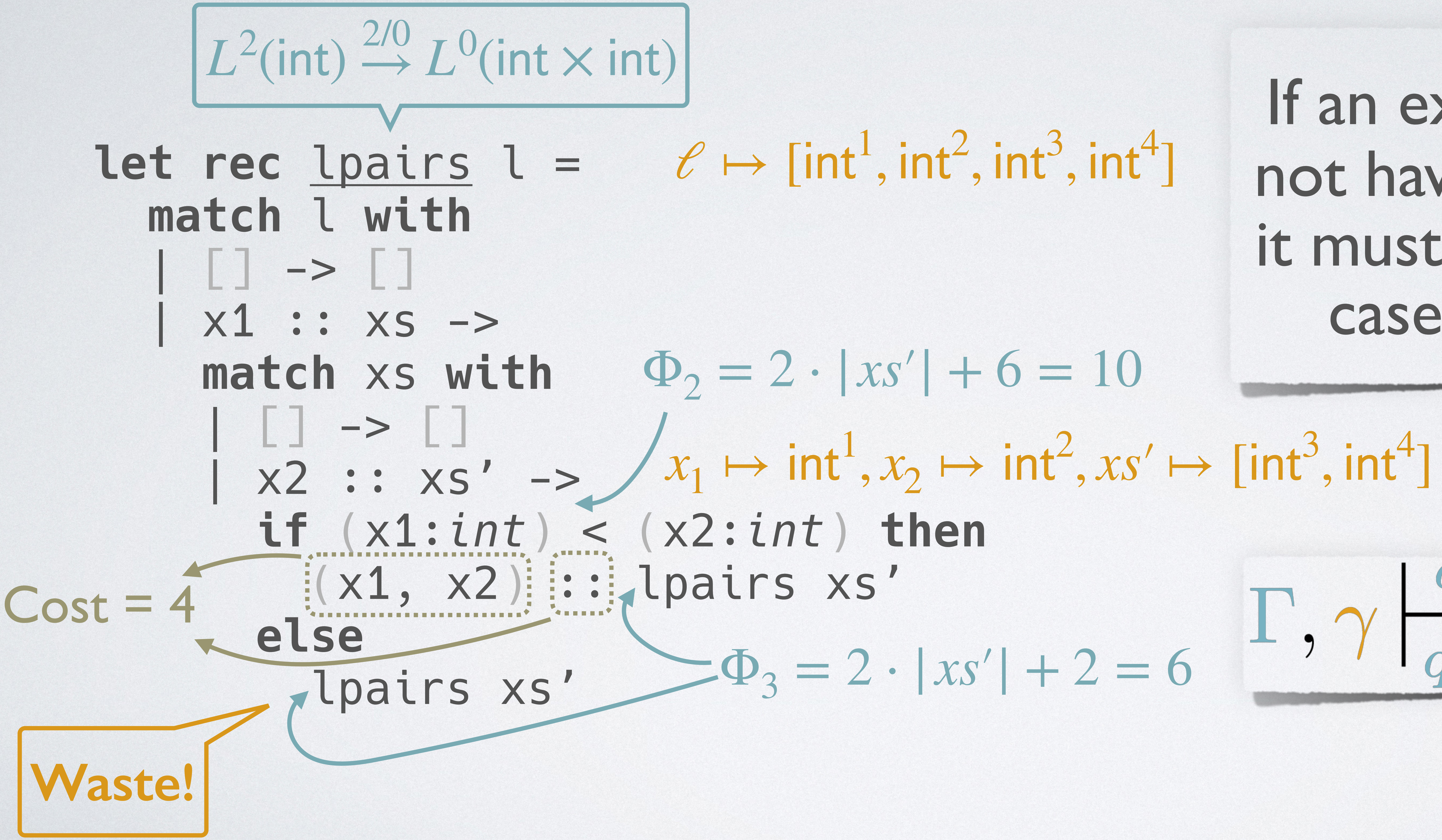
Cost = 4

$\Phi_3 = 2 \cdot |xs'| + 2 = 6$

Waste!

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

Type-Guided Symbolic Execution



$L^2(\text{int}) \xrightarrow{2/0} L^0(\text{int} \times \text{int})$

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

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

$\Phi_2 = 2 \cdot |xs'| + 6 = 10$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2, xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost = 4

$\Phi_3 = 2 \cdot |xs'| + 2 = 6$

Waste!

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

$$\Gamma, \gamma \vdash^{q}_{q'} e : T \Rightarrow \langle \psi, S \rangle$$

Soundness & Completeness

Soundness & Completeness

- **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).

Soundness & Completeness

- **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).

- **Relative completeness**: If there is an input of some given shape that causes the program to consume ***exactly*** the same amount of resource as the inferred ***upper bound*** (by RaML), then the algorithm is able to find ***a corresponding execution path***.

Speed up Input Generation

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

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

Speed up Input Generation

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

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

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

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?

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

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

Still Sound!

Speed up Input Generation

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

Then

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

Else

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

Still Sound!

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

Overview

- ☑ Motivation
- ☑ Resource Aware ML (RaML)
- ☑ Type-Guided Worst-Case Input Generation
- □ Evaluation

Implementation

- We implemented the 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 SMT solver Z3.

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

Summary

Type-Guided Symbolic Execution for Worst-Case Input Generation

Summary

Type-Guided Symbolic Execution for Worst-Case Input Generation

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

SUMMARY

TYPE-GUIDED SYMBOLIC EXECUTION FOR WORST-CASE INPUT GENERATION

- 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

SUMMARY

TYPE-GUIDED SYMBOLIC EXECUTION FOR WORST-CASE INPUT GENERATION

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

- Integrated with RaML
- Effective on 22 benchmark programs

Experimental Results

SUMMARY

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 SYMBOLIC EXECUTION FOR WORST-CASE INPUT GENERATION

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

- Integrated with RaML
- Effective on 22 benchmark programs

Experimental Results