Static analysis of Higher-order programs

Matthew Might

matt@might.net http://matt.might.net/

Why static analysis matters

Why static analysis matters

- Optimization
- Parallelism
- Verification

Why: Optimization

Why: Optimization

Hardware doubles performance every 18 months.

Moore's Law

Why: Optimization

Hardware doubles performance every 18 months.

Moore's Law

Compilers double performance every 18 years.

Proebsting's Law

Why: Optimization

Optimization is about
"freedom of expressions"

Liberating features

Liberating features

- Closures
- Virtual methods
- Laziness
- Coroutines
- Comprehensions
- Garbage collection
- Precise arithmetic
- Pattern matching
- Dynamic typing
- Monads
- Continuations
- Streams
- Polymorphism
- Bounds checks
- Exceptions
- Hybrid types

Liberating features

- Closures
- Virtual methods
- Laziness
- Coroutines
- Comprehensions
- Garbage collection
- Precise arithmetic
- Pattern matching
- Dynamic typing
- Monads
- Continuations
- Streams
- Polymorphism
- Bounds checks
- Exceptions
- Hybrid types

Why: Parallelism

Why: Parallelism

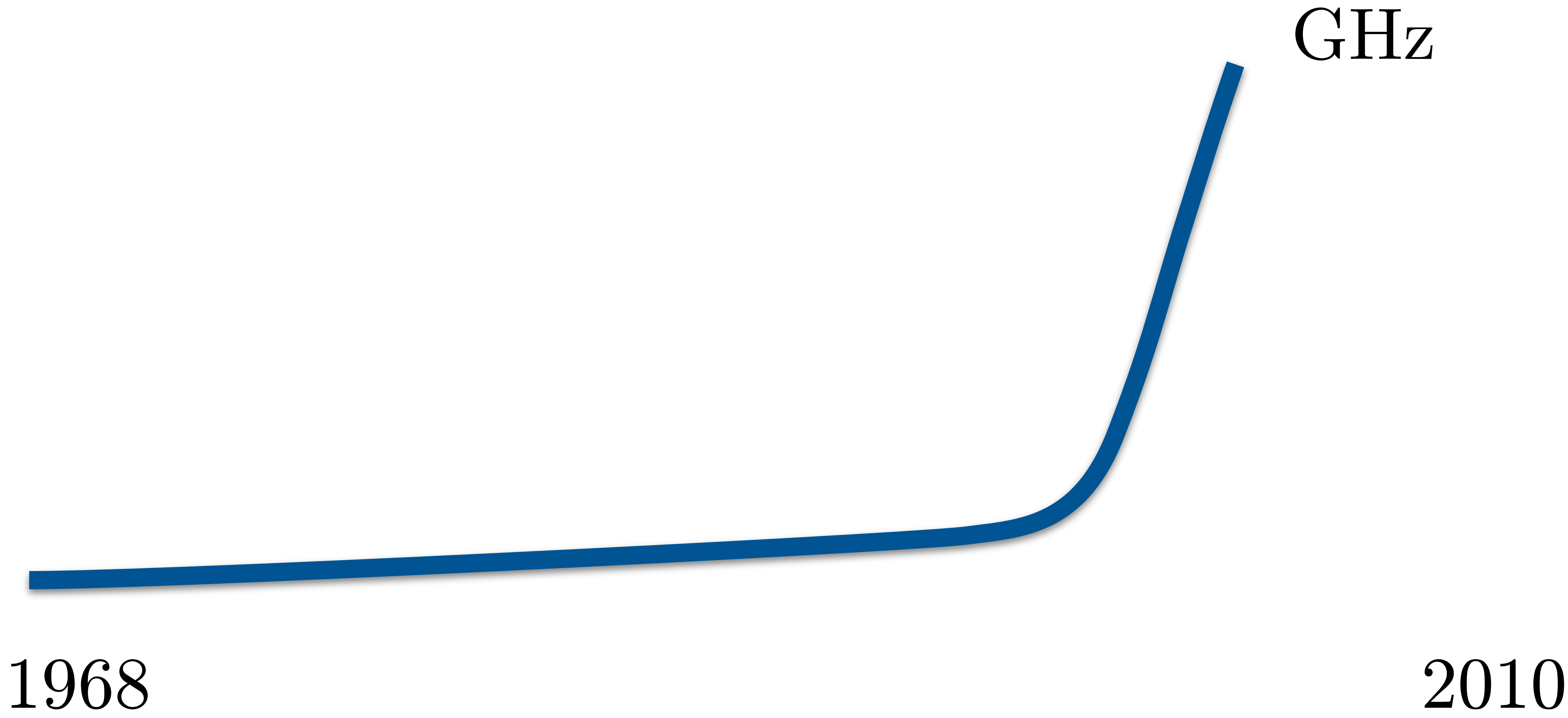
"...80 cores by 2011."

Paul Otellini, Intel

Why: Parallelism

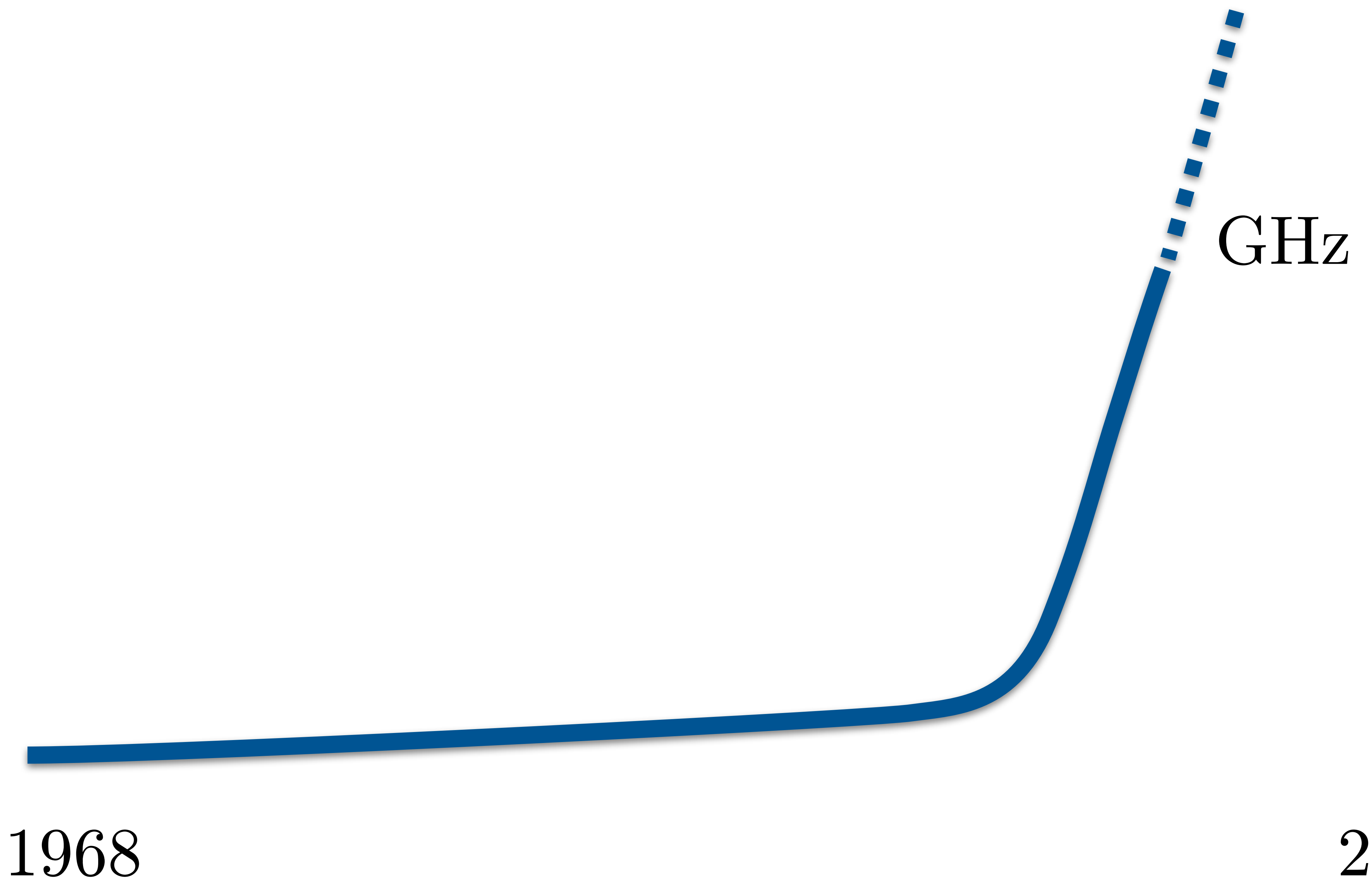
Source: Intel

Why: Parallelism



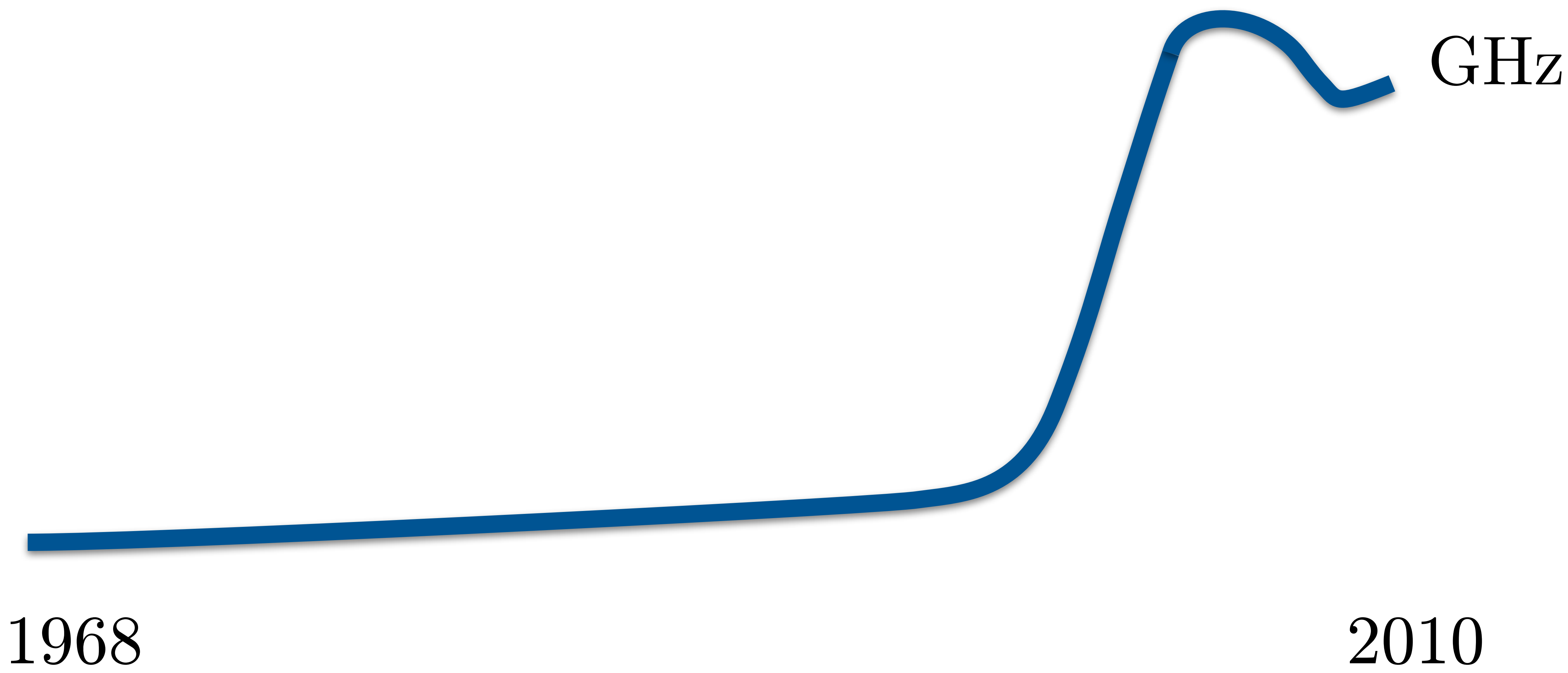
Source: Intel

Why: Parallelism



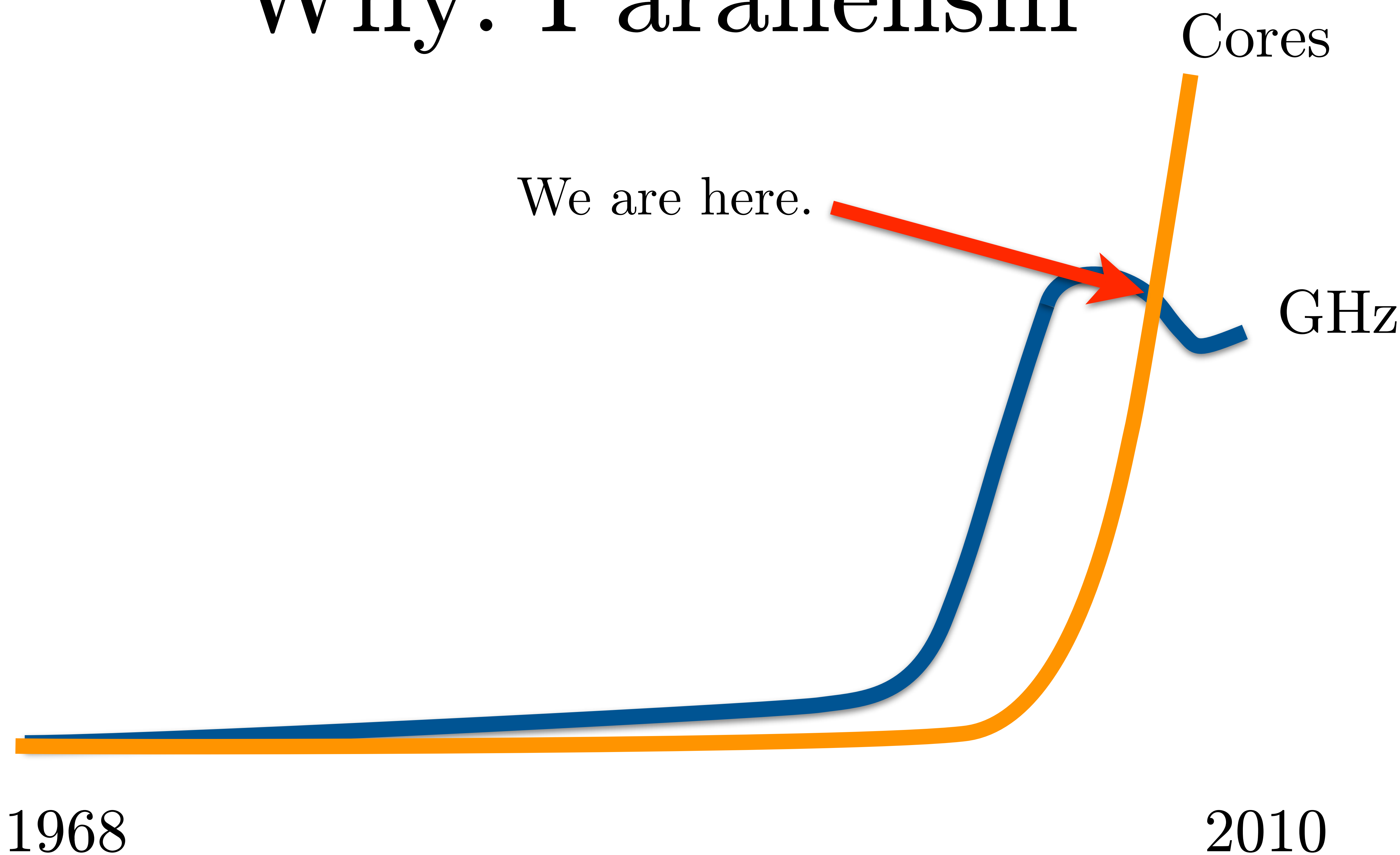
Source: Intel

Why: Parallelism



Source: Intel

Why: Parallelism



Source: Intel

Why: Parallelism

Static analysis can make:

- Sequential programs parallel
- Parallel programs correct
- Parallel paradigms feasible

Why: Parallelism

Static analysis can make:

- Sequential programs parallel
- Parallel programs correct
- Parallel paradigms feasible

Why: Verification

Why: Verification

Bugs cost U.S. economy \$60 billion annually.

NIST

\$80 billion in cyber-crime each year.

FBI

Why: Security matters

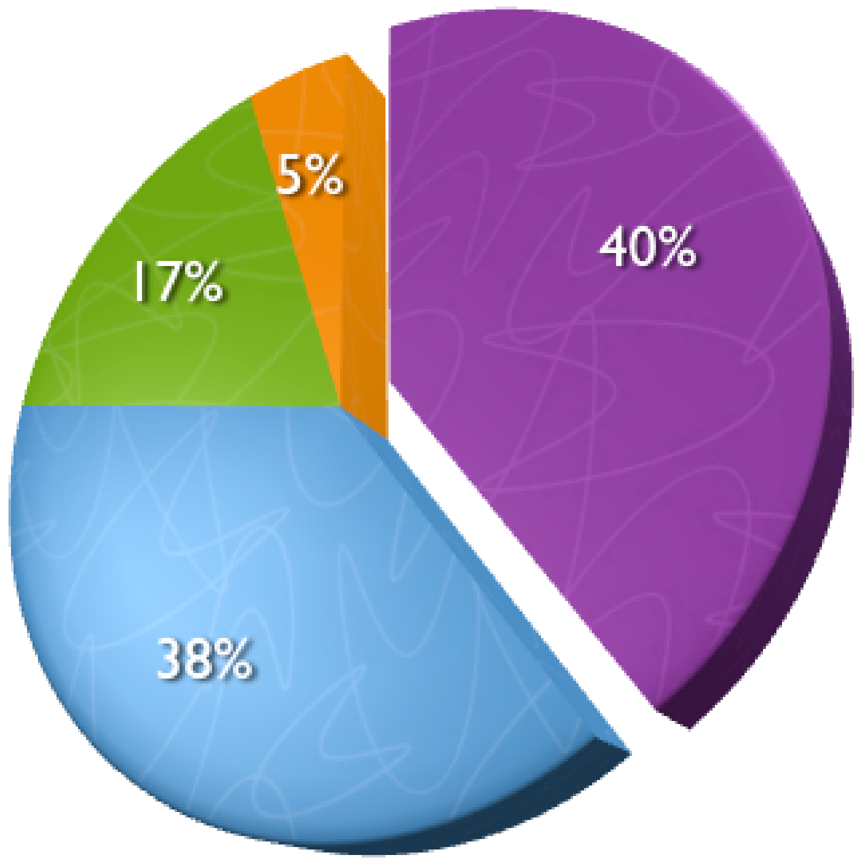
Vulnerabilities Fraud Dumb Employees DoS

Cost of cybercrime ($80bn)

Why: Security matters

Vulnerabilities Fraud Dumb Employees DoS

Cost of cybercrime ($80bn)

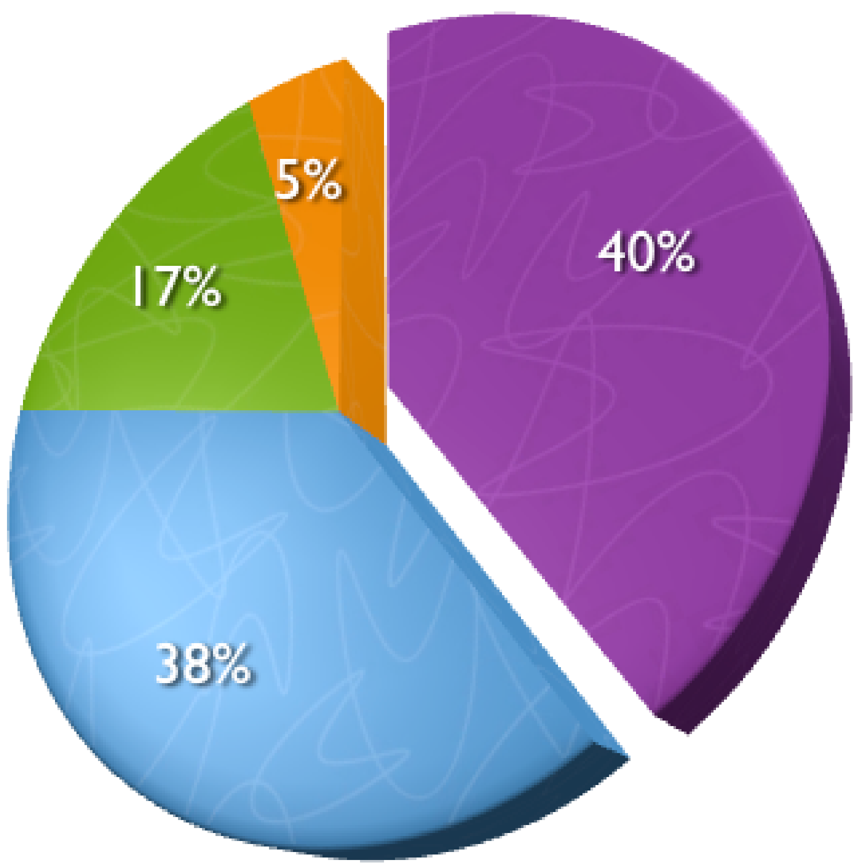


Source: CSI/FBI Survey 2007

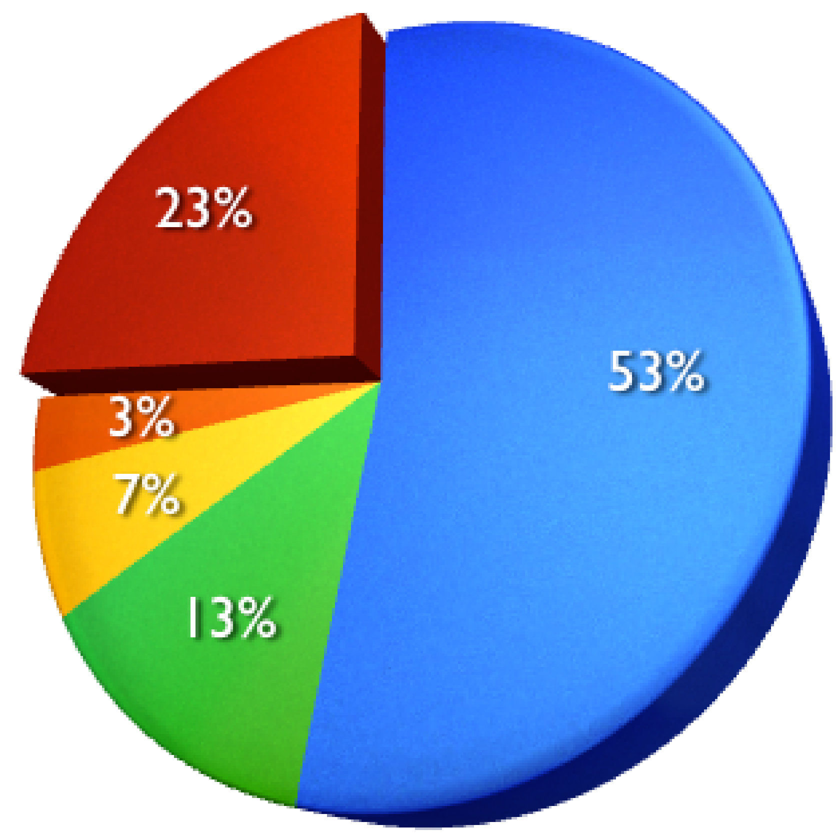
Why: Security matters

● Vulnerabilities ● Fraud ● Dumb Employees ● DoS

Cost of cybercrime ($80bn)



Type of vulnerability



● Buffer Overflow ● Injection ● Int Overflow ● Format String ● Other

Source: CSI/FBI Survey 2007

Source: US CERT, Feb 2008

Why: Verification matters

<exploding-rocket-video />

PowerPoint

A fatal exception 0E has occurred at 0137:BFFA21C9. The current application will be terminated.

* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applicatons.

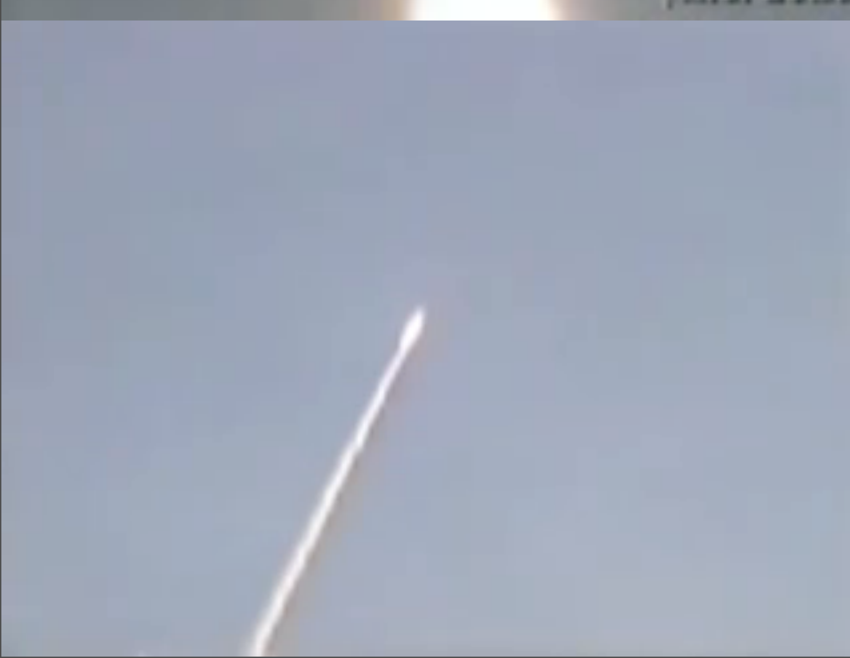
Press any key to continue _

<exploding-rocket-video />


ARIANESPACE



CNN
Interactive



Agenda

- Past: Control-flow analysis
- Present: Environment analysis
- Future: Logic-flow analysis

What is *higher-order*?

Higher-order languages

Higher-order = Computation as value

Higher-order languages

Higher-order = Computation as value

- Scheme
- Lisp
- ML
- Haskell

Higher-order languages

Higher-order = Computation as value

- Scheme
- Lisp
- ML
- Haskell
- Java
- C#
- C++
- Javascript

Objects and closures = Same challenge

Higher-order languages

Higher-order $=$ Computation as value

- Scheme
- Lisp
- ML
- Haskell

- Java
- C#
- C++
- Javascript

Objects and closures $=$ Same challenge

λ-calculus (Church, 1928)

λ-calculus (Church, 1928)

- Minimalist, universal language

Alonzo Church

λ-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:

v [variable]

Alonzo Church

λ-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:

v	[variable]
$e_1(e_2)$	[function application]

Alonzo Church

λ-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:

v	[variable]
$e_1(e_2)$	[function application]
$\lambda v.e$	[anonymous function]

Alonzo Church

λ-calculus domains

λ-calculus domains

- Closure = λ-Term × Environment

λ-calculus domains

- Closure = λ-Term $\times$ Environment
- Environment = Variable $\rightarrow$ Closure

λ-calculus semantics

λ-calculus semantics

$$\text{eval} : \text{Exp} \times \text{Environment} \to \text{Closure}$$

λ-calculus semantics

$$\text{eval} : \text{Exp} \times \text{Environment} \to \text{Closure}$$

$$\text{eval}([\![v]\!], env) = env(v)$$

λ-calculus semantics

$$\text{eval} : \text{Exp} \times \text{Environment} \to \text{Closure}$$

$$\text{eval}([\![v]\!], env) = env(v)$$

$$\text{eval}([\![\lambda v.e]\!],\ env) = ([\![\lambda v.e]\!],\ env)$$

λ-calculus semantics

$$\text{eval} : \text{Exp} \times \text{Environment} \rightarrow \text{Closure}$$

$$\text{eval}([\![v]\!], env) = env(v)$$

$$\text{eval}([\![\lambda v. e]\!],\ env) = ([\![\lambda v. e]\!],\ env)$$

$$\begin{aligned}\text{eval}([\![e_1(e_2)]\!], env) &= \text{eval}(e_b,\ env''), \text{ where} \\ ([\![\lambda v. e_b]\!],\ env') &= \text{eval}(e_1, env) \\ env'' &= env'[v \mapsto \text{eval}(e_2, env)]\end{aligned}$$

Higher-order analysis:
A brief history

Early 1980s

- Mostly Lisp, Scheme
- Poor performance relative to C, Fortran
- T: Optimizing Scheme compiler (1982)

Early 1980s

- Mostly Lisp, Scheme
- Poor performance relative to C, Fortran
- T: Optimizing Scheme compiler (1982)

"...every day, I went in to WRL, failed for 8 hours, then went home."

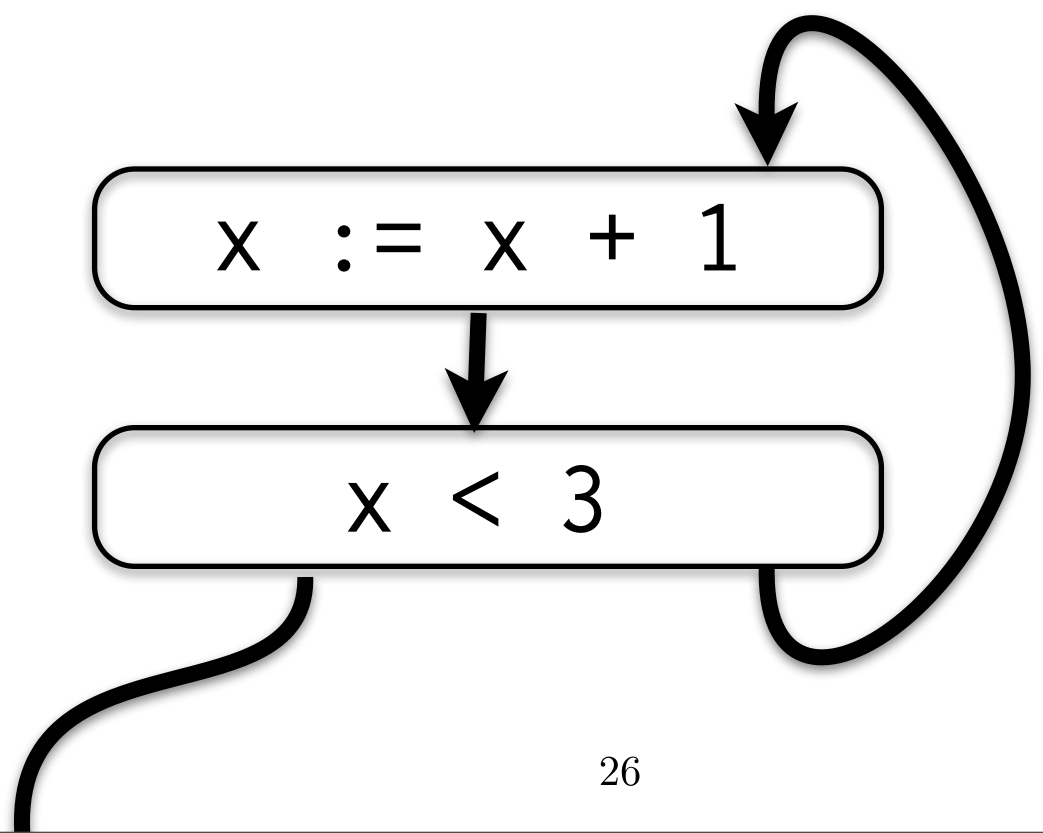
Olin Shivers, *History of T*

The control-flow problem

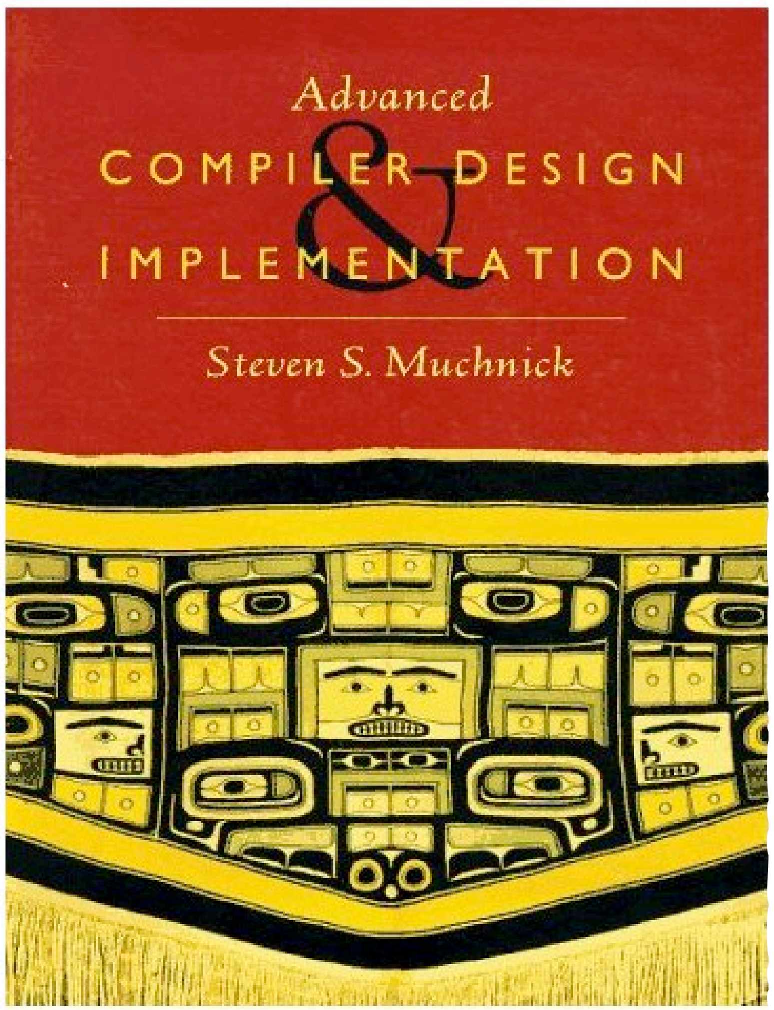
```
g:
 x := x + 1
 if x < 3
  goto g
 else
  return x
```

The control-flow problem

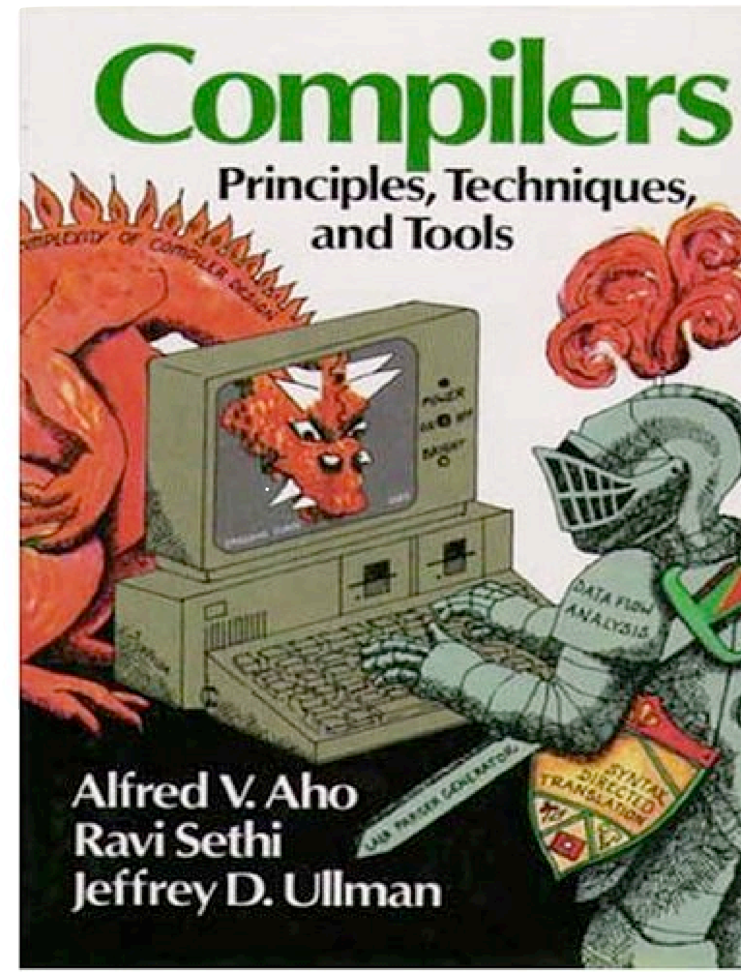
```
g:
 x := x + 1
 if x < 3
  goto g
 else
  return x
```

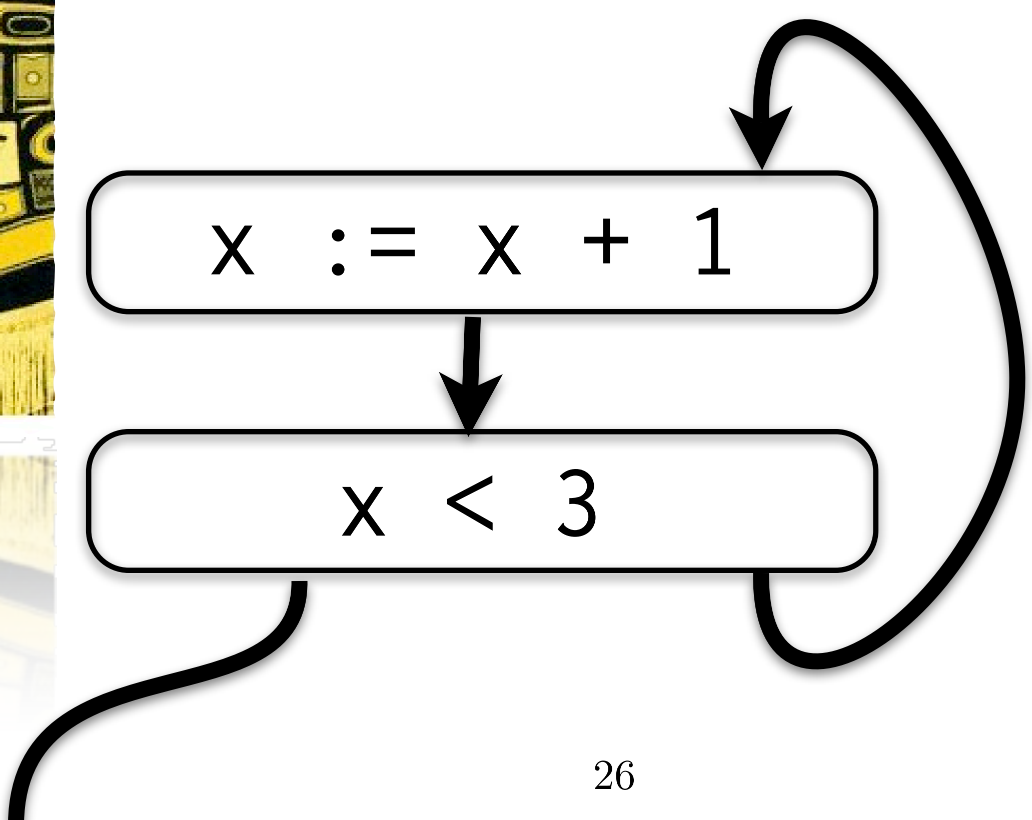


The control-flow problem

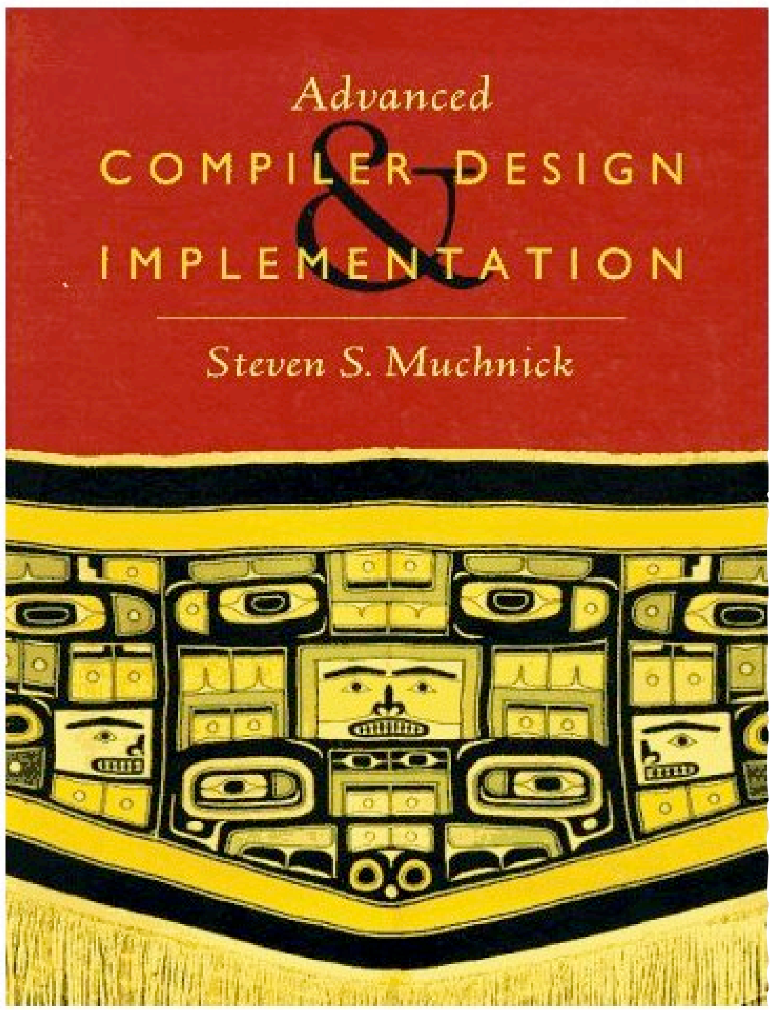



```
g:
 x := x + 1
 if x < 3
  goto g
 else
  return x
```

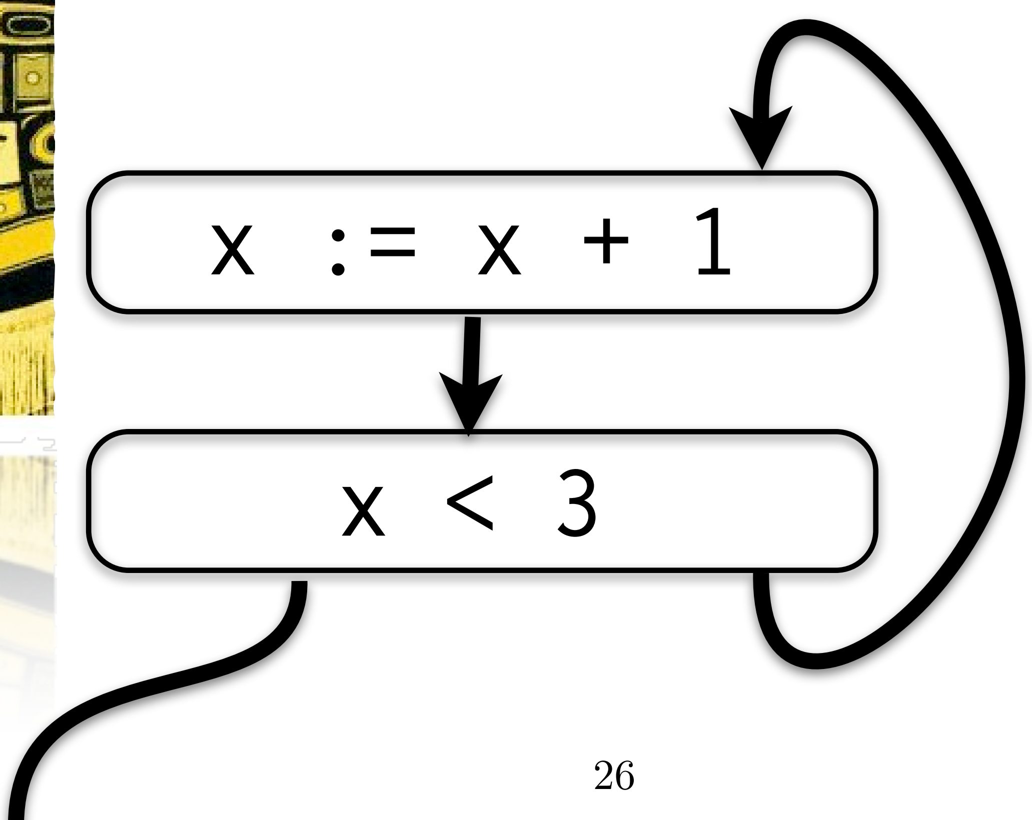


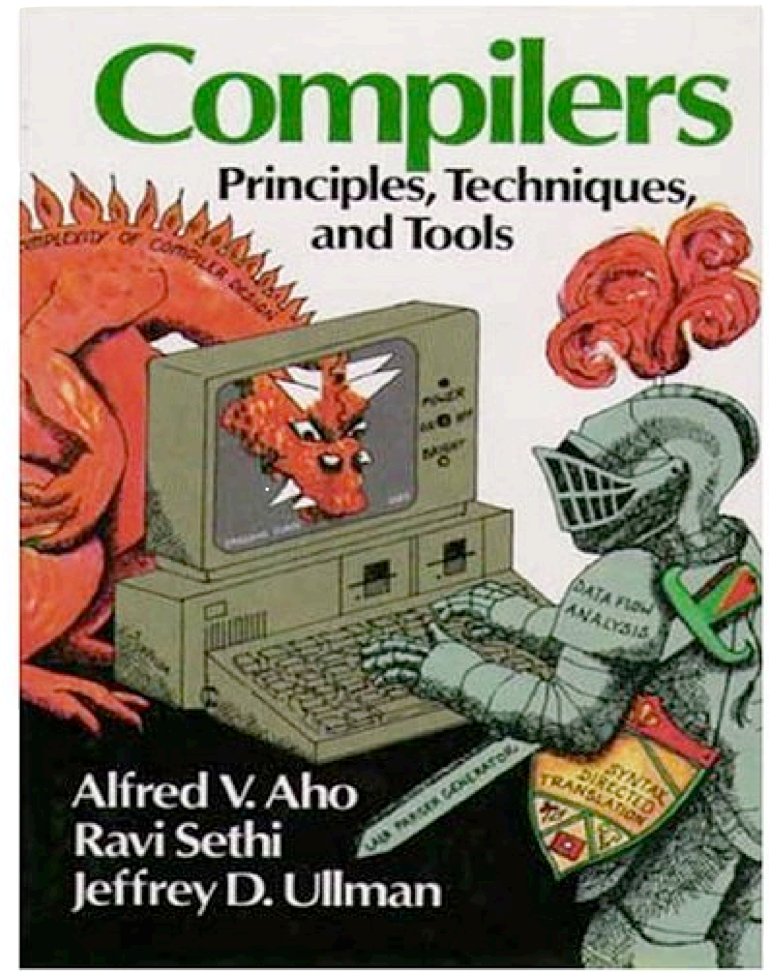


The control-flow problem

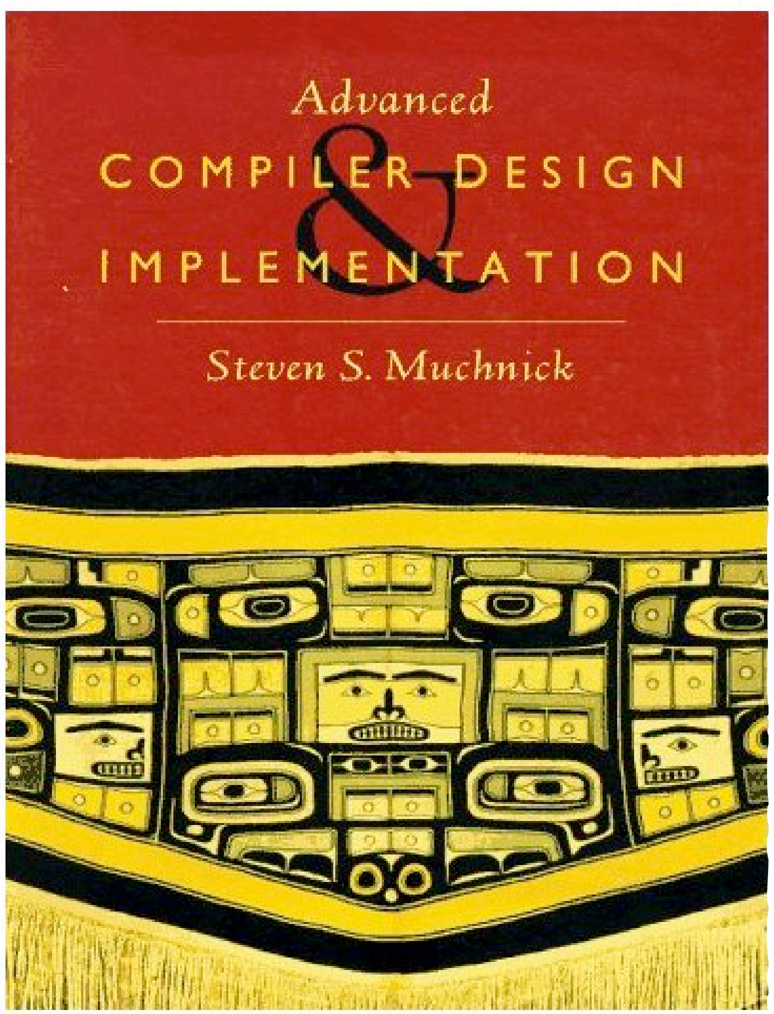



```
g(x) =
 let x = x + 1 in
 if x < 3 then
  g(x)
 else
  x
```

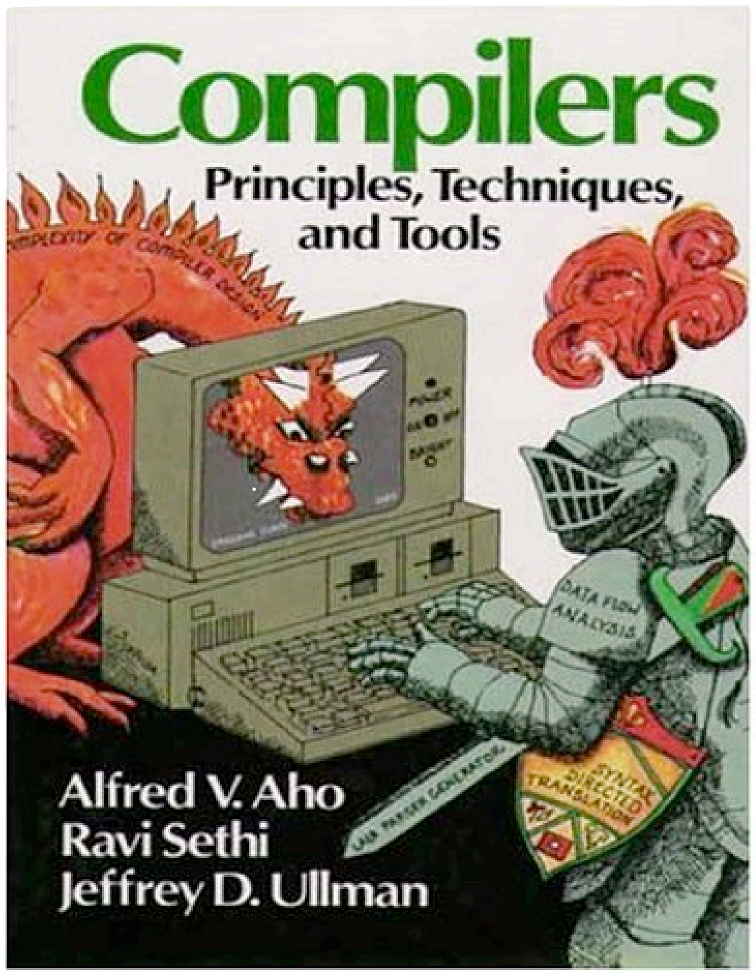


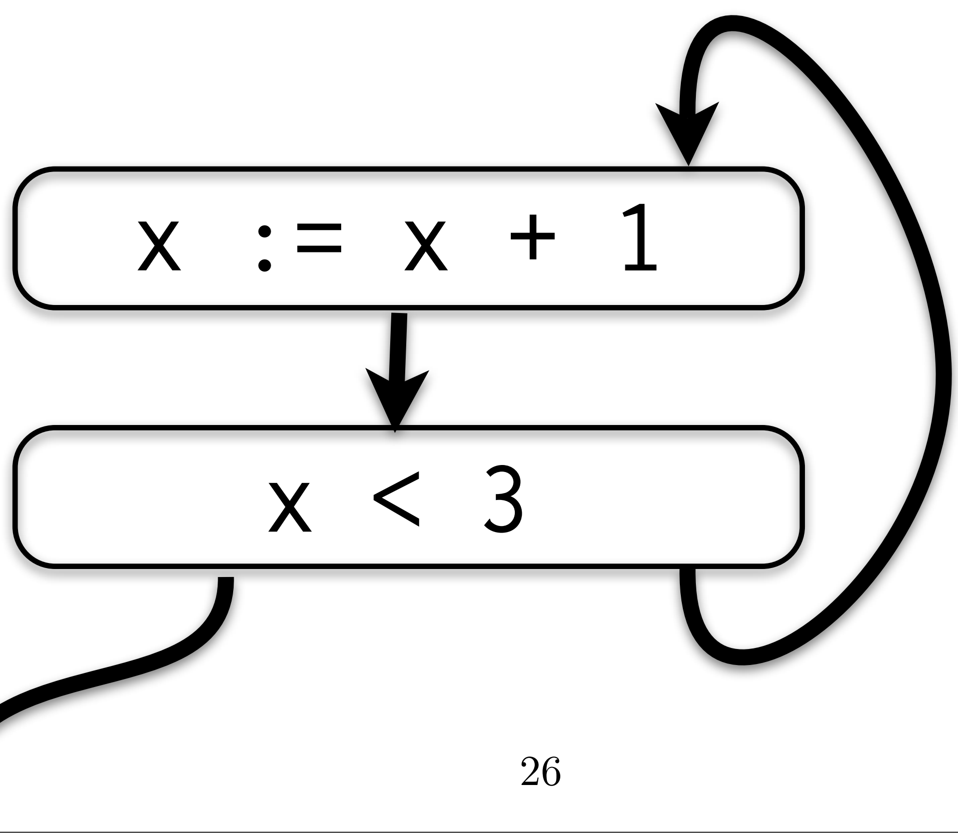


The control-flow problem

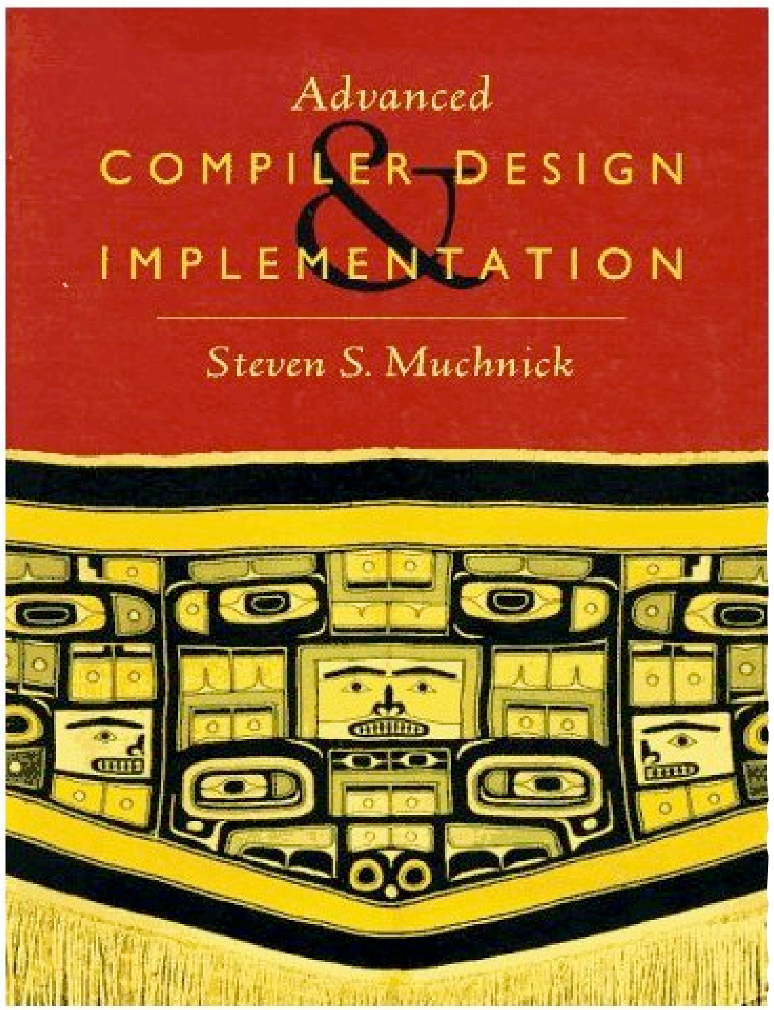



```
g(x) =
 let x = x + 1 in
 if x < 3 then
  h(x)
 else
  x
```

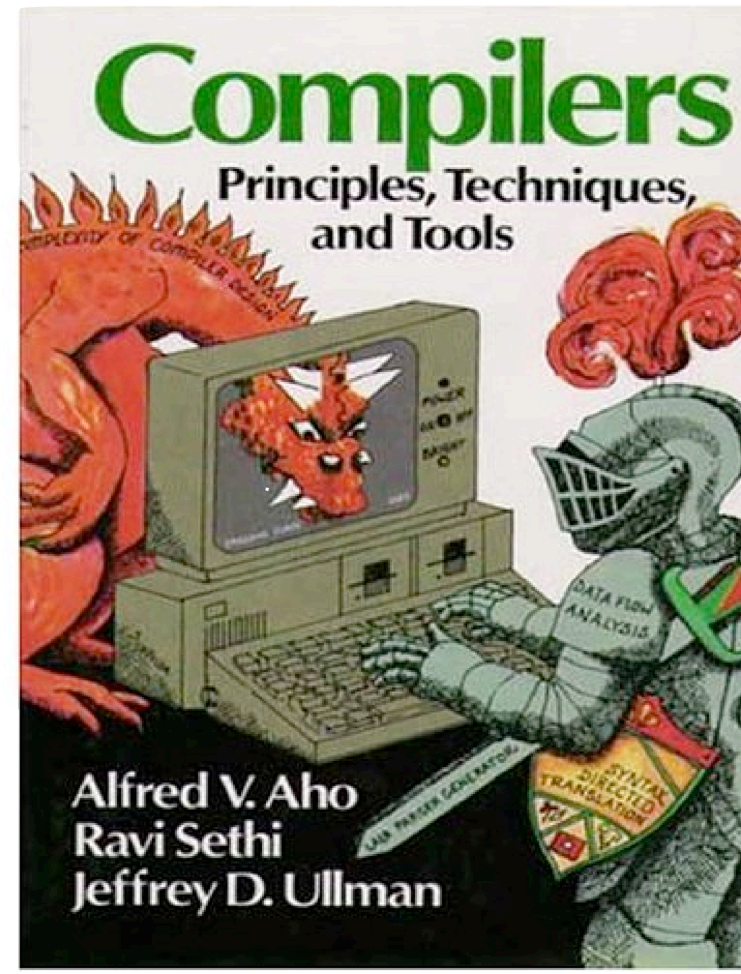




The control-flow problem




```
g(x) =
 let x = x + 1 in
 if x < 3 then
  h(x)
 else
  x
```



The control-flow problem

```
g(x) =
 let x = x + 1 in
 if x < 3 then
  h(x)
 else
  x
```

The control-flow problem

```
apply f x = f(x)
```

The control-flow problem

```
apply f x = f(x)
```

What procedures are called here?

The control-flow problem

```
apply f x = f(x)
```

```
h x = x + 1

apply h 4
```

$$\text{FlowsTo}[\mathsf{f}] = \{\mathsf{h}\}$$

The control-flow problem

```
apply f x = f(x)

h x = x + 1                g x = apply h 4

                           apply g 3
```

$$\text{FlowsTo}[\mathsf{f}] = \{\mathsf{h},\mathsf{g}\}$$

The control-flow problem

```
apply f x = f(x)

(apply apply apply) apply
```

The control-flow problem

```
animal.eat();
```

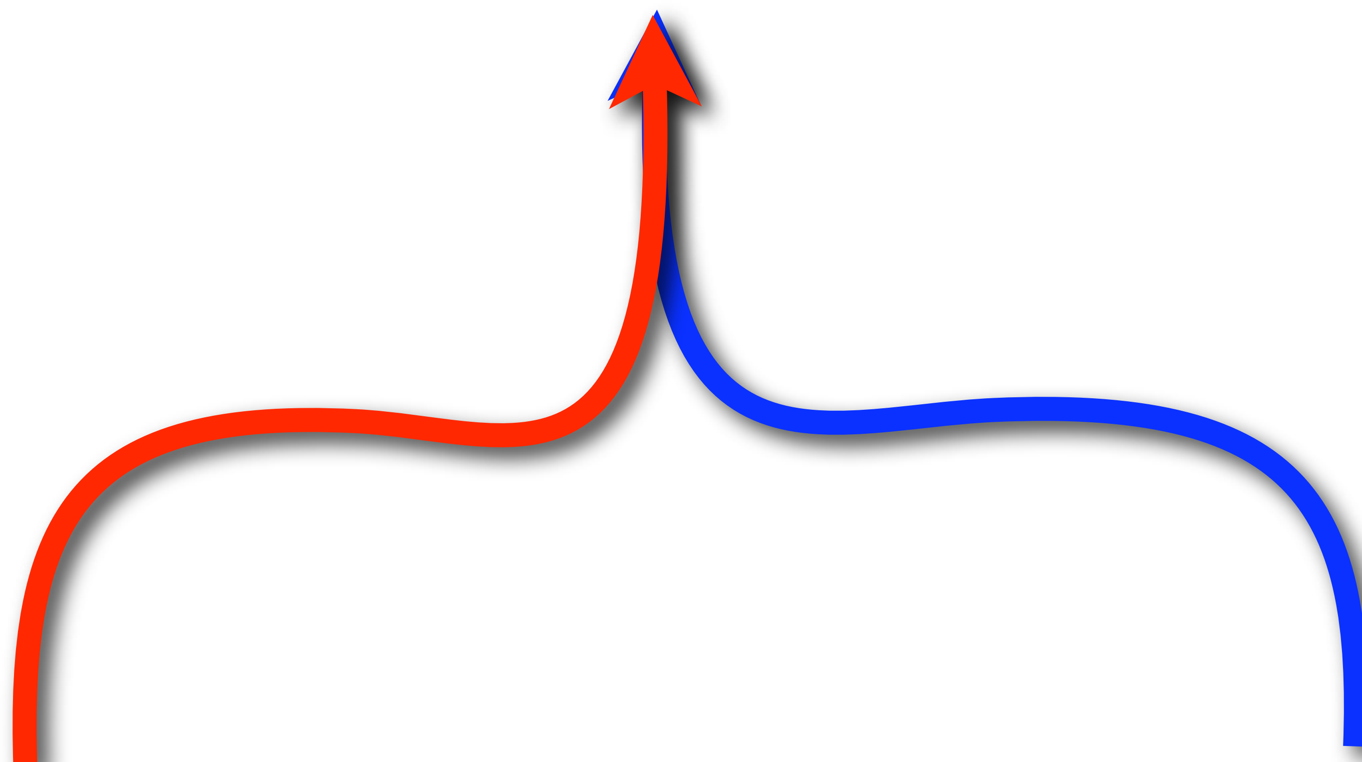
The control-flow problem

```
animal.eat();
```

Which classes flow here?

The control-flow problem

```
animal.eat();
```



```
class Dog : Animal {
 void eat() {...}
}
```

```
class Cat : Animal {
 void eat() {...}
}
```

The control-flow problem

Control-flow depends on data-flow,
but data-flow depends on control-flow.

Higher-order control-flow analysis (CFA)

Observations on flow

$$e_1(e_2)$$

Observations on flow

$\lambda v.e_{\mathrm{b}}$

$e_1(e_2)$

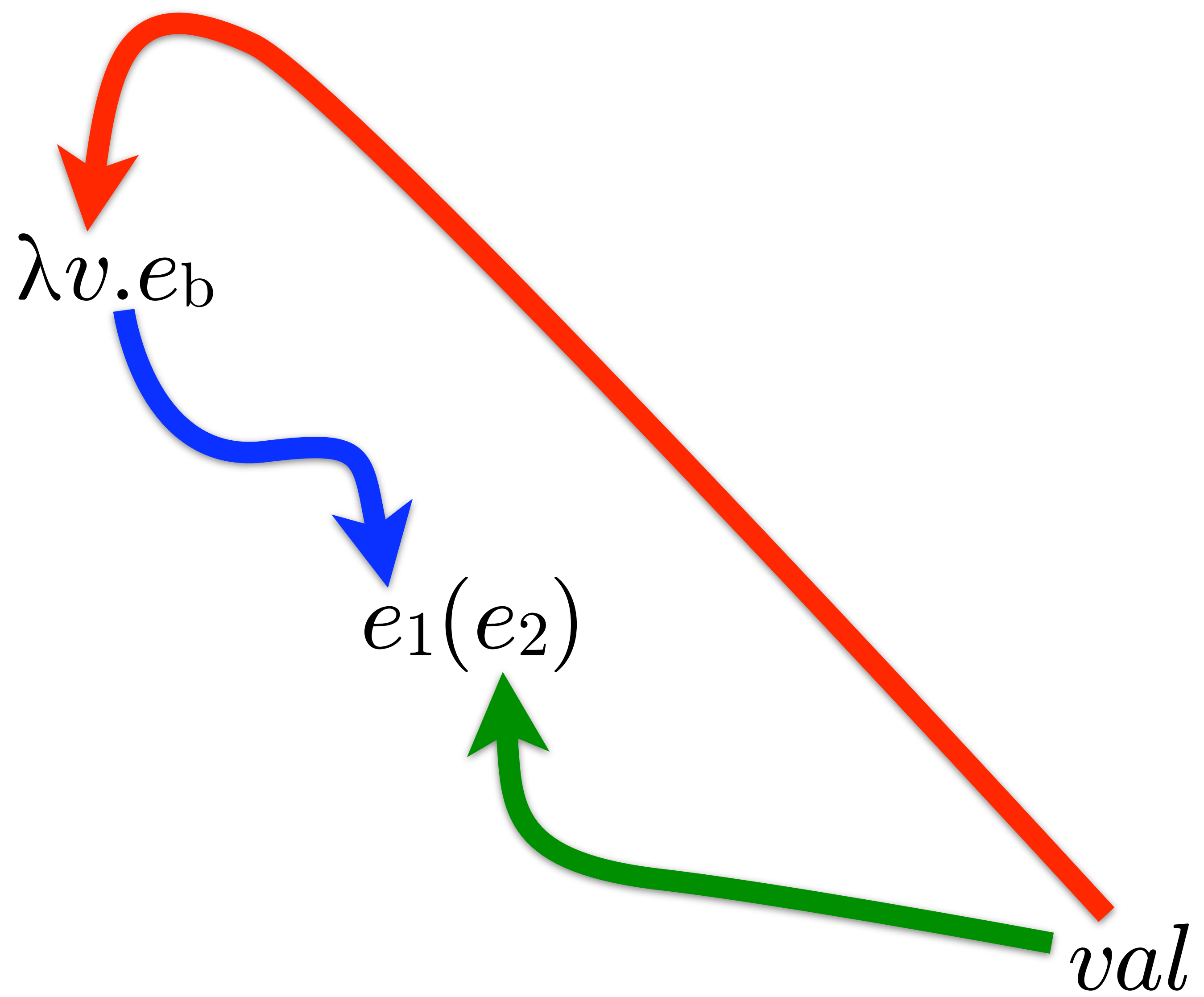
Observations on flow

$\lambda v.e_{\mathrm{b}}$

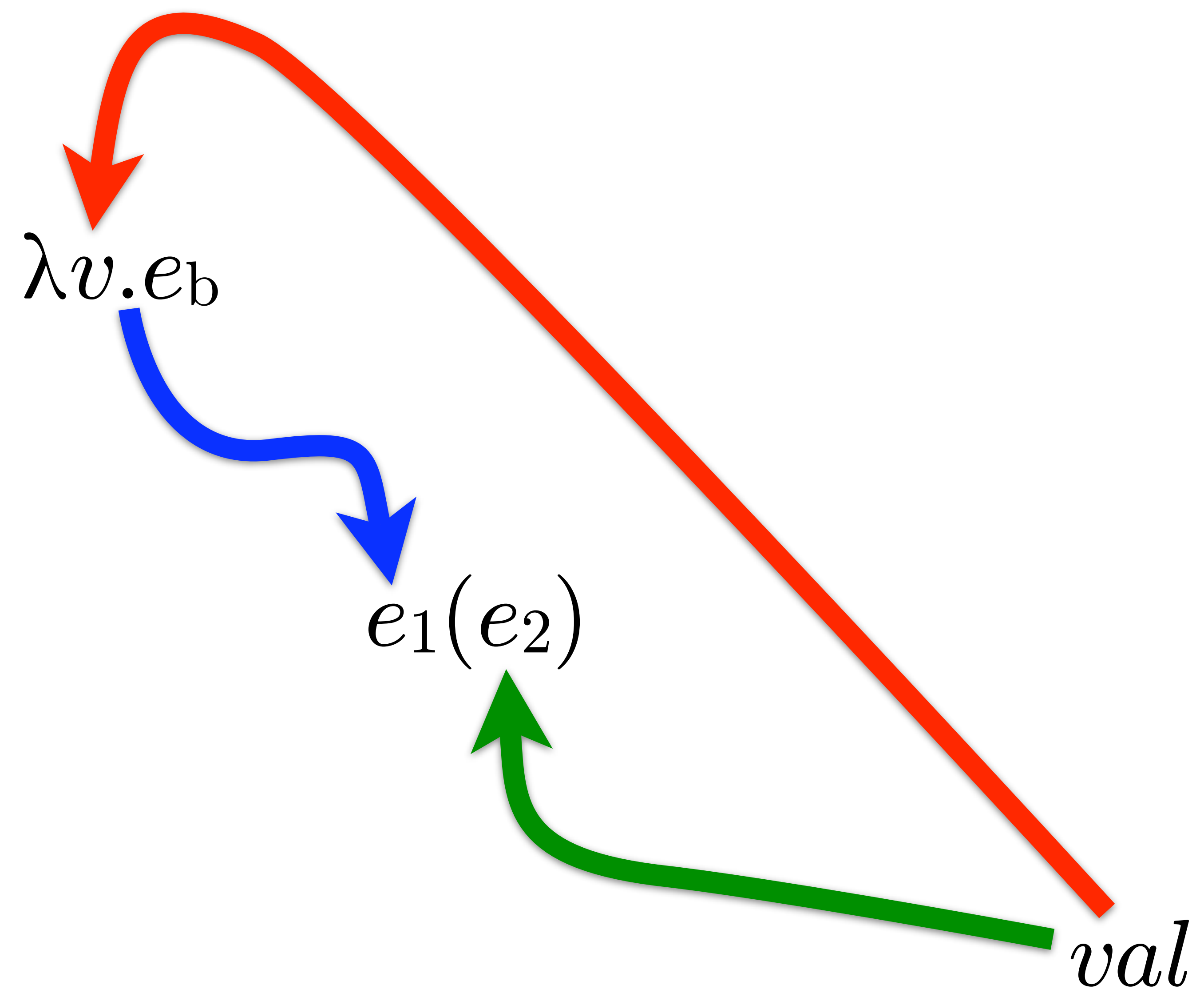
$e_1(e_2)$

val

Observations on flow



Observations on flow



$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1] \quad \text{and} \quad val \in \mathrm{FlowsTo}[e_2]}{val \in \mathrm{FlowsTo}[v]}$$

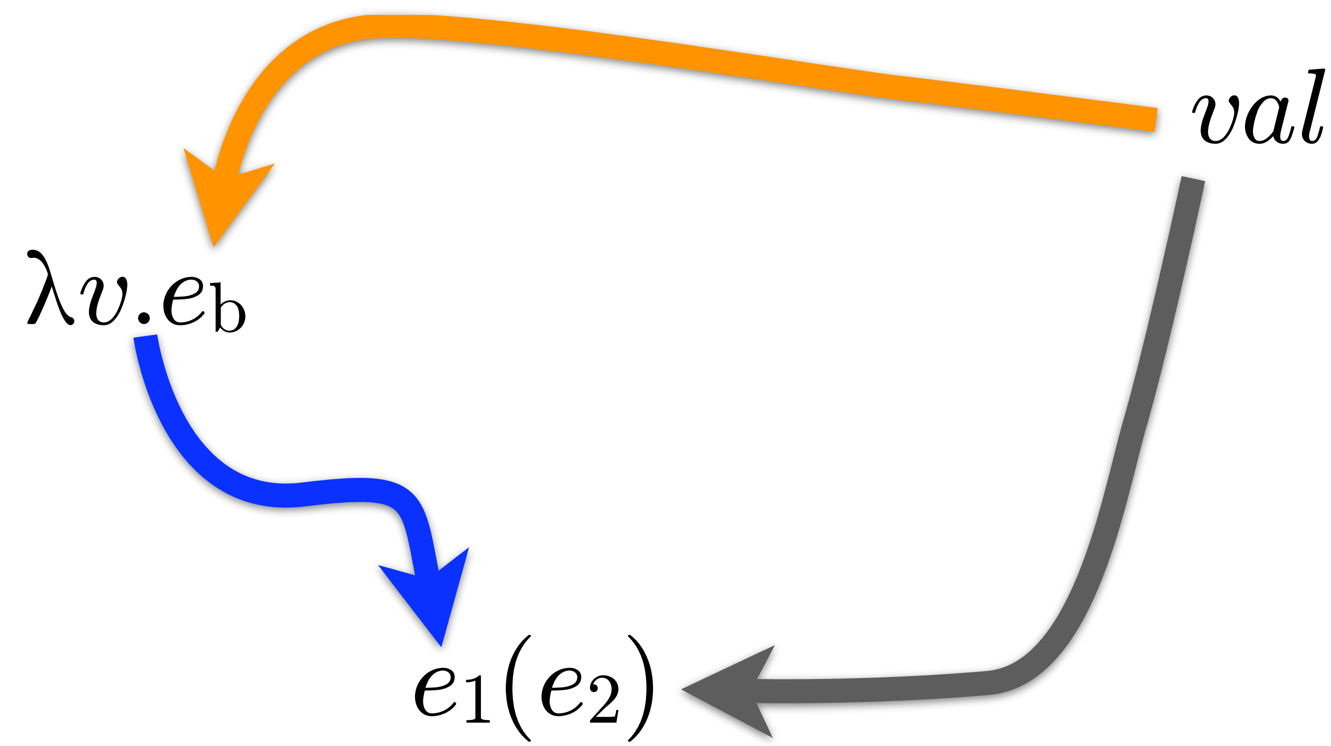
Observations on flow

val

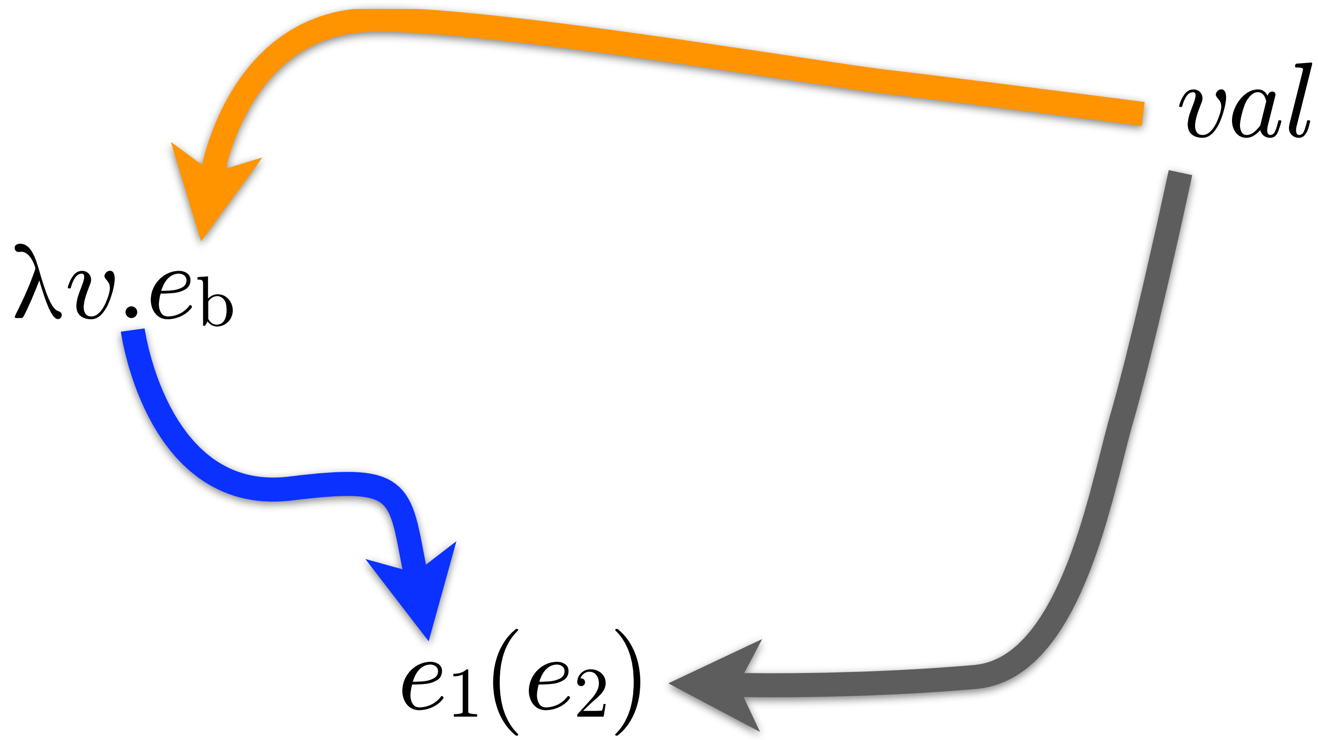
$\lambda v.e_{\mathrm{b}}$

$e_1(e_2)$

Observations on flow



Observations on flow



$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1] \quad \text{and} \quad val \in \mathrm{FlowsTo}[e_{\mathrm{b}}]}{val \in \mathrm{FlowsTo}[e_1(e_2)]}$$

0CFA (Shivers, 1988)

$$\frac{\lambda v.e_{\mathrm{b}} \in \text{FlowsTo}[e_1] \quad \text{and} \quad val \in \text{FlowsTo}[e_{\mathrm{b}}]}{val \in \text{FlowsTo}[e_1(e_2)]}$$

0CFA (Shivers, 1988)

$$\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[\lambda v.e_{\mathrm{b}}]$$

$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1] \quad \text{and} \quad \mathit{val} \in \mathrm{FlowsTo}[e_{\mathrm{b}}]}{\mathit{val} \in \mathrm{FlowsTo}[e_1(e_2)]}$$

$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1] \quad \text{and} \quad \mathit{val} \in \mathrm{FlowsTo}[e_2]}{\mathit{val} \in \mathrm{FlowsTo}[v]}$$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

Example: 0CFA

$\text{FlowsTo}[\texttt{id}] = \{\lambda\texttt{x.x}\}$

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

Example: 0CFA

$\text{FlowsTo}[\texttt{id}] = \{\lambda\texttt{x.x}\}$

$\text{FlowsTo}[\texttt{x}] = \{\texttt{3}\}$

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\text{FlowsTo}[\texttt{id}] = \{\lambda\texttt{x.x}\}$

$\text{FlowsTo}[\texttt{x}] = \{\texttt{3}\}$

$\text{FlowsTo}[\texttt{id(3)}] = \{\texttt{3}\}$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$$\text{FlowsTo}[\texttt{id}] = \{\texttt{λx.x}\}$$

$$\text{FlowsTo}[\texttt{x}] = \{\texttt{3}\}$$

$$\text{FlowsTo}[\texttt{id(3)}] = \{\texttt{3}\}$$

$$\text{FlowsTo}[\texttt{v1}] = \{\texttt{3}\}$$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$$\text{FlowsTo}[\texttt{id}] = \{\texttt{λx.x}\}$$

$$\text{FlowsTo}[\texttt{x}] = \{\texttt{3,4}\}$$

$$\text{FlowsTo}[\texttt{id(3)}] = \{\texttt{3}\}$$

$$\text{FlowsTo}[\texttt{v1}] = \{\texttt{3}\}$$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\mathrm{FlowsTo}[\texttt{id}] = \{\texttt{λx.x}\}$

$\mathrm{FlowsTo}[\texttt{x}] = \{\texttt{3,4}\}$

$\mathrm{FlowsTo}[\texttt{id(3)}] = \{\texttt{3}\}$

$\mathrm{FlowsTo}[\texttt{v1}] = \{\texttt{3}\}$

$\mathrm{FlowsTo}[\texttt{id(4)}] = \{\texttt{3,4}\}$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\text{FlowsTo}[\texttt{id}] = \{\lambda\texttt{x.x}\}$

$\text{FlowsTo}[\texttt{x}] = \{\texttt{3,4}\}$

$\text{FlowsTo}[\texttt{id(3)}] = \{\texttt{3}\}$

$\text{FlowsTo}[\texttt{v1}] = \{\texttt{3}\}$

$\text{FlowsTo}[\texttt{id(4)}] = \{\texttt{3,4}\}$

$\text{FlowsTo}[\texttt{v2}] = \{\texttt{3,4}\}$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\text{FlowsTo}[\texttt{id}] = \{\texttt{λx.x}\}$

$\text{FlowsTo}[\texttt{x}] = \{\texttt{3,4}\}$

$\text{FlowsTo}[\texttt{id(3)}] = \{\texttt{3,4}\}$

$\text{FlowsTo}[\texttt{v1}] = \{\texttt{3}\}$

$\text{FlowsTo}[\texttt{id(4)}] = \{\texttt{3,4}\}$

$\text{FlowsTo}[\texttt{v2}] = \{\texttt{3,4}\}$

Example: 0CFA

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\mathrm{FlowsTo}[\mathtt{id}] = \{\mathtt{\lambda x.x}\}$

$\mathrm{FlowsTo}[\mathtt{x}] = \{\mathtt{3,4}\}$

$\mathrm{FlowsTo}[\mathtt{id(3)}] = \{\mathtt{3,4}\}$

$\mathrm{FlowsTo}[\mathtt{v1}] = \{\mathtt{3,4}\}$

$\mathrm{FlowsTo}[\mathtt{id(4)}] = \{\mathtt{3,4}\}$

$\mathrm{FlowsTo}[\mathtt{v2}] = \{\mathtt{3,4}\}$

0CFA recap

- It worked!
- It's not fast: $O(n^3)$
- It's imprecise.

The Palsberg paradigm

"Higher-order analysis =
first-order analysis + control-flow analysis."

Jens Palsberg

 Source: "Closure analysis in constraint form." 1995.

0CFA (Shivers, 1988)

For each application $e_1\ e_2$:

$$(\lambda x.e_\mathrm{b}) \in \mathrm{FlowsTo}[e_1]$$
$$\Rightarrow$$
$$\mathrm{FlowsTo}[e_\mathrm{b}] \subseteq \mathrm{FlowsTo}[(e_1\ e_2)]$$

$$(\lambda x.e_\mathrm{b}) \in \mathrm{FlowsTo}[e_1]$$
$$\Rightarrow$$
$$\mathrm{FlowsTo}[e_2] \subseteq \mathrm{FlowsTo}[x]$$

VFA (Heinglen, 1992)

For each application $e_1\ e_2$:

$$(\lambda x.e_{\mathrm{b}}) \in \mathrm{FlowsTo}[e_1]$$
$$\Rightarrow$$
$$\mathrm{FlowsTo}[e_{\mathrm{b}}] = \mathrm{FlowsTo}[(e_1\ e_2)]$$

$$(\lambda x.e_{\mathrm{b}}) \in \mathrm{FlowsTo}[e_1]$$
$$\Rightarrow$$
$$\mathrm{FlowsTo}[e_2] = \mathrm{FlowsTo}[x]$$

VFA

- Fast: Almost linear
- Awful precision: FlowsTo[3] $=$ {3,4}

k-CFA (Shivers, 1991)

k-CFA (Shivers, 1991)

- Splits variable flow sets by calling context

k-CFA (Shivers, 1991)

- Splits variable flow sets by calling context

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\text{FlowsTo}[\texttt{x}, \texttt{id(3)}] = \{\texttt{3}\}$

$\text{FlowsTo}[\texttt{x}, \texttt{id(4)}] = \{\texttt{4}\}$

k-CFA (Shivers, 1991)

- Splits variable flow sets by calling context
- EXPTIME-Complete (Van Horn & Mairson, 2008)

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

$\text{FlowsTo}[\texttt{x}, \texttt{id(3)}] = \{\texttt{3}\}$

$\text{FlowsTo}[\texttt{x}, \texttt{id(4)}] = \{\texttt{4}\}$

poly-CFA (Wright, 1995)

poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism

poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct

poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct

```
let id = λx.x
    v1 = id(3)
    v2 = id(4)
 in v2
```

poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct

```
let id = λx.x
    v1 = (λx₁.x₁)(3)
    v2 = (λx₂.x₂)(4)
 in v2
```

FlowsTo[v1] $= \{$3$\}$

FlowsTo[v2] $= \{$4$\}$

poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct
- Speed, precision not necessarily tradeoff

```
let id = λx.x
    v1 = (λx₁.x₁)(3)
    v2 = (λx₂.x₂)(4)
 in v2
```

$\text{FlowsTo}[\mathtt{v1}] = \{\mathtt{3}\}$

$\text{FlowsTo}[\mathtt{v2}] = \{\mathtt{4}\}$

Other work

- Reformulations
  - As abstract interpretations
  - As type systems
- Demand-driven
- Flow sensitivity
- FlowsTo cloning (by point, context)
- Complexity bounds

Aftermath

- Optimization: Common sub-exp. elimination
- Optimization: Copy propagation
- Optimization: Context-sensitive inlining
- Optimization: Induction variable elimination
- Optimization: Invariant hoisting
- Optimization: Global register allocation
- ...
- Stalin, MLton: Parity with C

Control-flow analysis is not enough.

Half the problem

λ-terms don't flow anywhere.

Half the problem

λ-terms don't flow anywhere.

Closures do.

Half the problem

λ-terms don't flow anywhere.

Closures do.

$$\text{Closure} = \lambda\text{-Term} \times \text{Environment}$$

Half the problem

λ-terms don't flow anywhere.

Closures do.

Closure = λ-Term

Half the problem

Classes don't flow anywhere.

Objects do.

$$\text{Object} = \text{Class} \times \text{Record}$$

Why environment matters

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
f x h = if x = 0
           then h()
           else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
f x h = if x = 0
          then h()
          else λ().x                 (λ().x,x→3)

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
                                     (λ().x,x→3)
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

(λ().x,x→3)

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
                                                  (λ().x,x→3)
f x h = if x = 0
          then h()
          else λ().x

f 0 (f 3 nil)
```

Safe to inline?

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

(λ().x,x→3)

Safe to inline?

Inlining

```
f x h = if x = 0
        then h()
        else λ().x

f 0 (f 3 nil)
```

```
(λ().x,x→3)
```

Safe to inline?

Inlining

```
(λ().x,x→3)

f x h = if x = 0
        then (λ().x)()
        else λ().x

f 0 (f 3 nil)
```

$$[\mathrm{x} \rightarrow 3] \quad v. \quad [\mathrm{x} \rightarrow 0]$$

Safe to inline? No.

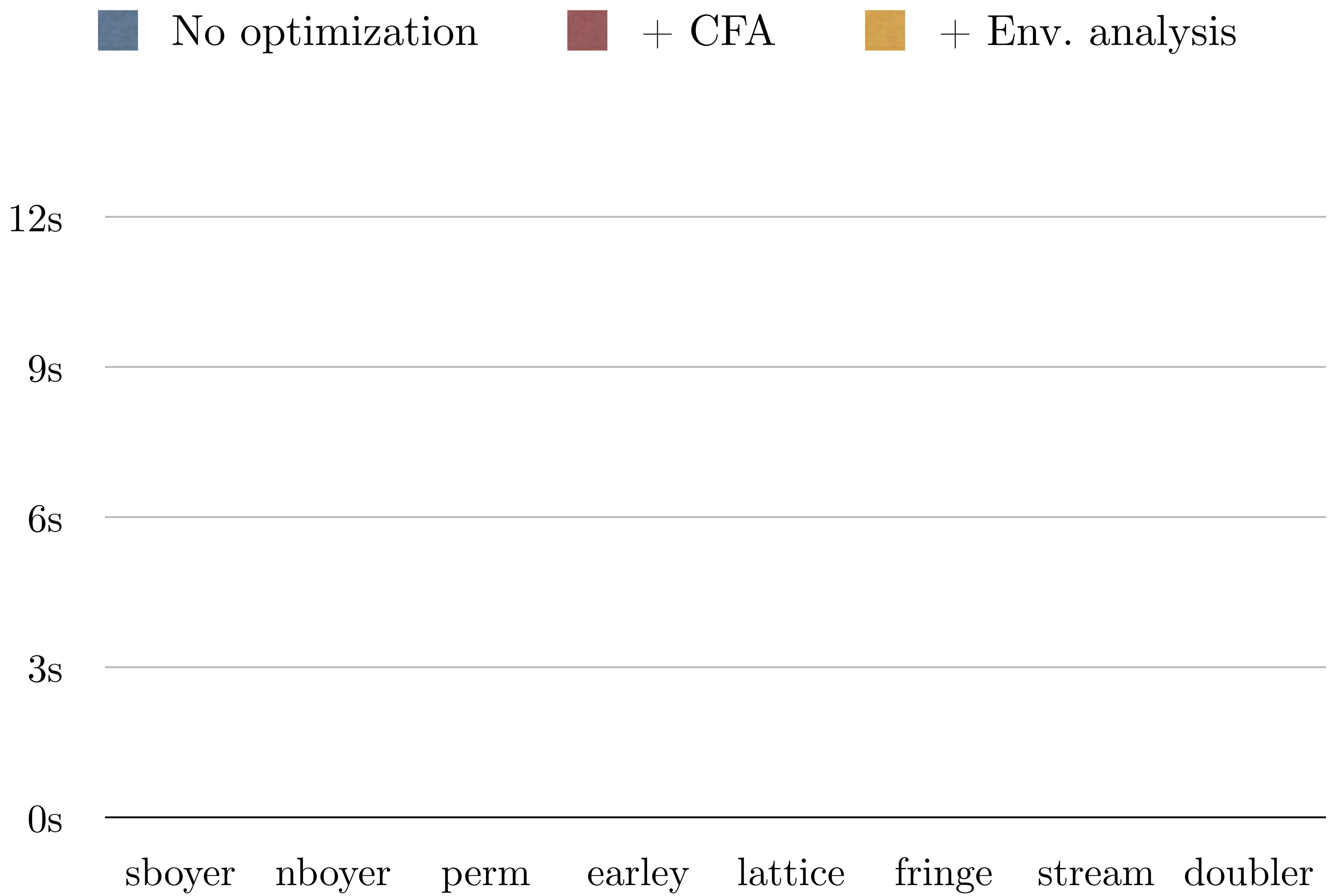
And also...

- Super-beta copy propagation
- Globalization (Sestoft)
- Rematerialization
- Teleportation/environment lifting
- Register-allocated environments
- Static environment allocation
- Stack-aware inlining
- Lightweight closure conversion (Wand & Steckler)
- Generalized escape analysis
- Lightweight continuation promotion
- Coroutine fusion
- ...

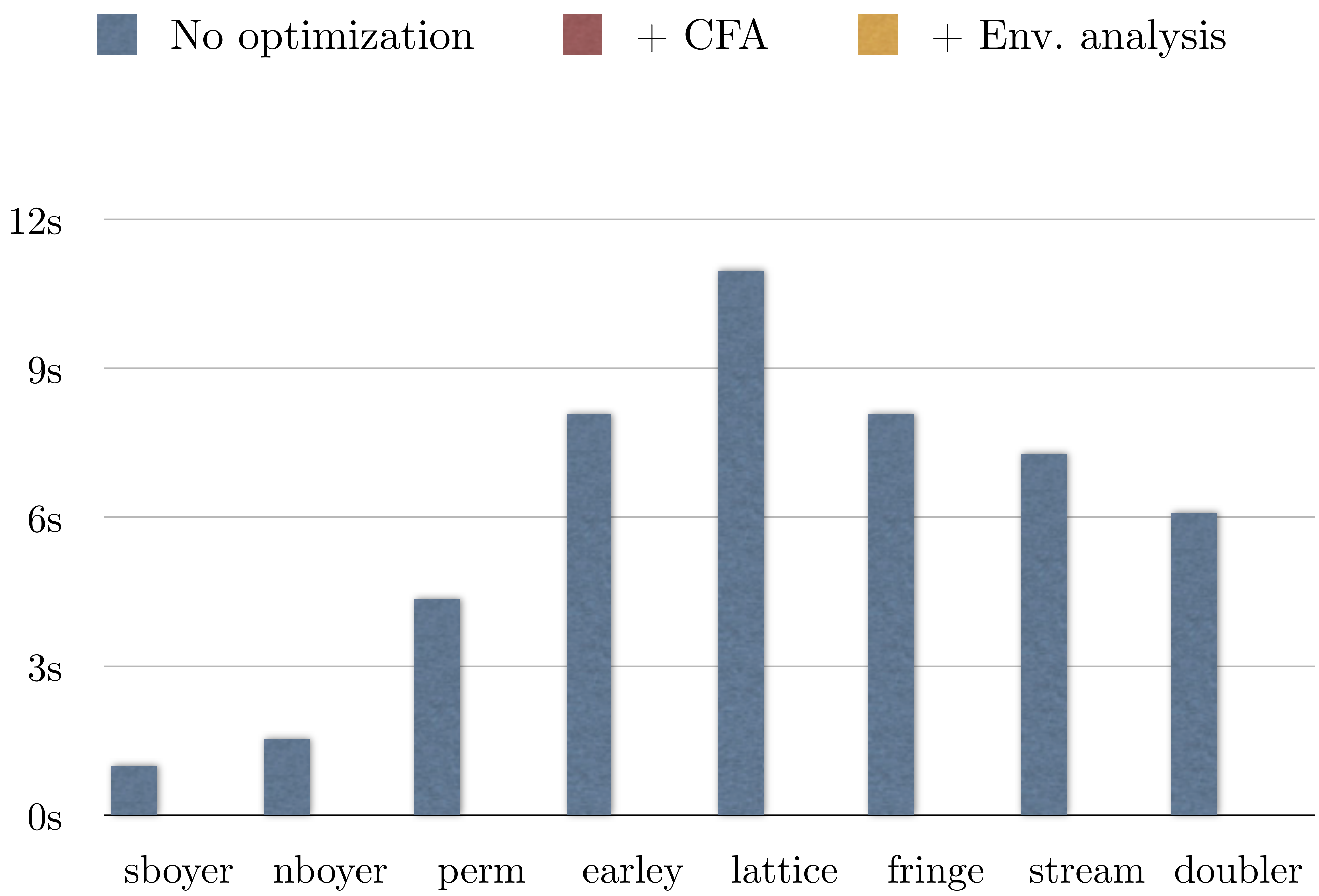
Program run times

No optimization $\quad$ + CFA $\quad$ + Env. analysis

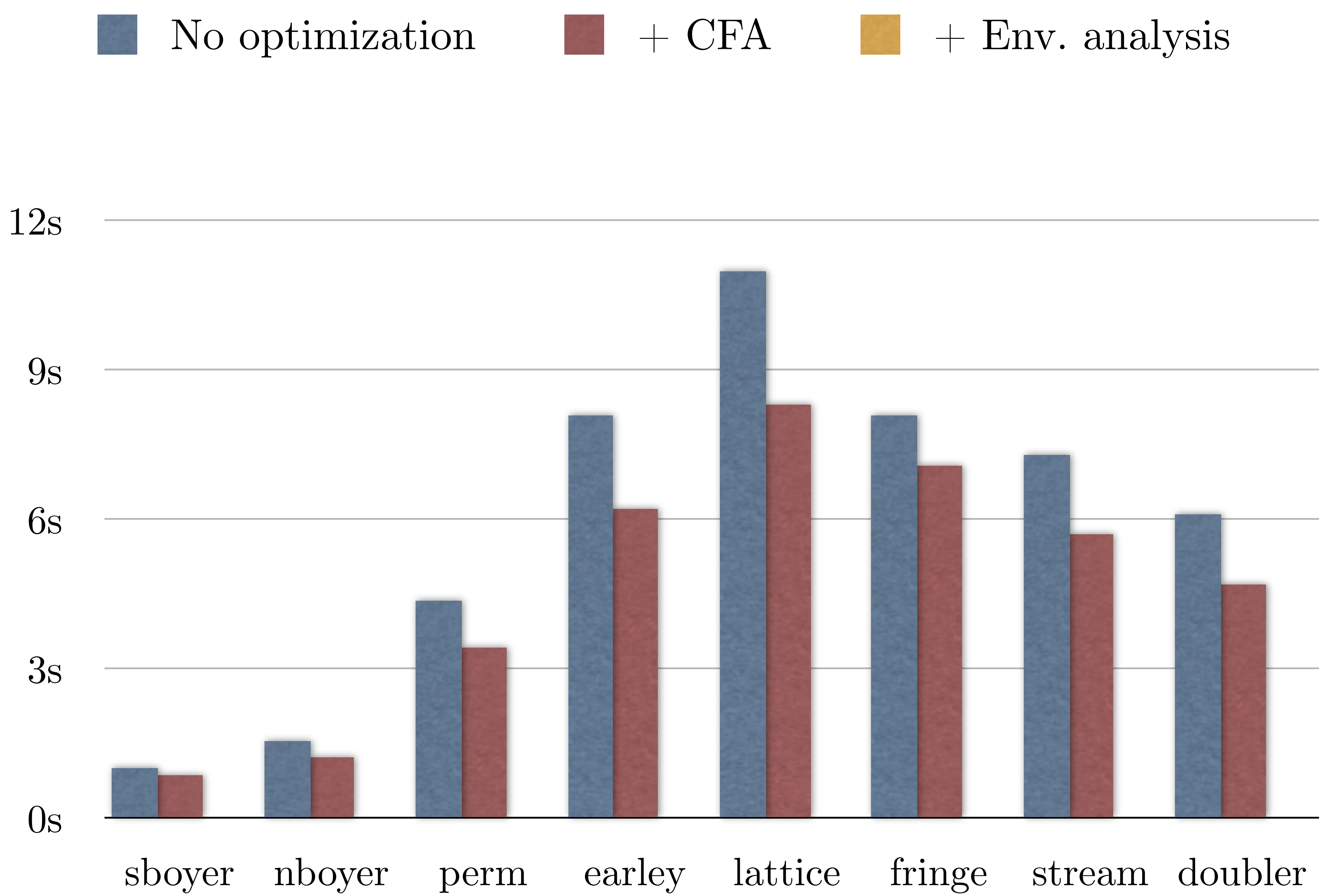
Program run times



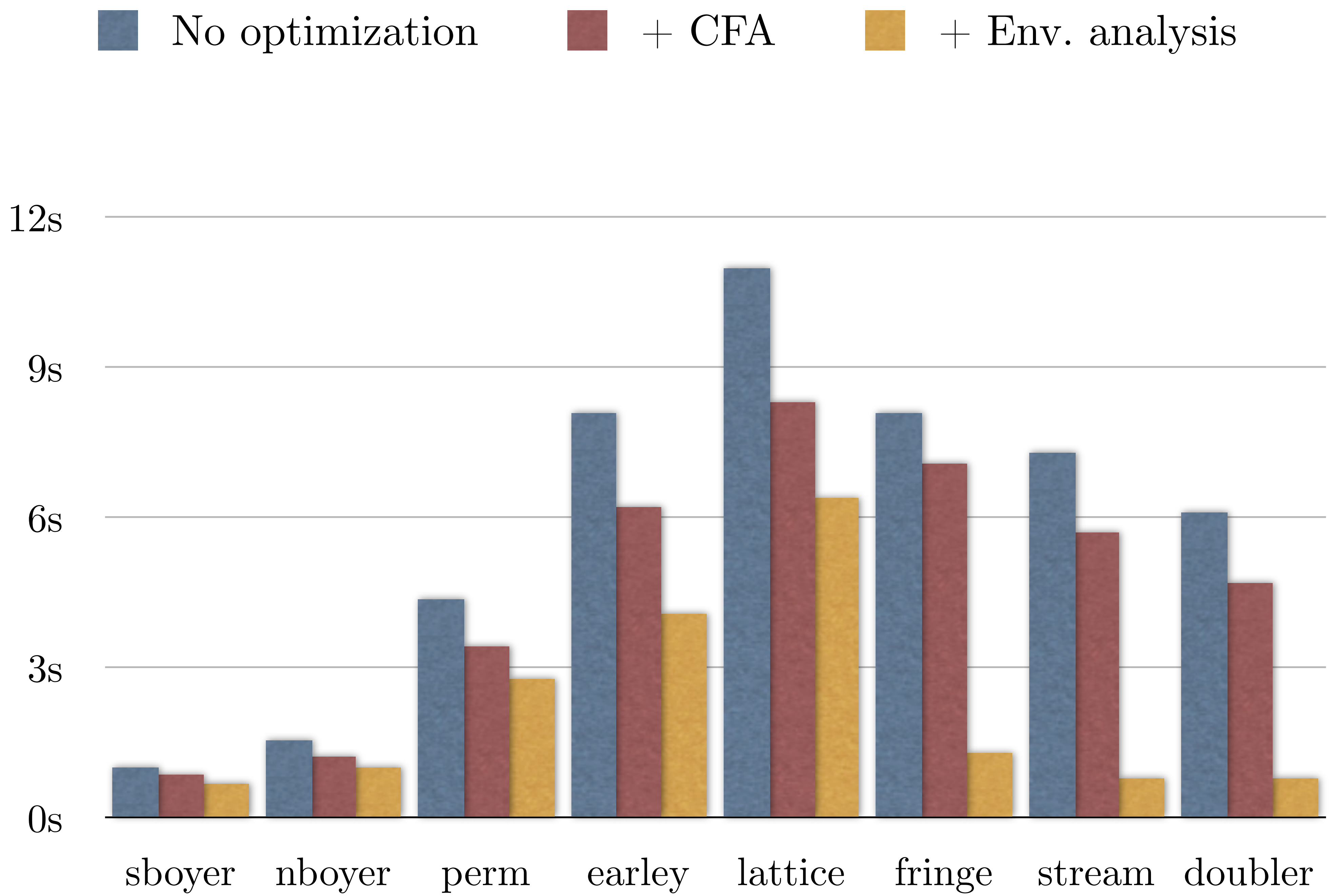
Program run times



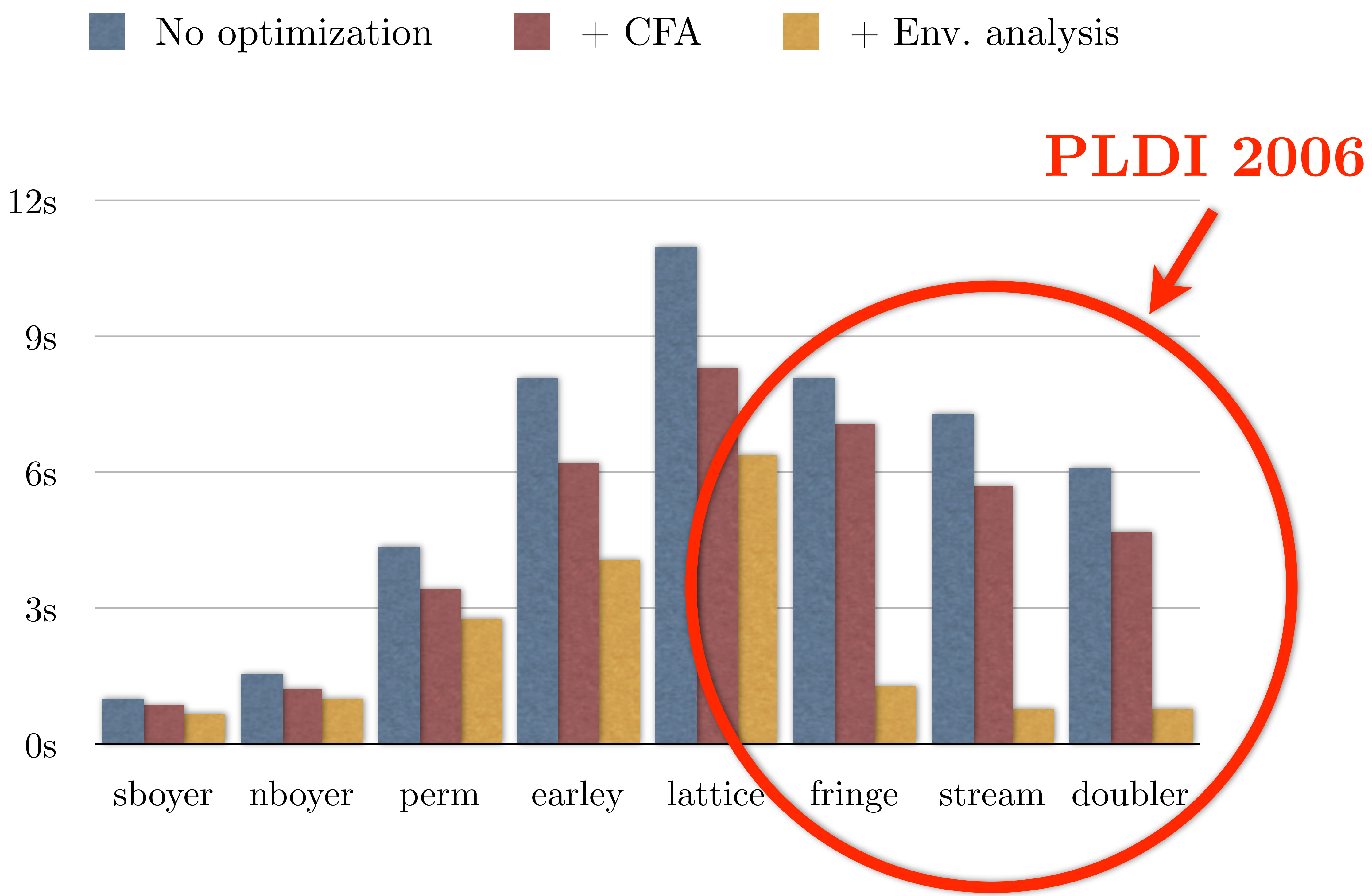
Program run times



Program run times



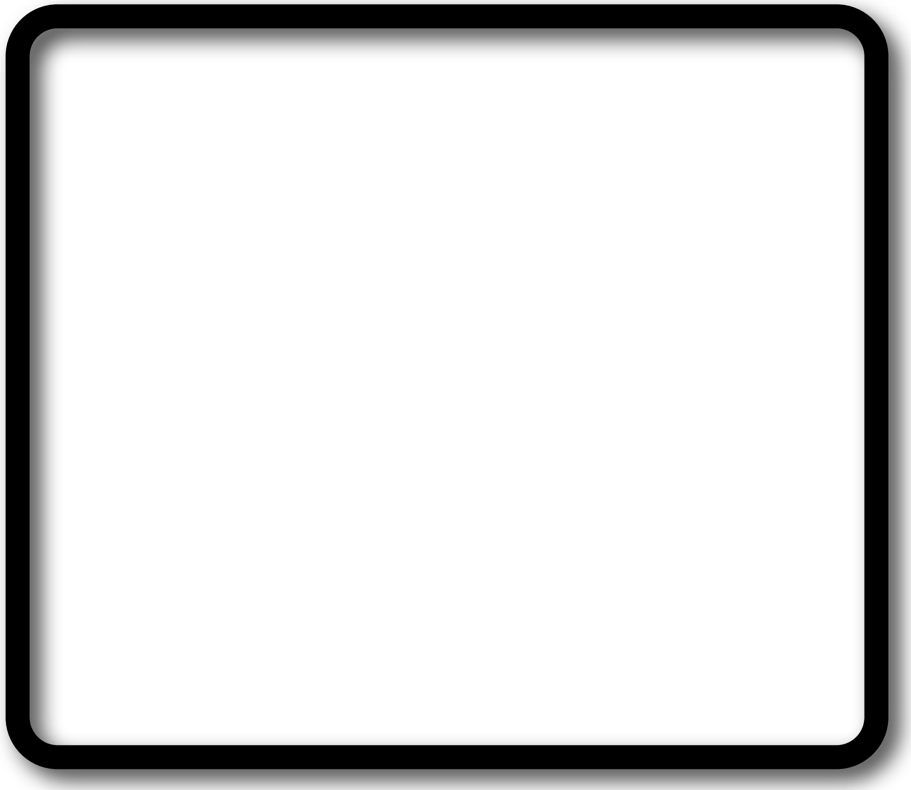
Program run times



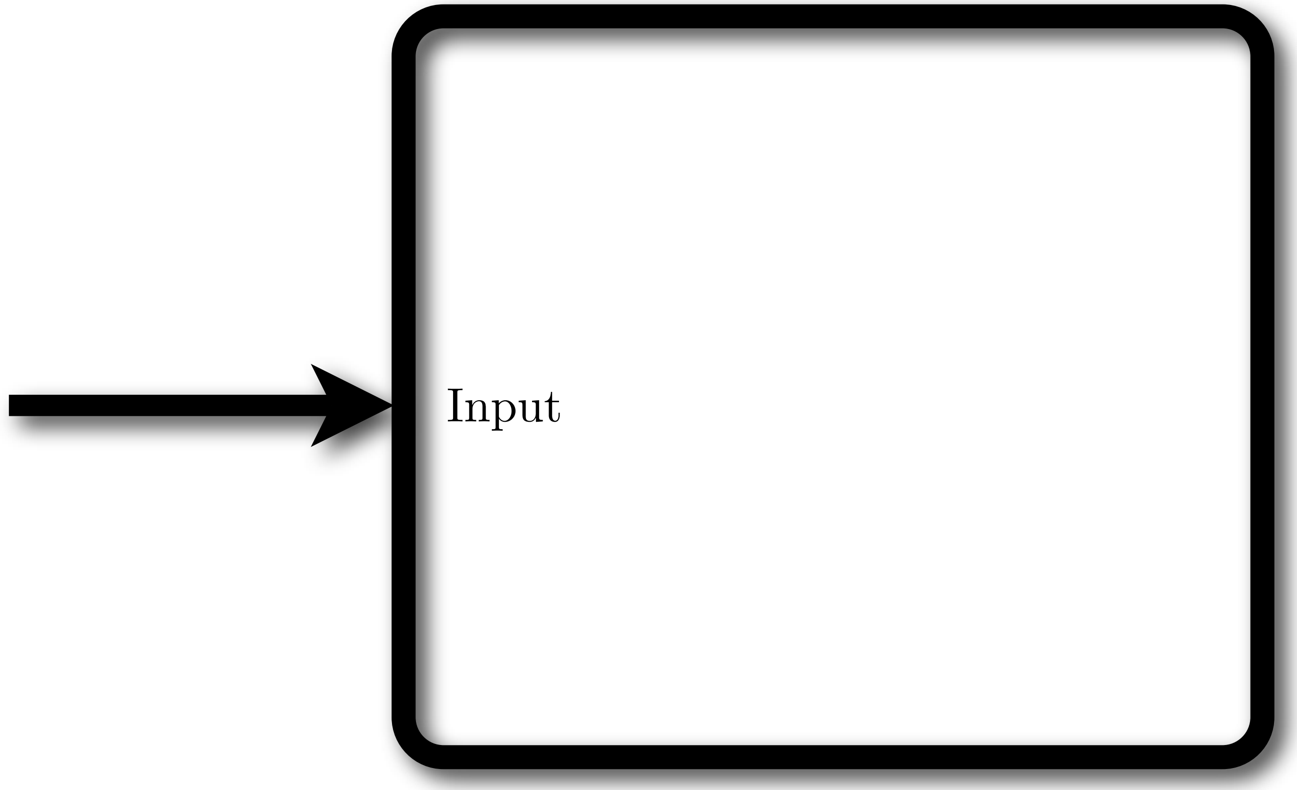
<coroutine-fusion>

Coroutine paradigm

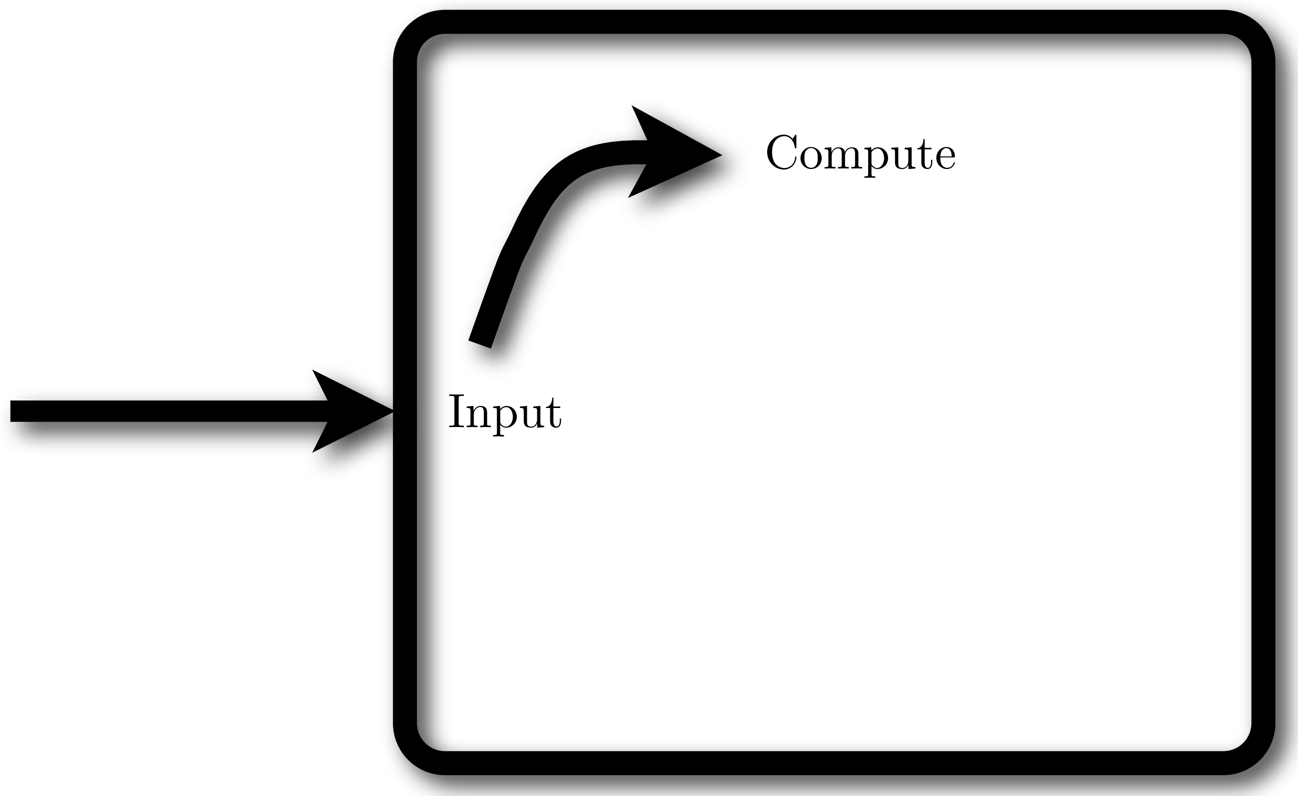
Coroutine paradigm



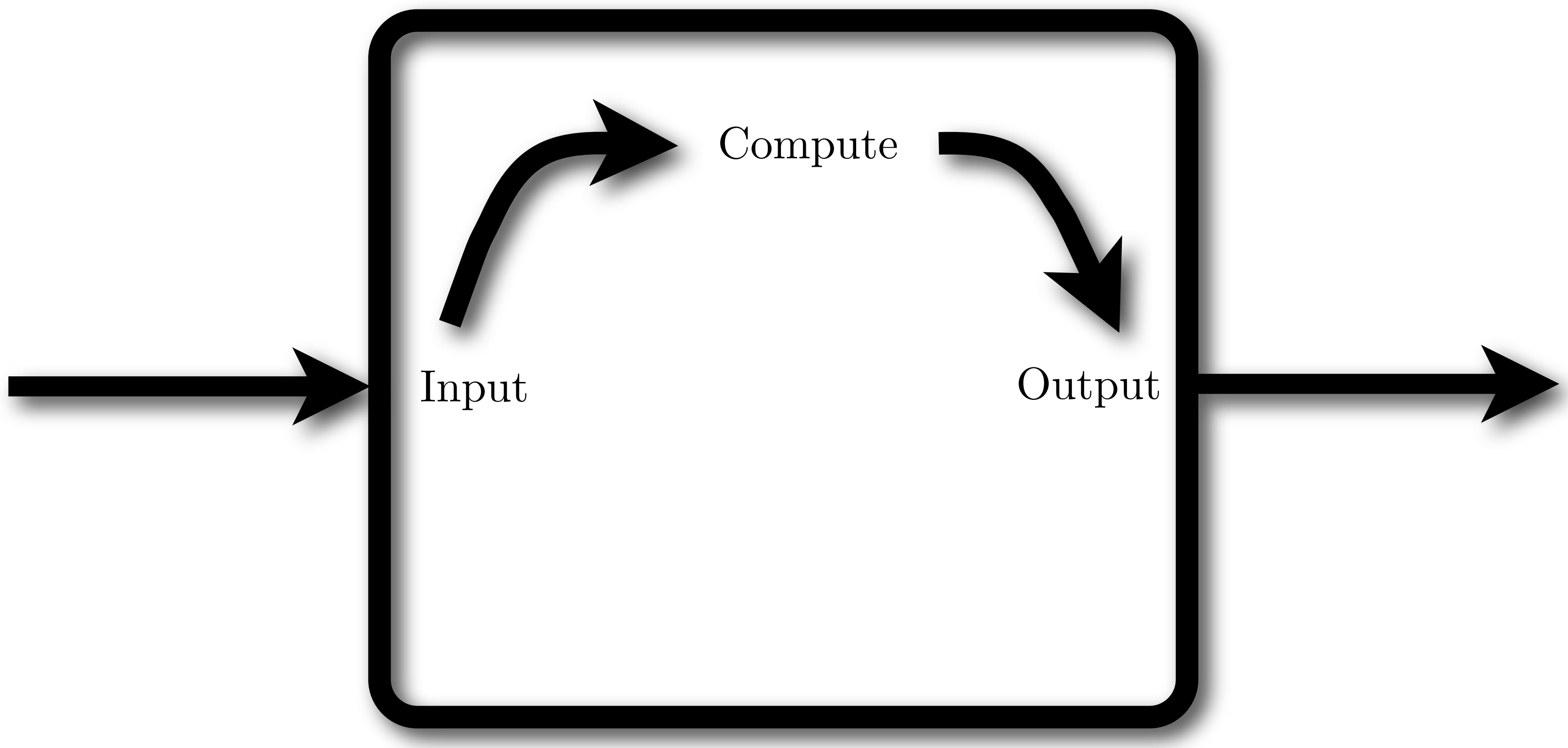
Coroutine paradigm



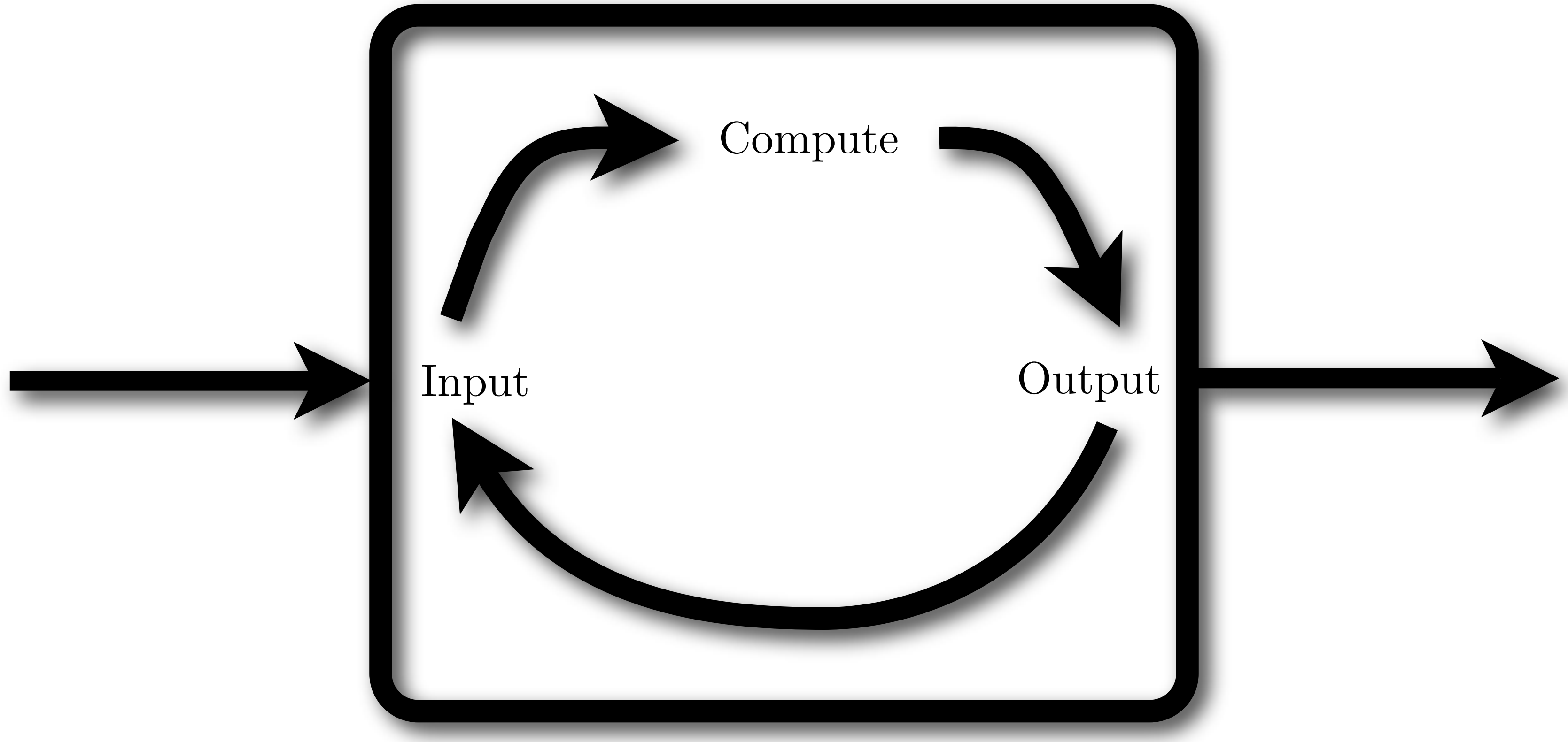
Coroutine paradigm



Coroutine paradigm



Coroutine paradigm



Coroutine API

- `get`: Receive upstream value
- `put`: Send value downstream

Coroutine API

- `get`: Receive upstream value
- `put`: Send value downstream

```
doubler() =
 put(2 × get()) ;
 doubler()
```

Coroutine API

- `get`: Receive upstream value
- `put`: Send value downstream

```
doubler() =
 put(2 × get()) ;
 doubler()
```

→ `get`

Coroutine API

- `get`: Receive upstream value
- `put`: Send value downstream

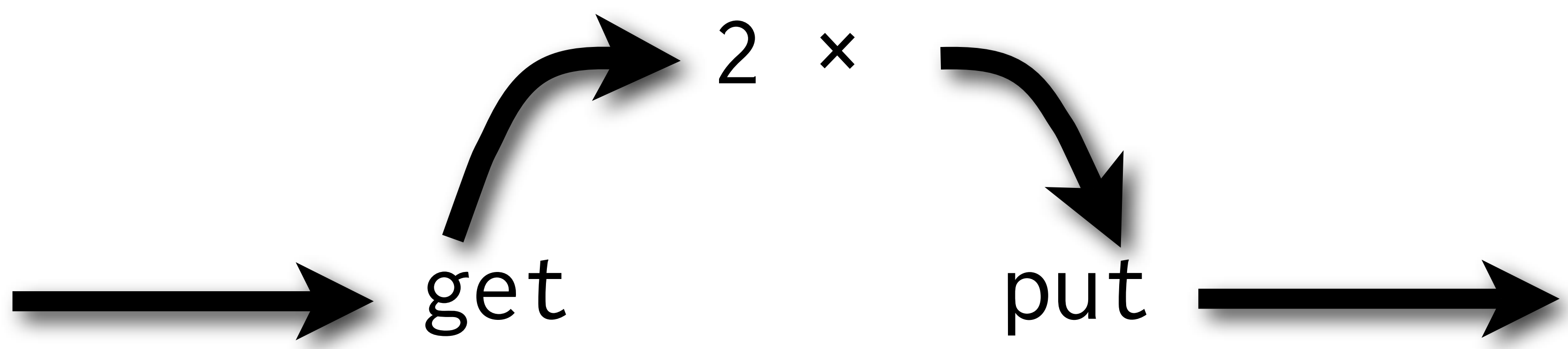
```
doubler() =
 put(2 × get()) ;
 doubler()
```

```
         2 ×
get
```

Coroutine API

- `get`: Receive upstream value
- `put`: Send value downstream

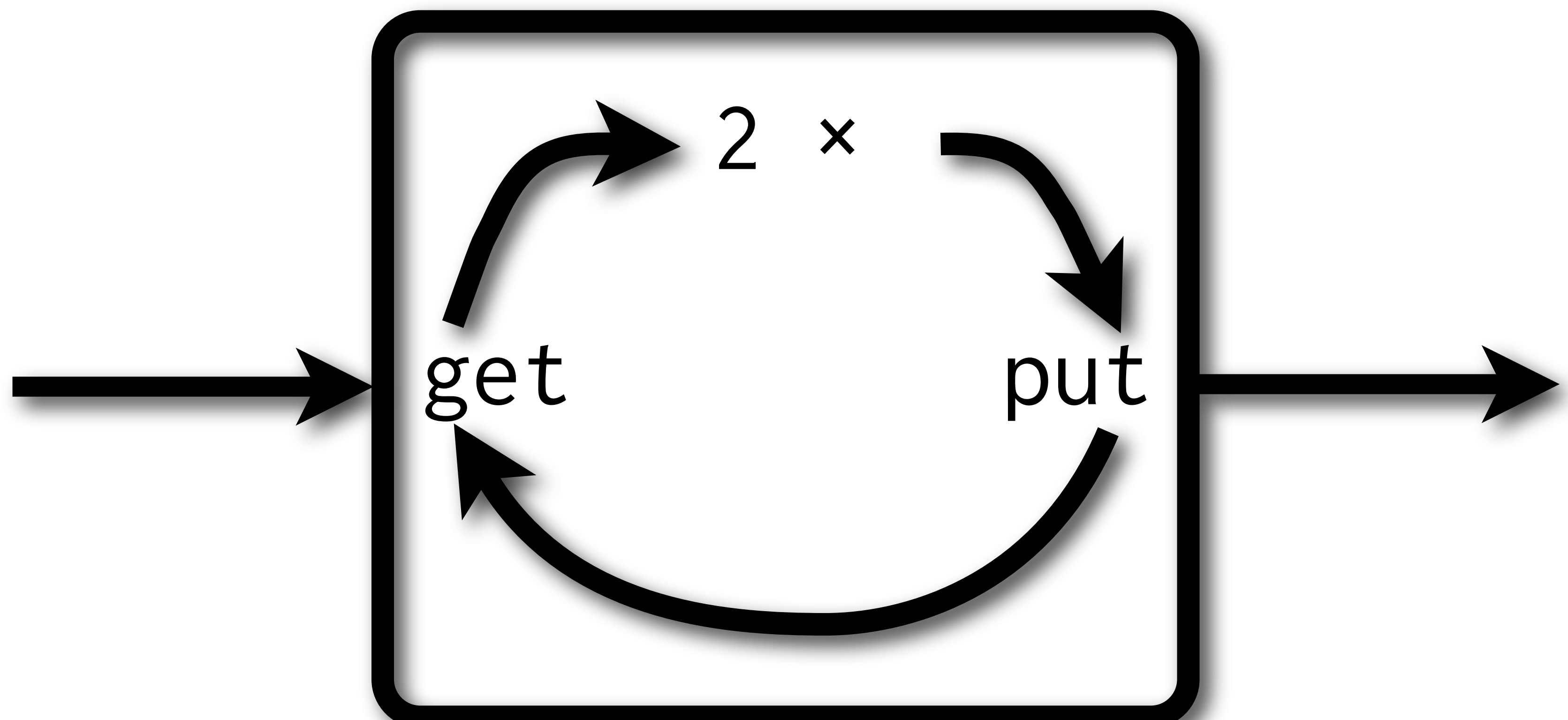
```
doubler() =
 put(2 × get()) ;
 doubler()
```



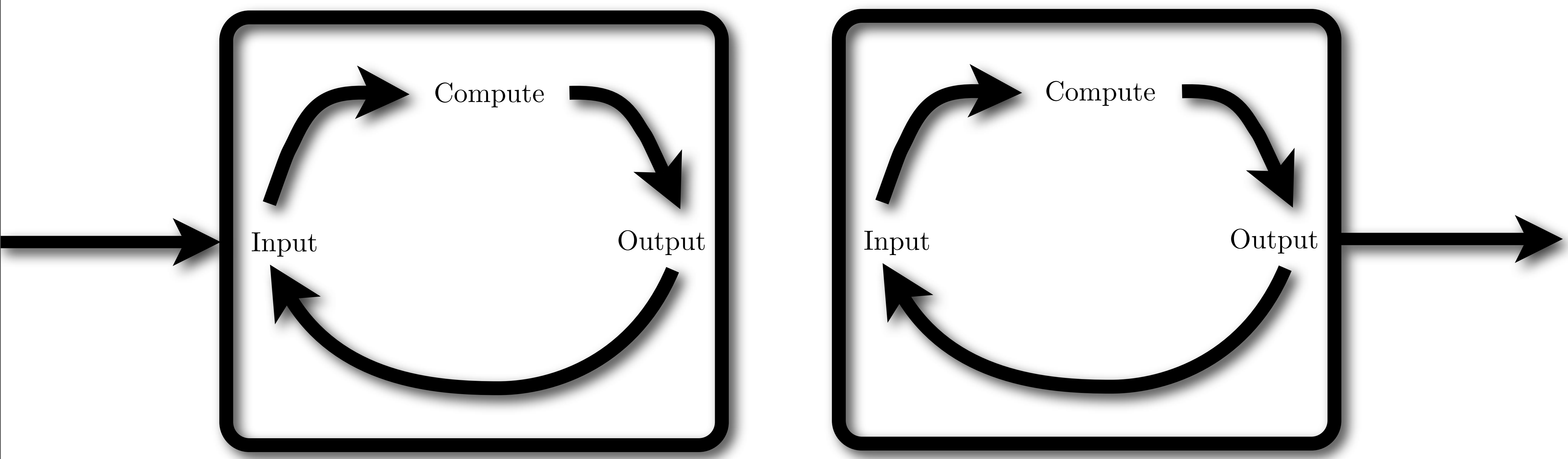
Coroutine API

- `get`: Receive upstream value
- `put`: Send value downstream

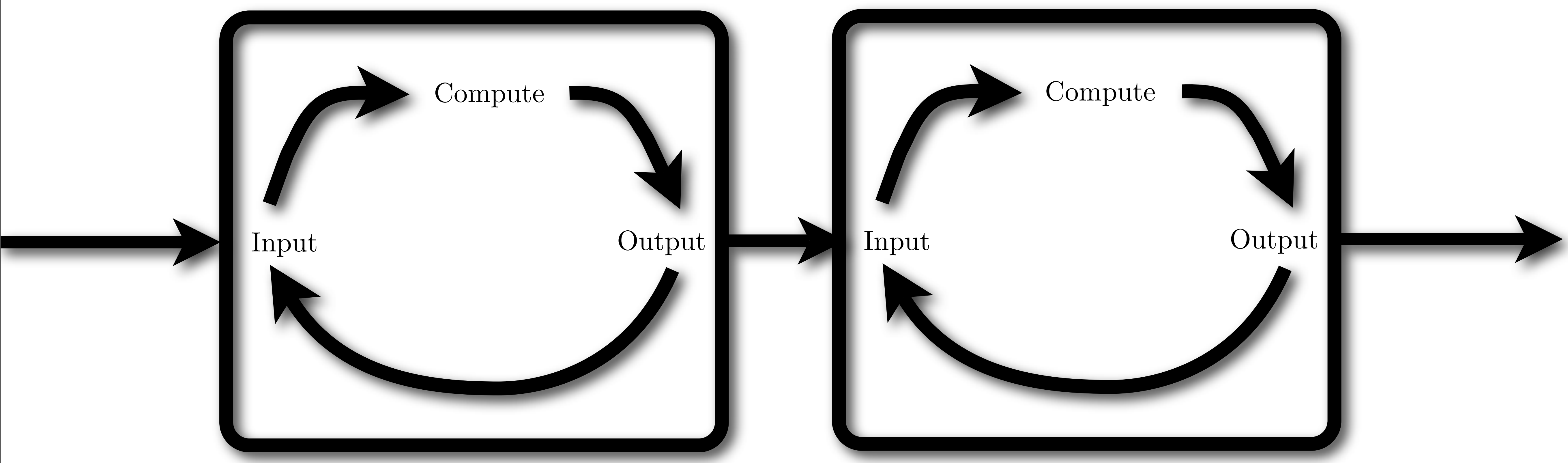
```
doubler() =
 put(2 × get()) ;
 doubler()
```



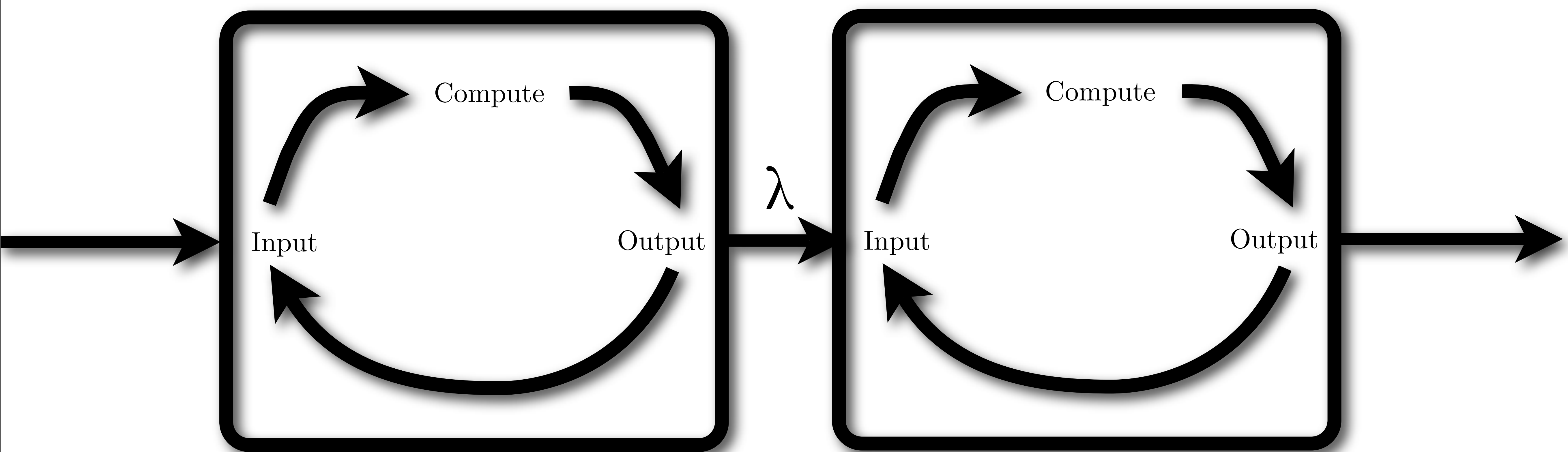
Coroutine fusion



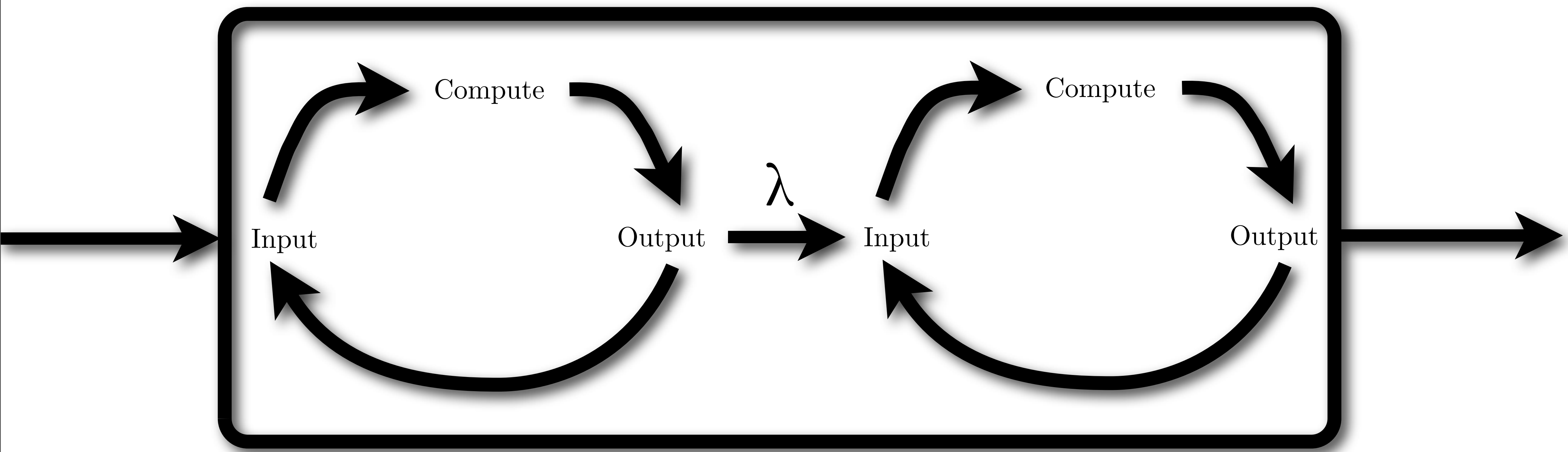
Coroutine fusion



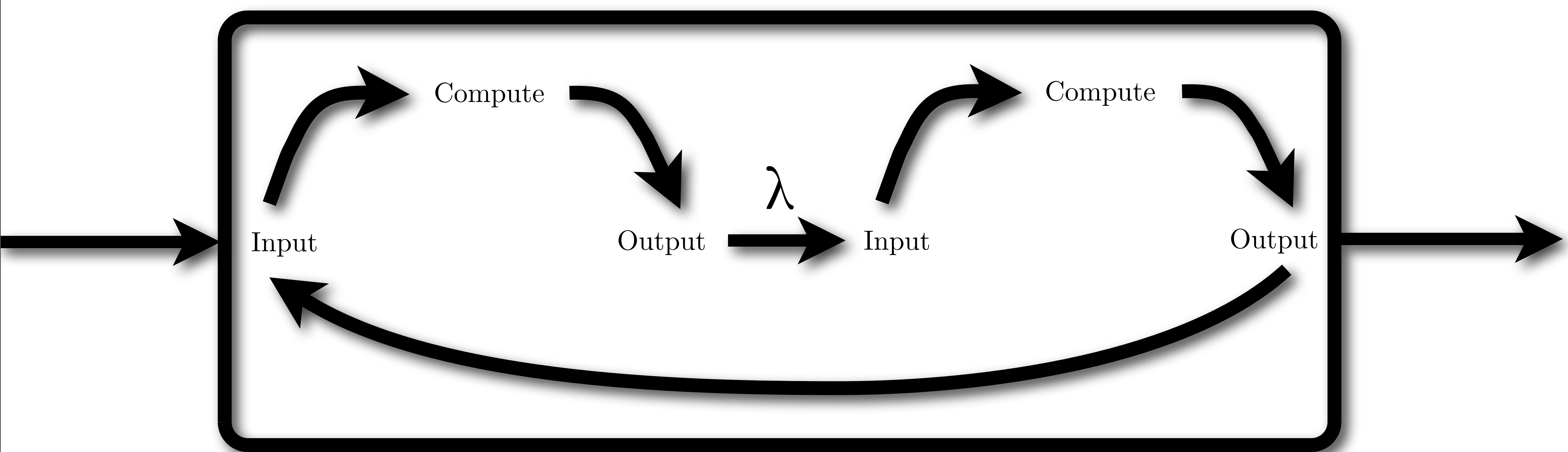
Coroutine fusion



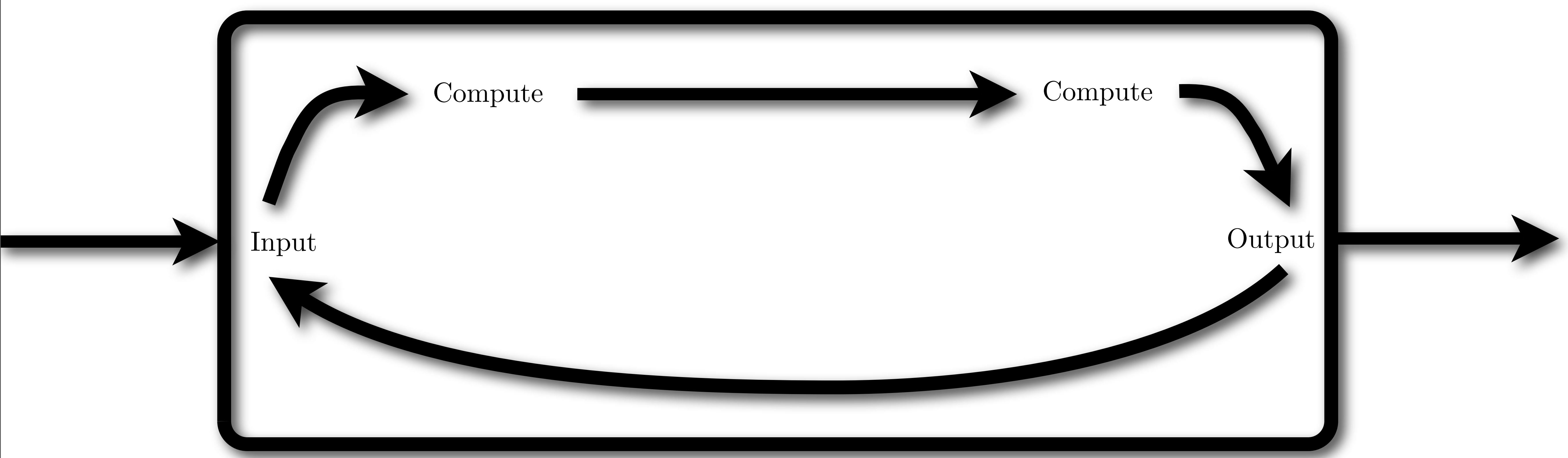
Coroutine fusion



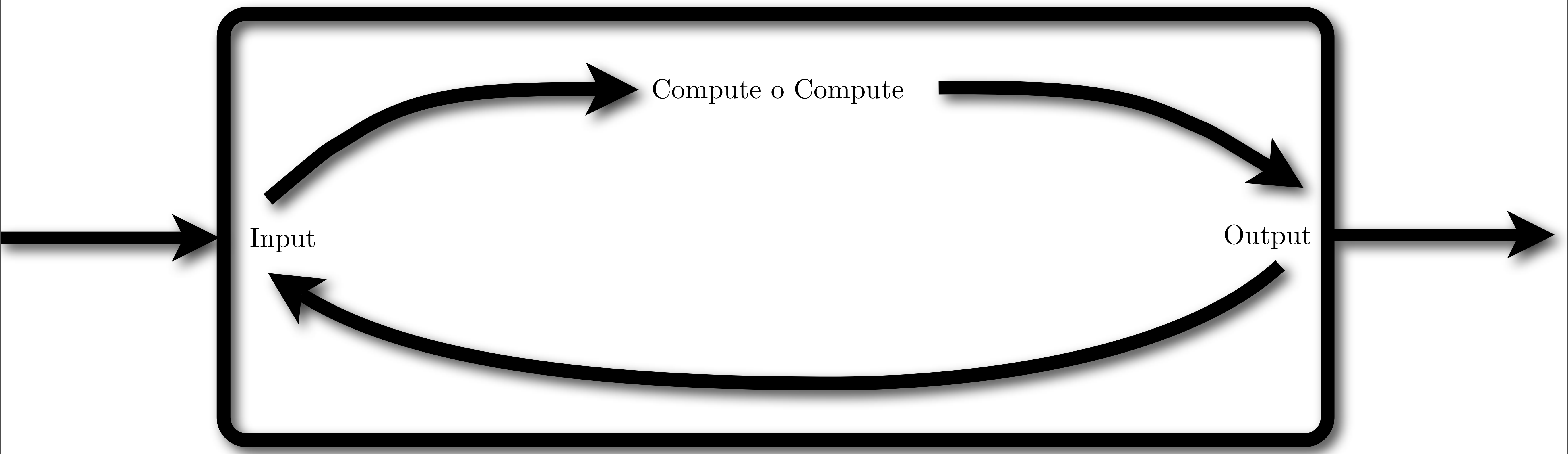
Coroutine fusion



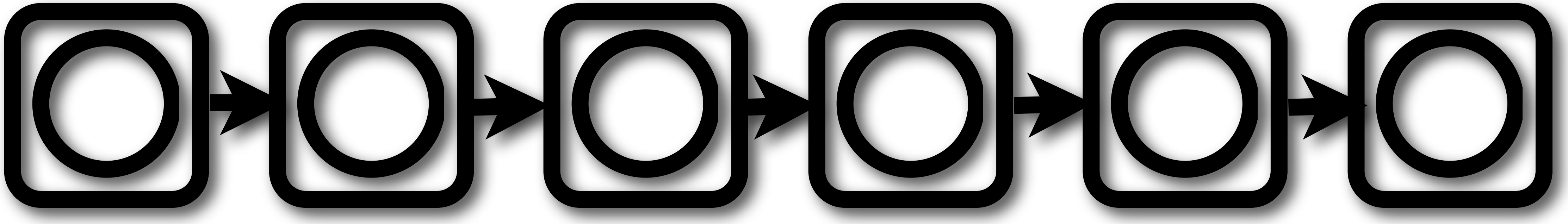
Coroutine fusion



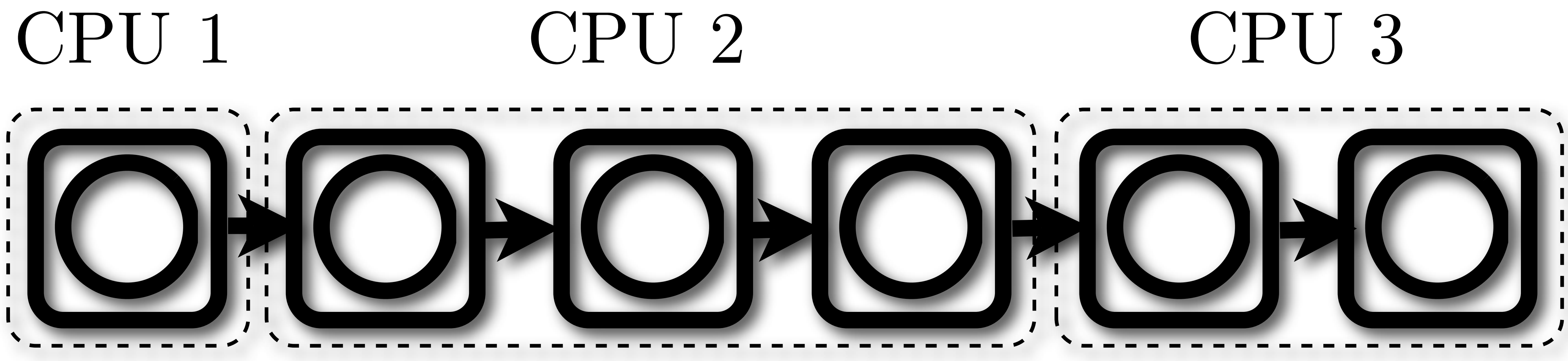
Coroutine fusion



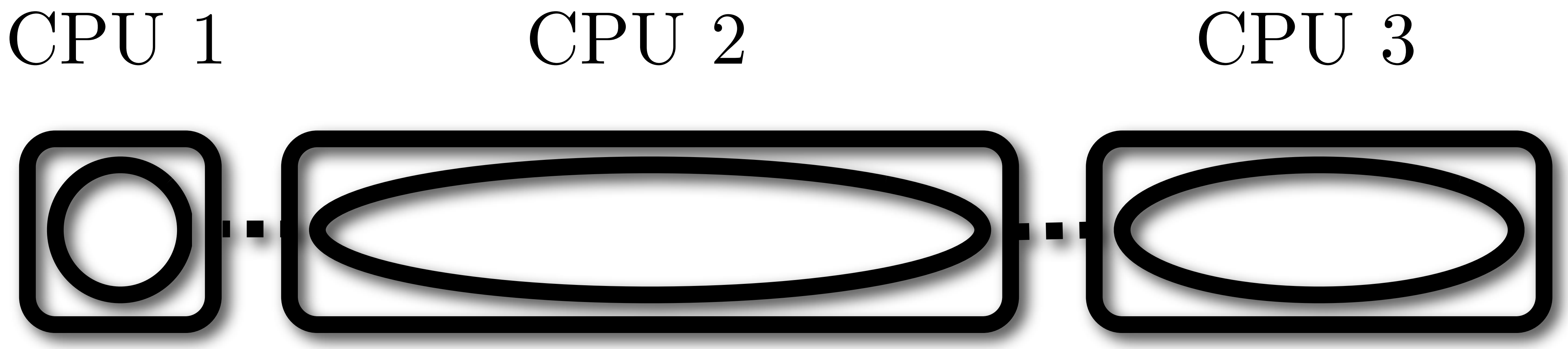
Coroutines to parallelism



Coroutines to parallelism

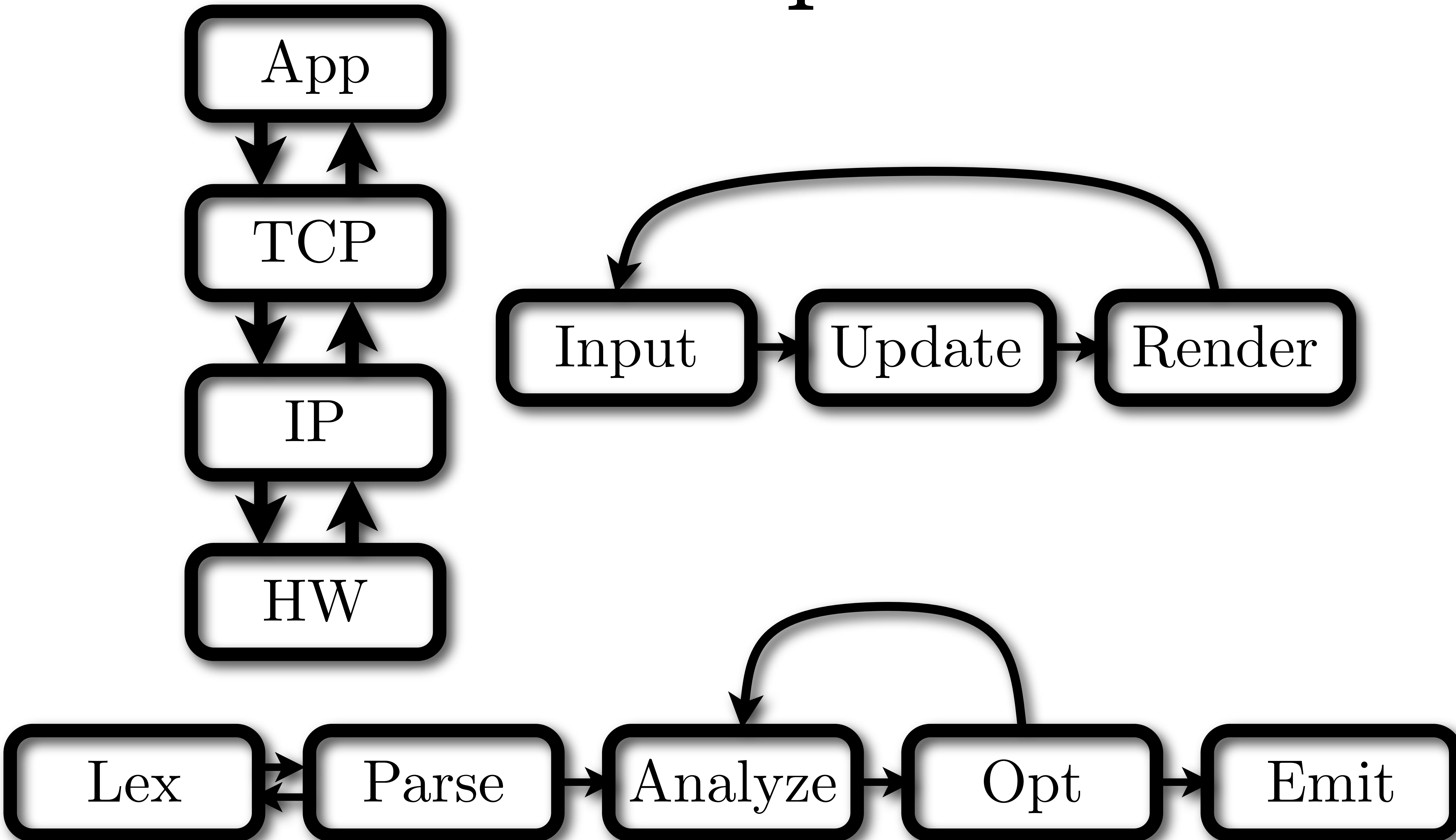


Coroutines to parallelism



Examples

Examples



</coroutine-fusion>

How to save the environment

The environment problem

Given two environments, are they equivalent?

The environment problem

Given two environments, are they equivalent?

- What does *equivalent* mean?

The environment problem

Given two environments, are they equivalent?

- What does *equivalent* mean?
- *Which* two environments?

Equivalence

env_1 is equivalent to env_2

over variables v_1, ... ,v_n

if $env_1(v_i) = env_2(v_i)$

Equivalence

o_1 is equivalent to o_2
over fields f_1, ... ,f_n
if $o_1.f_i = o_2.f_i$

Equivalence

o_1 is equivalent to o_2
over fields f_1, ... ,f_n
if $o_1.f_i = o_2.f_i$

The environment problem is a higher-order analog of the must-alias problem.

Which environments?

```
f(x) = λz.x+z

loop n =
 print f(n)(n) ;
 loop (n+2)

loop 0
```

Which environments?

```
f(x) = λz.x+z

loop n =
 print f(n)(n) ;
 loop (n+2)

loop 0
```

Which environments?

```
f(x) = λz.x+z

loop n =
 print f(n)(n) ;
 loop (n+2)

loop 0
```

x → 0

x → 2

x → 4

x → 6

x → 8

. . .

Building environment analysis

Building environment analysis

1. Build concrete machine for λ-calculus

Building environment analysis

1. Build concrete machine for λ-calculus
2. Abstract into analysis (Cousot^2, 1977)

Building environment analysis

1. Build concrete machine for λ-calculus
2. Abstract into analysis (Cousot^2, 1977)
3. Use counting to derive equivalence

Concrete machine

Concrete machine

- Convert program e into machine state s_0

$$e \downarrow s_0$$

Concrete machine

- Convert program e into machine state s_0
- Transition from state s_n to state s_{n+1}

$$\begin{tikzcd}
e \arrow[d] \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & \ldots
\end{tikzcd}$$

Abstract intepretation

$$
\begin{tikzcd}
e \arrow[d] \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & \ldots
\end{tikzcd}
$$

Abstract intepretation

$$
\begin{tikzcd}
e \arrow[d] & & & & & \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & \ldots
\end{tikzcd}
$$

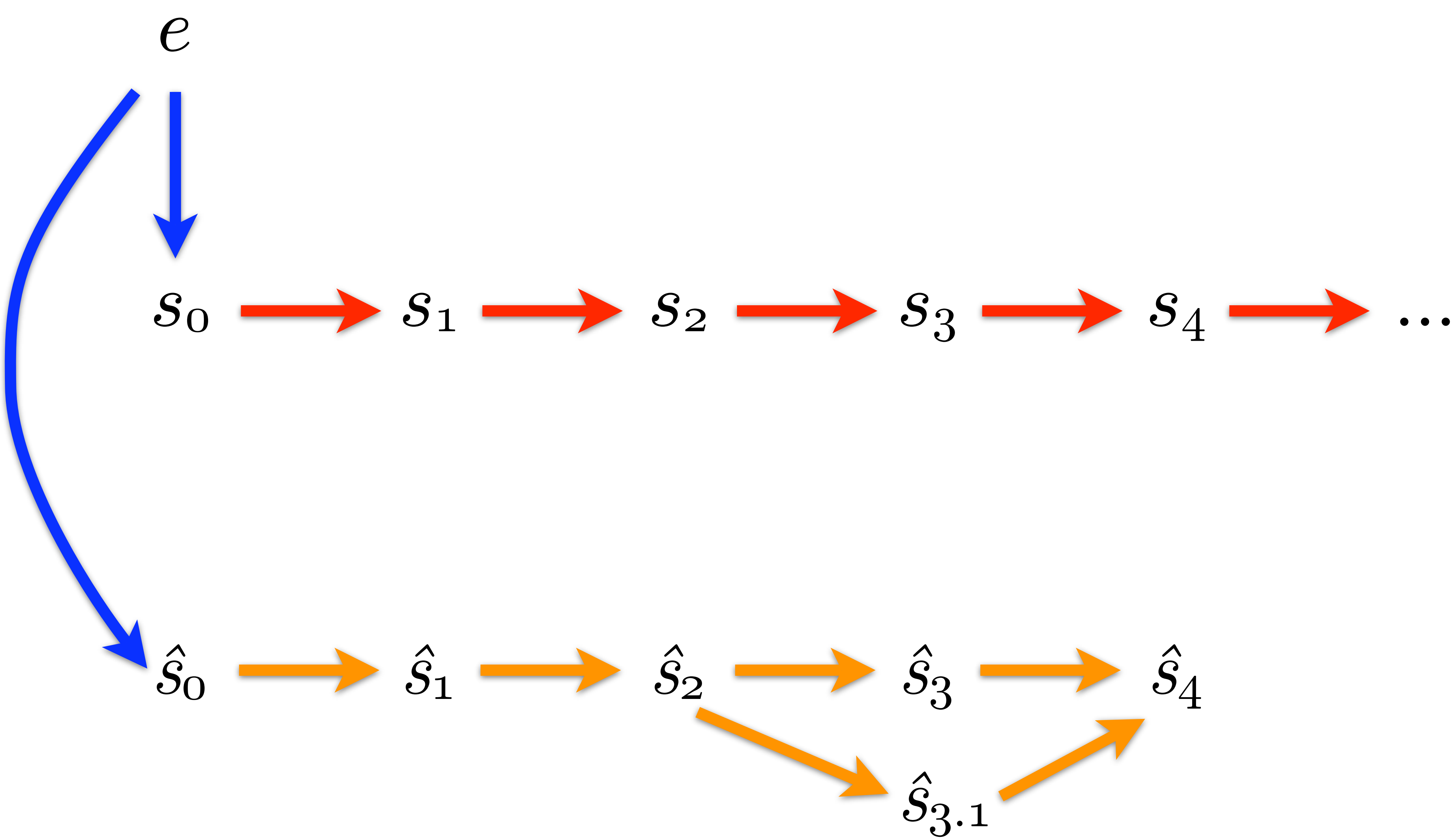
Abstract intepretation

$$
\begin{tikzcd}
e \arrow[d] \arrow[dd, bend right] \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & \ldots \\
\hat{s}_0
\end{tikzcd}
$$

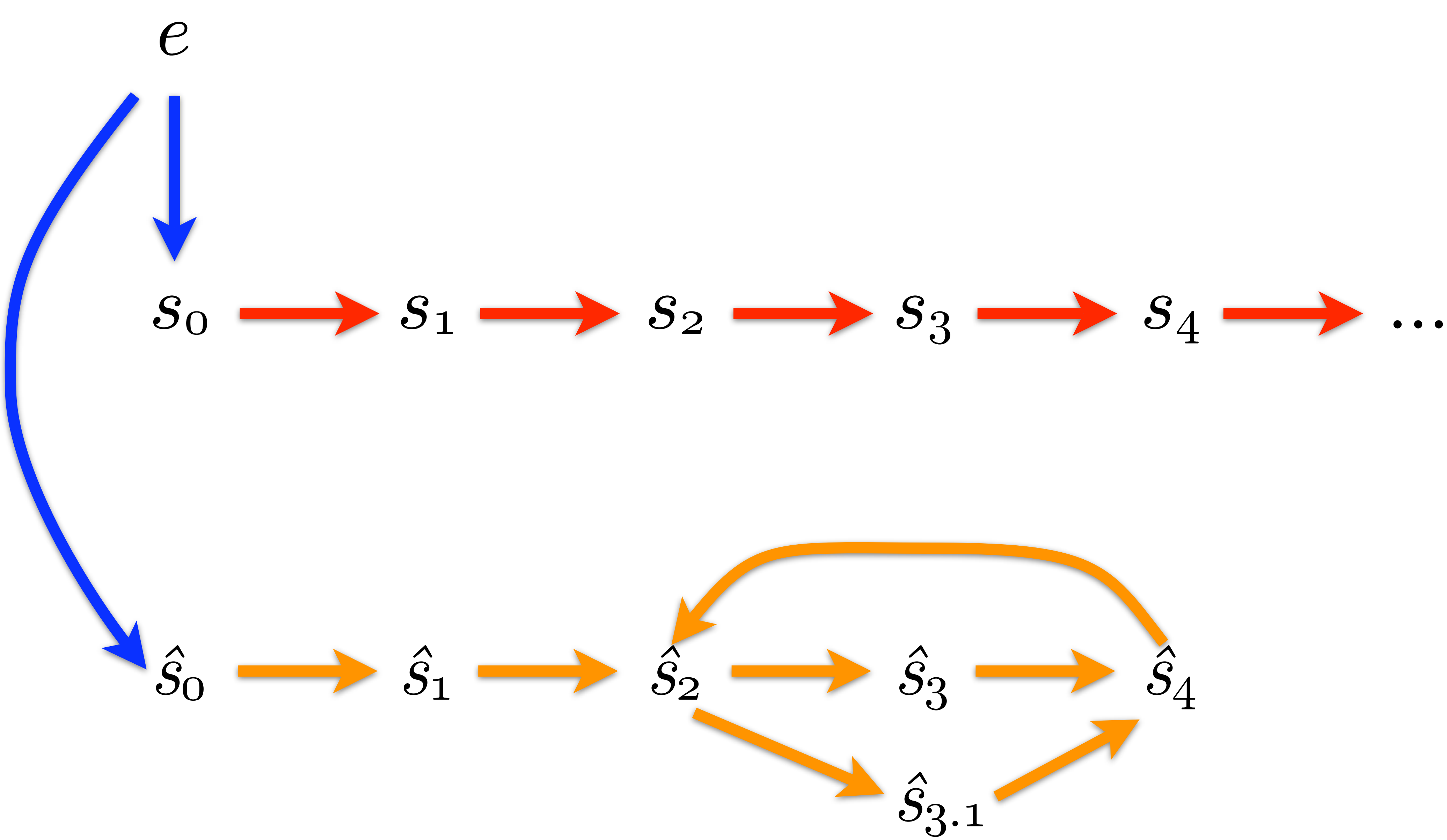
Abstract intepretation

$$
\begin{tikzcd}
e \arrow[d] \arrow[dd, bend right] & & & & & \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & \dots \\
\hat{s}_0 \arrow[r] & \hat{s}_1 \arrow[r] & \hat{s}_2 \arrow[r] \arrow[dr] & \hat{s}_3 & & \\
 & & & \hat{s}_{3.1} & &
\end{tikzcd}
$$

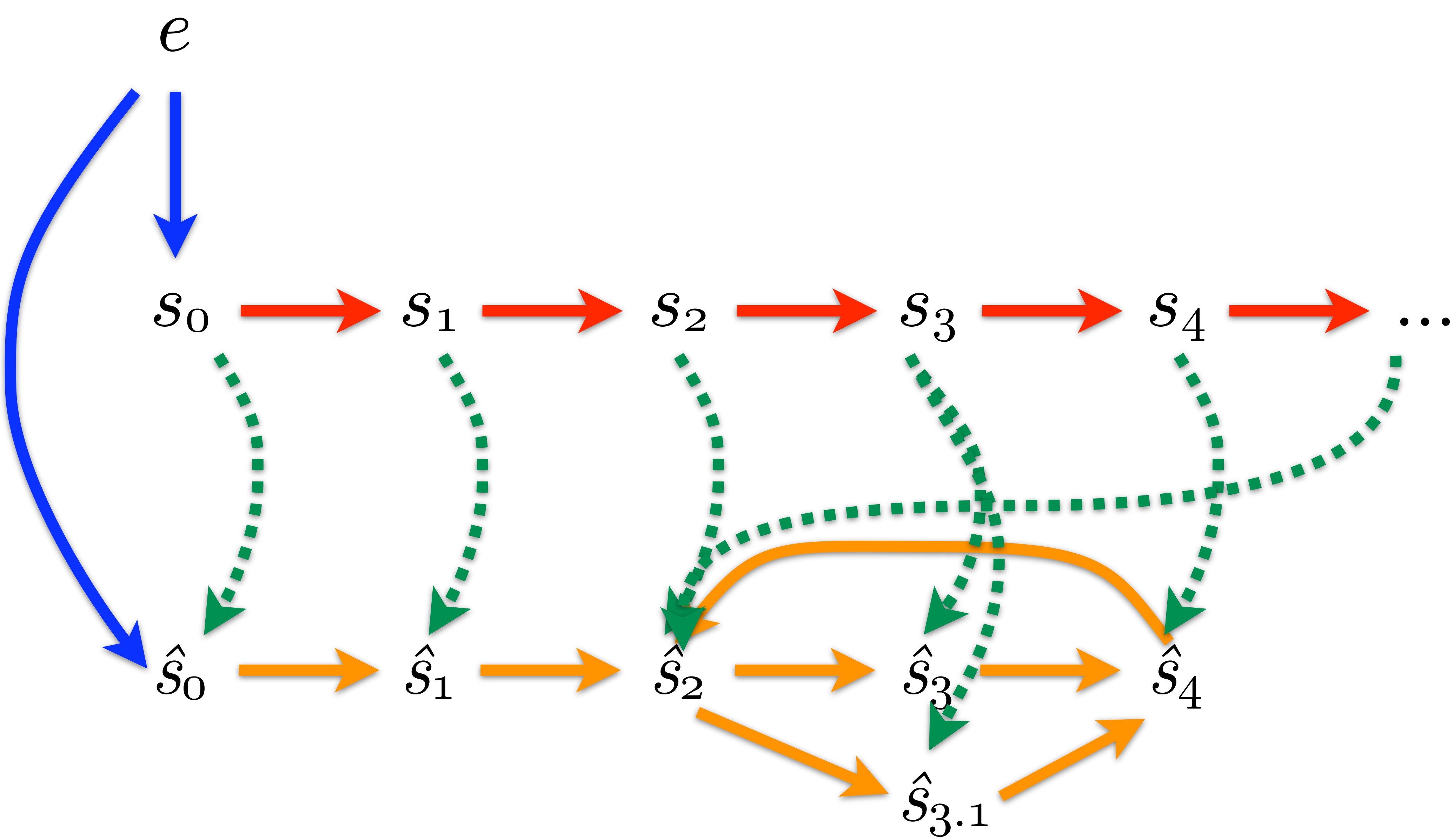
Abstract intepretation



Abstract intepretation



Abstract intepretation



Theorem: The abstract simulates the concrete.

CPS λ-calculus syntax

- Two expression term types (e)
  - v
  - $\lambda v_1 \ldots v_n.call$
- One call term type ($call$)
  - $(e_0\ e_1 \ldots e_n)$

Concrete semantics (1)

Concrete semantics (1)

- Machine state: $(call,\ env,\ mem)$

Concrete semantics (1)

- Machine state: $(call, env, mem)$
  - env : Var $\to$ Addr [address of variable]

Concrete semantics (1)

- Machine state: $(call, env, mem)$
  - env : Var $\rightarrow$ Addr [address of variable]
  - mem : Addr $\rightarrow$ Value [value of address]

Concrete semantics (1)

- Machine state: $(call,\ env,\ mem)$
  - env : Var $\rightarrow$ Addr [address of variable]
  - mem : Addr $\rightarrow$ Value [value of address]
- eval$(v, env, mem) = mem(env(v))$

Concrete semantics (1)

- Machine state: $(call, env, mem)$
  - env : Var $\to$ Addr [address of variable]
  - mem : Addr $\to$ Value [value of address]
- $\text{eval}(v, env, mem) = mem(env(v))$
- $\text{eval}(\lambda, env, mem) = (\lambda, env)$

Concrete semantics (2)

$$(\llbracket (e_0\ e_1\ \ldots\ e_n) \rrbracket,\ env,\ mem) \Rightarrow (call,\ env'',mem')$$

Concrete semantics (2)

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ env,\ mem) \Rightarrow (call,\ env'',mem')$$

$$val_i = \mathrm{eval}(e_i,\ env,\ mem)$$

Concrete semantics (2)

$$(\llbracket(e_0\ e_1\ \ldots\ e_n)\rrbracket,\ env,\ mem) \Rightarrow (call,\ env'',mem')$$

$$val_i = \mathrm{eval}(e_i,\ env,\ mem)$$
$$val_0 = (\llbracket\lambda v_1\ \ldots\ v_n.call\rrbracket,env')$$

Concrete semantics (2)

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ env,\ mem) \Rightarrow (call,\ env'',mem')$$

$$val_i = \text{eval}(e_i,\ env,\ mem)$$
$$val_0 = ([\![\lambda v_1\ \ldots\ v_n.call]\!],env')$$
$$a_i = \text{alloc}(s,\ v_i)$$

Concrete semantics (2)

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ env,\ mem) \Rightarrow (call,\ env'', mem')$$

$$\begin{aligned} val_i &= \mathrm{eval}(e_i,\ env,\ mem) \\ val_0 &= ([\![\lambda v_1\ \ldots\ v_n.call]\!], env') \\ a_i &= \mathrm{alloc}(s,\ v_i) \\ env'' &= env'[v_i \rightarrow a_i] \end{aligned}$$

Concrete semantics (2)

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ env,\ mem) \Rightarrow (call,\ env'',mem')$$

$$\begin{aligned}
val_i &= \mathrm{eval}(e_i,\ env,\ mem) \\
val_0 &= ([\![\lambda v_1\ \ldots\ v_n.call]\!],env') \\
a_i &= \mathrm{alloc}(s,\ v_i) \\
env'' &= env'[v_i \to\ a_i] \\
mem' &= mem[a_i \to\ val_i]
\end{aligned}$$

Abstract semantics

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ env,\ mem) \Rightarrow (call,\ env'',mem')$$

$$\begin{aligned}
val_i &= \mathrm{eval}(e_i,\ env,\ mem) \\
val_0 &= ([\![\lambda v_1\ \ldots\ v_n.call]\!],env') \\
a_i &= \mathrm{alloc}(s,\ v_i) \\
env'' &= env'[v_i \rightarrow\ a_i] \\
mem' &= mem[a_i \rightarrow\ val_i]
\end{aligned}$$

Abstract semantics

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ \overline{env},\ \overline{mem}) \Rightarrow (call,\ \overline{env}'',\overline{mem}')$$

$$\begin{aligned}
\overline{val}_i &= \overline{\mathrm{eval}}(e_i,\ \overline{env},\ \overline{mem}) \\
\overline{val}_0 &= ([\![\lambda v_1\ \ldots\ v_n.call]\!],\overline{env}') \\
\overline{a}_i &= \overline{\mathrm{alloc}}(\overline{s},\ v_i) \\
\overline{env}'' &= \overline{env}'[v_i \to \overline{a}_i] \\
\overline{mem}' &= \overline{mem}[\overline{a}_i \to \overline{val}_i]
\end{aligned}$$

Abstract semantics

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ \overline{env},\ \overline{mem}) \Rightarrow (call,\ \overline{env}'',\overline{mem}')$$

$$\overline{val}_i = \overline{\mathrm{eval}}(e_i,\ \overline{env},\ \overline{mem})$$

$$\overline{val}_0 = ([\![\lambda v_1\ \ldots\ v_n.call]\!],\overline{env}')$$

$$\overline{a}_i = \overline{\mathrm{alloc}}(\overline{s},\ v_i)$$

$$\overline{env}'' = \overline{env}'[v_i \to \overline{a}_i]$$

$$\overline{mem}' = \overline{mem}[\overline{a}_i \to \overline{val}_i]$$

$$\overline{\mathrm{Addr}} = \mathrm{finite}$$

Abstract semantics

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ \overline{env},\ \overline{mem}) \Rightarrow (call,\ \overline{env}'',\overline{mem}')$$

$$\overline{val}_i = \overline{\mathrm{eval}}(e_i,\ \overline{env},\ \overline{mem})$$
$$\overline{val}_0 = ([\![\lambda v_1\ \ldots\ v_n.call]\!],\overline{env}')$$
$$\bar{a}_i = \overline{\mathrm{alloc}}(\bar{s},\ v_i)$$
$$\overline{env}'' = \overline{env}'[v_i \to \bar{a}_i]$$
$$\overline{mem}' = \overline{mem}\ \sqcup\ [\bar{a}_i \to \overline{val}_i]$$

$$\overline{\mathrm{Addr}} = \text{finite}$$

Abstract semantics

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ \overline{env},\ \overline{mem}) \Rightarrow (call,\ \overline{env}'',\overline{mem}')$$

$$\overline{val}_i = \overline{\text{eval}}(e_i,\ \overline{env},\ \overline{mem})$$

$$\overline{val}_0 = ([\![\lambda v_1\ \ldots\ v_n.call]\!],\overline{env}')$$

$$\bar{a}_i = \overline{\text{alloc}}(\bar{s},\ v_i)$$

$$\overline{env}'' = \overline{env}'[v_i \to \bar{a}_i]$$

$$\overline{mem}' = \overline{mem}\ \sqcup\ [\bar{a}_i \to \overline{val}_i]$$

$$\overline{\text{Addr}} = \text{finite}$$

$$\text{Value} = \text{Closure}$$

Abstract semantics

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ \overline{env},\ \overline{mem}) \Rightarrow (call,\ \overline{env}'',\overline{mem}')$$

$$\begin{aligned}
\overline{val}_i &= \overline{\text{eval}}(e_i,\ \overline{env},\ \overline{mem})\\
\overline{val}_0 &= ([\![\lambda v_1\ \ldots\ v_n.call]\!],\overline{env}')\\
\bar{a}_i &= \overline{\text{alloc}}(\bar{s},\ v_i)\\
\overline{env}'' &= \overline{env}'[v_i \to \bar{a}_i]\\
\overline{mem}' &= \overline{mem}\ \sqcup\ [\bar{a}_i \to \overline{val}_i]
\end{aligned}$$

$$\begin{aligned}
\overline{\text{Addr}} &= \text{finite}\\
\overline{\text{Value}} &= \mathcal{P}(\overline{\text{Closure}}\ )
\end{aligned}$$

Universal CFA

$$([\![(e_0\ e_1\ \ldots\ e_n)]\!],\ \overline{env},\ \overline{mem}) \Rightarrow (call,\ \overline{env}'',\overline{mem}')$$

$$\overline{val}_i = \overline{\text{eval}}(e_i,\ \overline{env},\ \overline{mem})$$

$$\overline{val}_0 \ni ([\![\lambda v_1\ \ldots\ v_n.call]\!],\overline{env}')$$

$$\bar{a}_i = \overline{\text{alloc}}(\bar{s},\ v_i)$$

$$\overline{env}'' = \overline{env}'[v_i \rightarrow \bar{a}_i]$$

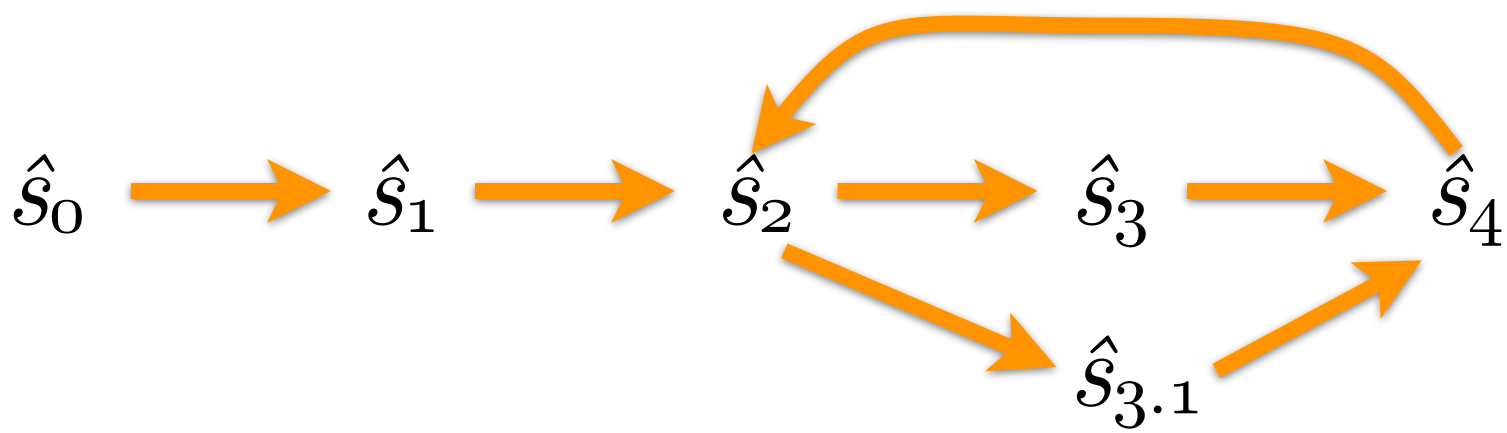
$$\overline{mem}' = \overline{mem}\ \sqcup\ [\bar{a}_i \rightarrow \overline{val}_i]$$

$$\overline{\text{Addr}} = \text{finite}$$

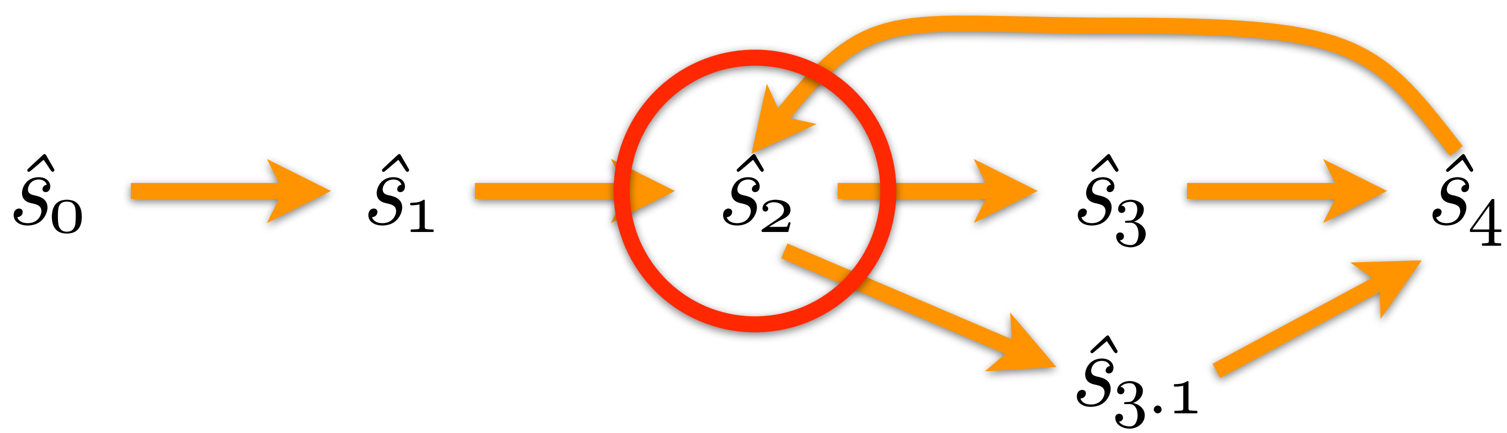
$$\overline{\text{Value}} = \mathcal{P}(\overline{\text{Closure}})$$

Which environments

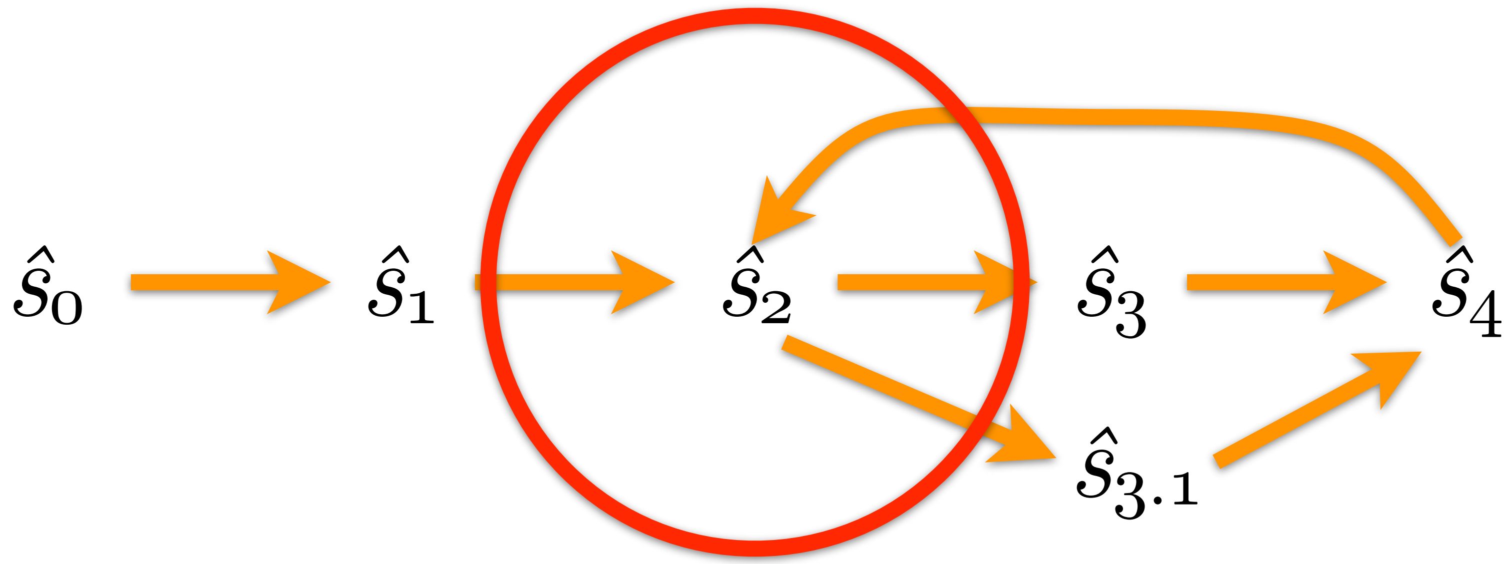
Which environments



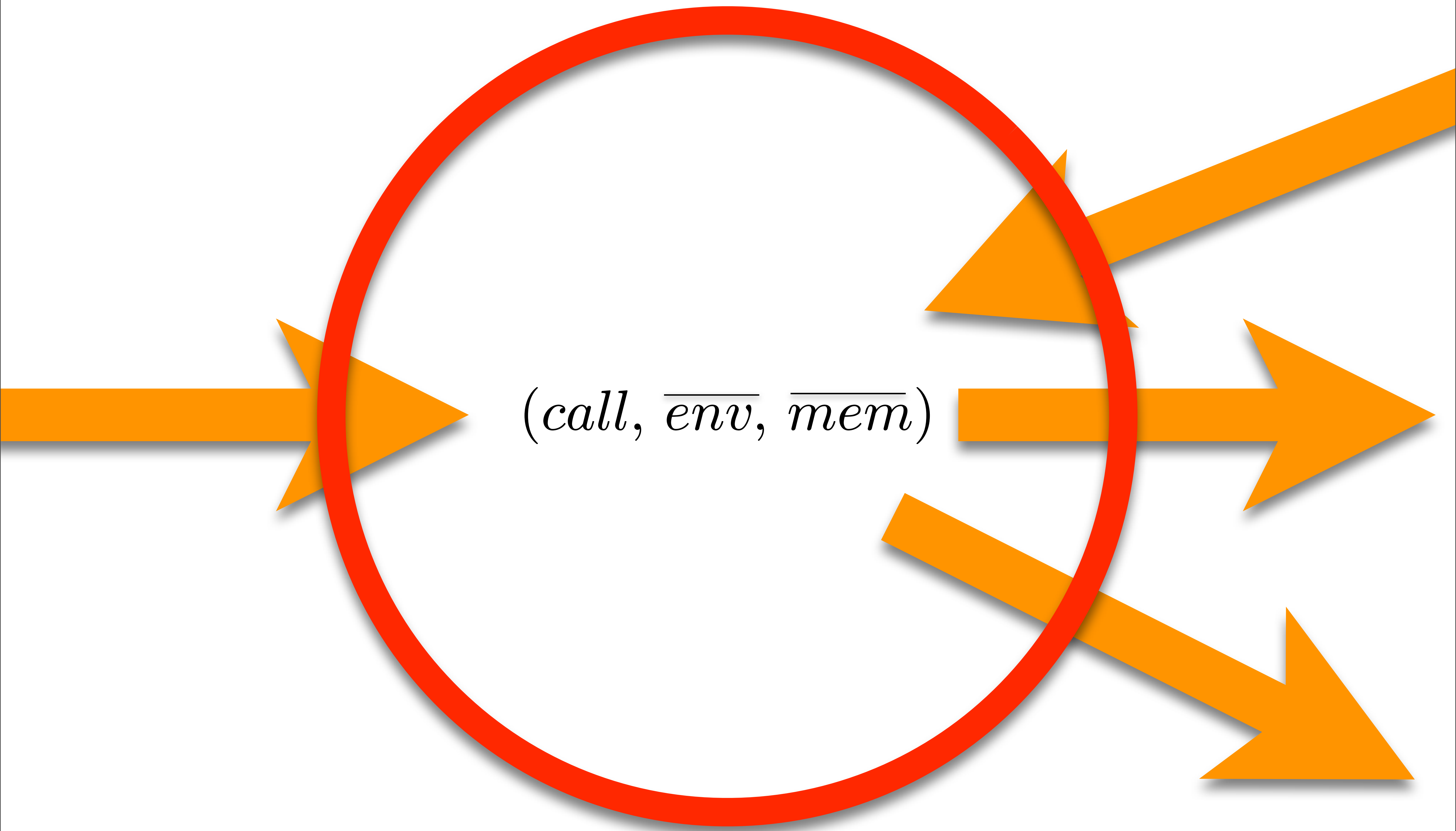
Which environments



Which environments



Which environments



Which environments

$$(call, \overline{env}, \overline{mem})$$

Which environments

$$(call, \overline{env}, \overline{mem})$$

Which environments

$(\lambda_1, [\mathsf{x} \to \bar{a}_1])$ $(\lambda_2, [\mathsf{x} \to \bar{a}_2])$

$$(call, \overline{env}, \overline{mem})$$

Which environments

$(\lambda_1, [x \to \bar{a}_1])$ $(\lambda_2, [x \to \bar{a}_2])$

If $\bar{a}_1 = \bar{a}_2$, are
the concrete environments
they represent equivalent?

Which environments

$(\lambda_1, [\mathsf{x} \to \bar{a}_1])$ $\quad\quad$ $(\lambda_2, [\mathsf{x} \to \bar{a}_2])$

$$\{a_1,\ a_2\}$$

If $\bar{a}_1 = \bar{a}_2$, are
the concrete environments
they represent equivalent?

Maybe.

Which environments

$(\lambda_1, [\mathsf{x} \to \bar{a}_1])$ $\qquad\qquad$ $(\lambda_2, [\mathsf{x} \to \bar{a}_2])$

$$\{\ a\ \}$$

If $\bar{a}_1 = \bar{a}_2$, are
the concrete environments
they represent equivalent?

Yes!

Which environments

$(\lambda_1, [x \to \bar{a}_1])$ $(\lambda_2, [x \to \bar{a}_2])$

If $\bar{a}_1 = \bar{a}_2$, are
the concrete environments
they represent equivalent?

It depends.

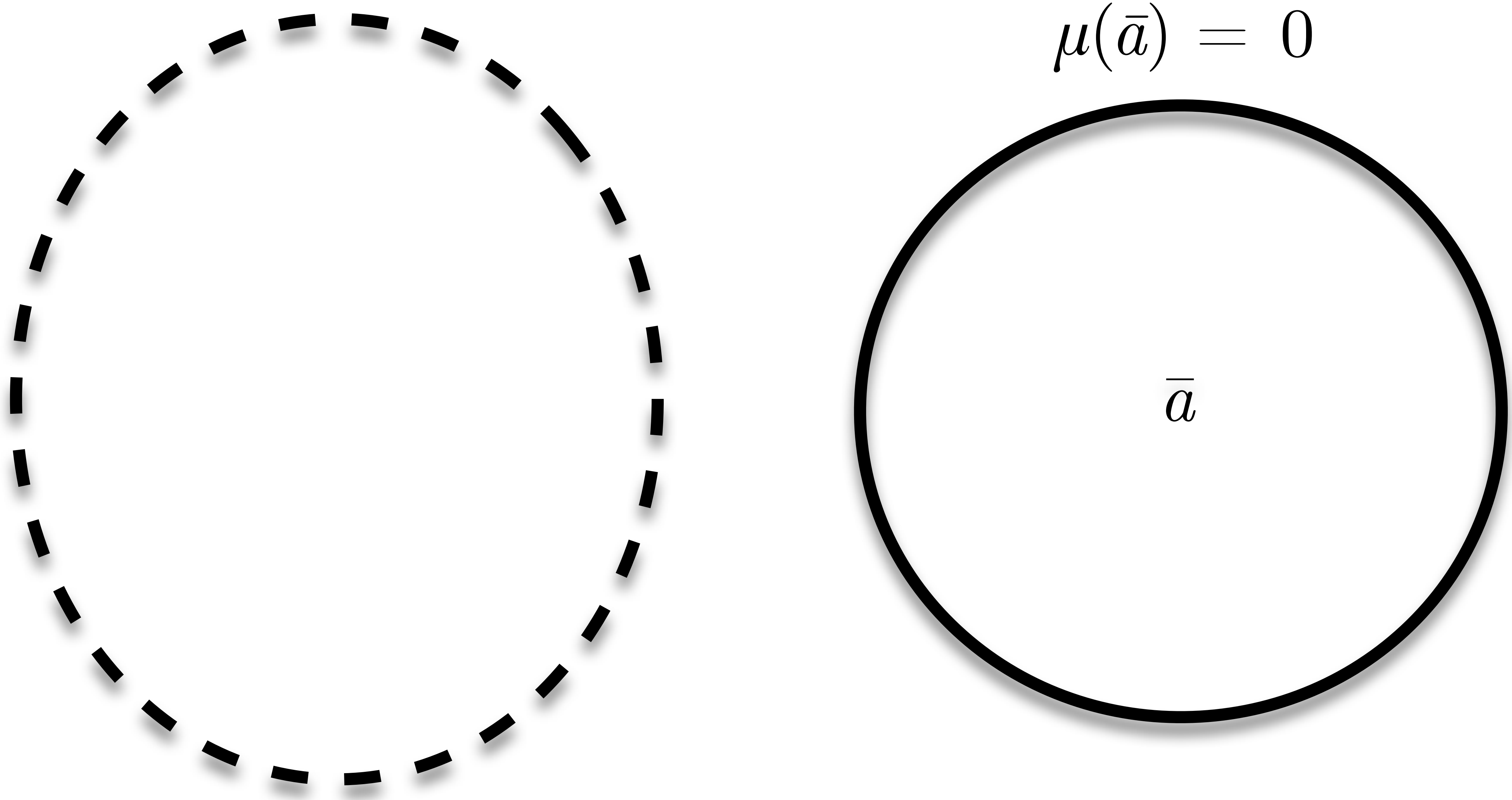
Counting principle

If $\{x\} = \{y\}$, then $x = y$.

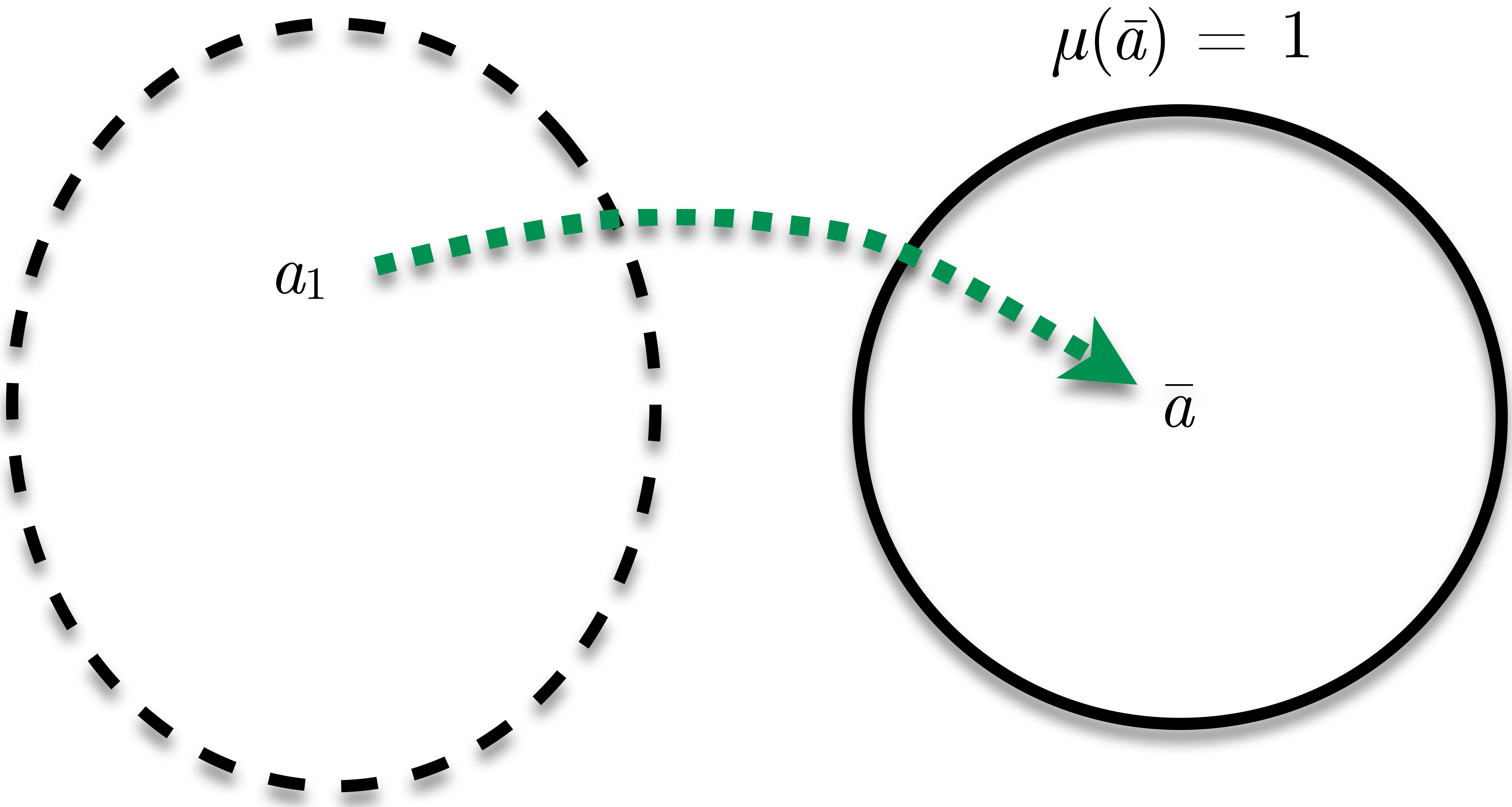
Counting

- Add measure to each state: μ
- $\mu : \overline{\text{Addr}} \rightarrow \{0,1,\infty\}$ counts counterparts

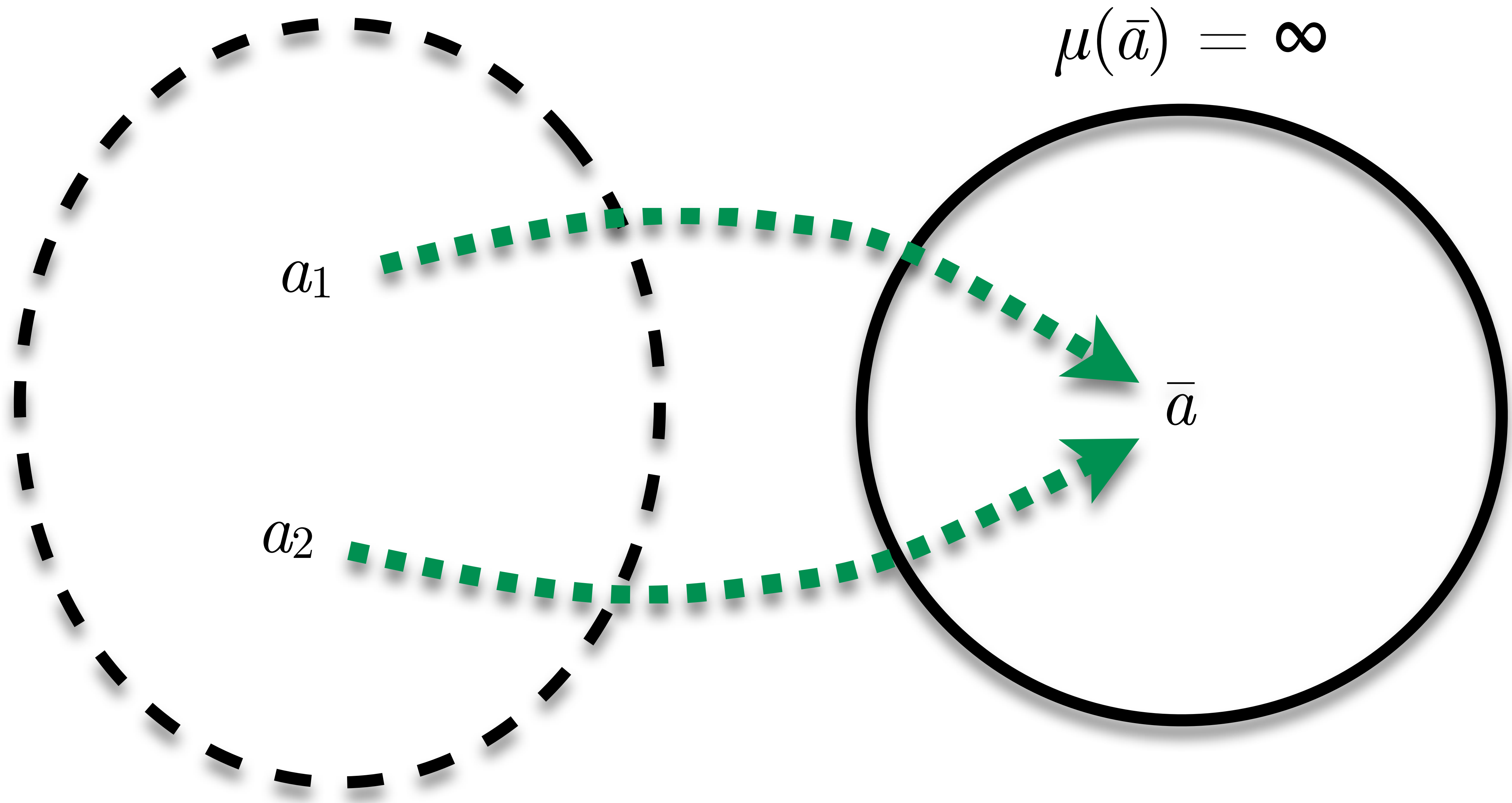
Counting



Counting



Counting



Abstract semantics

$$([\![(e_0 \ e_1 \ \dots \ e_n)]\!], \ \overline{env}, \ \overline{mem}) \Rightarrow (call, \ \overline{env}'', \overline{mem}')$$

$$\begin{aligned}
\overline{val}_i &= \overline{\mathrm{eval}}(e_i, \ \overline{env}, \ \overline{mem}) \\
\bar{a}_i &= \overline{\mathrm{alloc}}(\bar{s}, \ v_i) \\
\overline{val}_0 &\ni ([\![\lambda v_1 \ \dots \ v_n . call]\!], \overline{env}') \\
\overline{env}'' &= \overline{env}'[v_i \to \bar{a}_i] \\
\overline{mem}' &= \overline{mem} \ \sqcup [\bar{a}_i \to \overline{val}_i]
\end{aligned}$$

Abstract semantics

$$([\![(e_0 \ e_1 \ \ldots \ e_n)]\!], \ \overline{env}, \ \overline{mem}, \mu) \Rightarrow (call, \ \overline{env}'', \overline{mem}', \mu')$$

$$\begin{aligned}
\overline{val}_i &= \overline{\text{eval}}(e_i, \ \overline{env}, \ \overline{mem}) \\
\bar{a}_i &= \overline{\text{alloc}}(\bar{s}, \ v_i) \\
\overline{val}_0 &\ni ([\![\lambda v_1 \ \ldots \ v_n . call]\!], \overline{env}') \\
\overline{env}'' &= \overline{env}'[v_i \to \overline{a_i}] \\
\overline{mem}' &= \overline{mem} \ \sqcup [\overline{a}_i \to \overline{val}_i]
\end{aligned}$$

Abstract semantics

$$([\![(e_0 \ e_1 \ \ldots \ e_n)]\!], \ \overline{env}, \ \overline{mem}, \mu) \Rightarrow (call, \ \overline{env}'', \overline{mem}', \mu')$$

$$\begin{aligned}
\overline{val}_i &= \overline{\mathrm{eval}}(e_i, \ \overline{env}, \ \overline{mem}) \\
\bar{a}_i &= \overline{\mathrm{alloc}}(\bar{s}, \ v_i) \\
\overline{val}_0 &\ni ([\![\lambda v_1 \ \ldots \ v_n.call]\!], \overline{env}') \\
\overline{env}'' &= \overline{env}'[v_i \rightarrow \ \overline{a_i}] \\
\overline{mem}' &= \overline{mem} \ \sqcup [\overline{a}_i \rightarrow \ \overline{val}_i] \\
\mu' &= \mu \oplus [\overline{a}_i \rightarrow 1]
\end{aligned}$$

Example: Inlining

```
λv k.(k a)
```

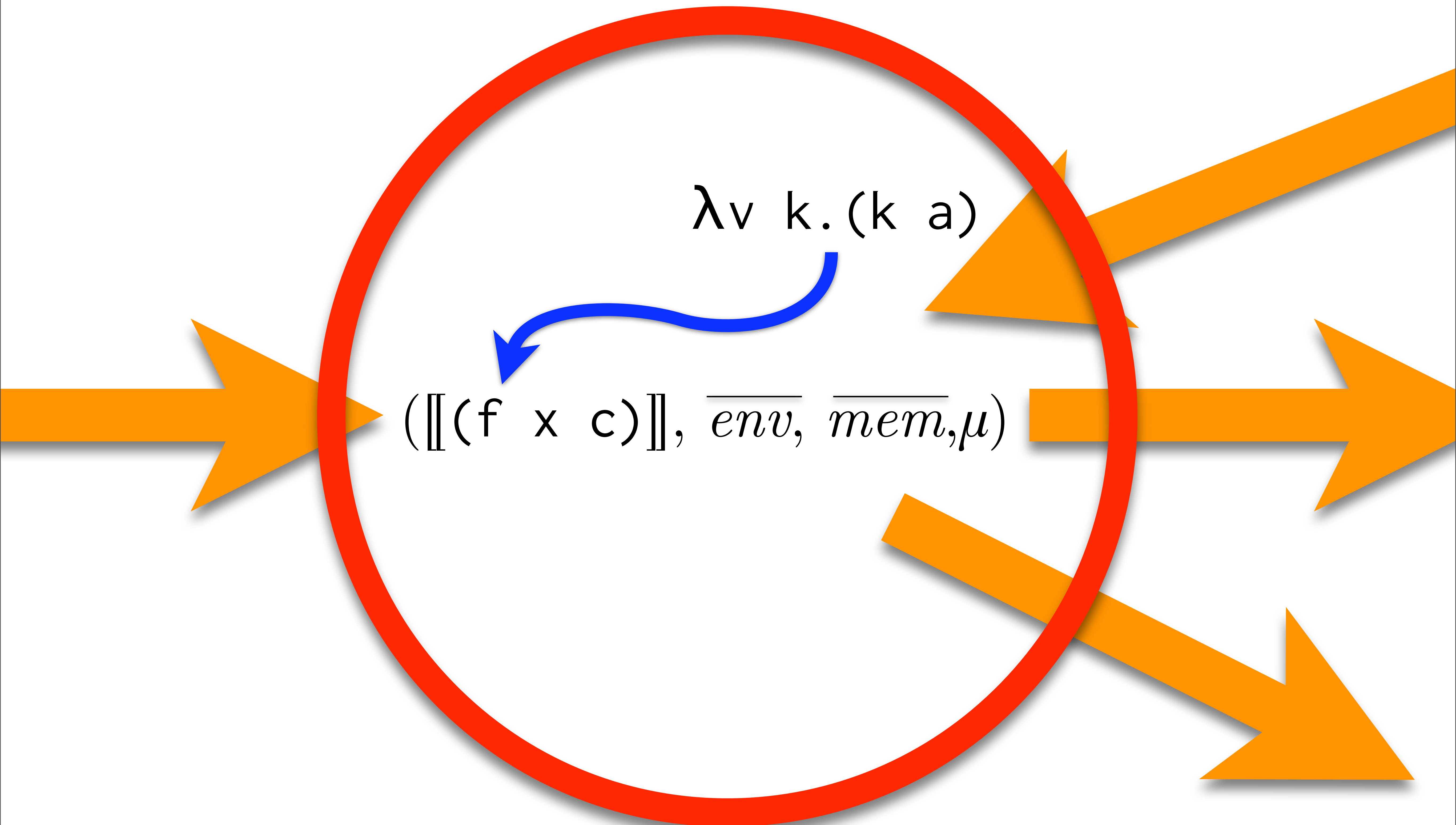
```
(f x c)
```

Example: Inlining

```
λv k.(k a)

(f x c)
```

Example: Inlining



Example: Inlining

$$\overline{mem}(\overline{env}[\![\mathsf{f}]\!]) = \{([\![\,\mathtt{\lambda v\ k.(k\ a)}]\!],\ \overline{env}')\}$$

$$([\![\mathtt{(f\ x\ c)}]\!],\ \overline{env},\ \overline{mem},\mu)$$

Example: Inlining

$$\overline{mem}(\overline{env}[\![\mathsf{f}]\!]) = \{([\![\,\mathtt{\lambda v\ k.(k\ a)}]\!],\ \overline{env}')\}$$

$$([\![\mathtt{(f\ x\ c)}]\!],\ \overline{env},\ \overline{mem}, \mu)$$

1. $\overline{a} = \overline{env}[\![\mathsf{a}]\!] = \overline{env}'[\![\mathsf{a}]\!]$

Example: Inlining

$$\overline{mem}(\overline{env}[\![\mathsf{f}]\!]) = \{([\![\lambda\mathsf{v}\ \mathsf{k}.(\mathsf{k}\ \mathsf{a})]\!], \overline{env}')\}$$

$$([\![\mathsf{(f\ x\ c)}]\!], \overline{env}, \overline{mem}, \mu)$$

1. $\overline{a} = \overline{env}[\![\mathsf{a}]\!] = \overline{env}'[\![\mathsf{a}]\!]$
2. $\mu(\overline{a}) = 1$

Summary: Counting

Summary: Counting

- Simple

Summary: Counting

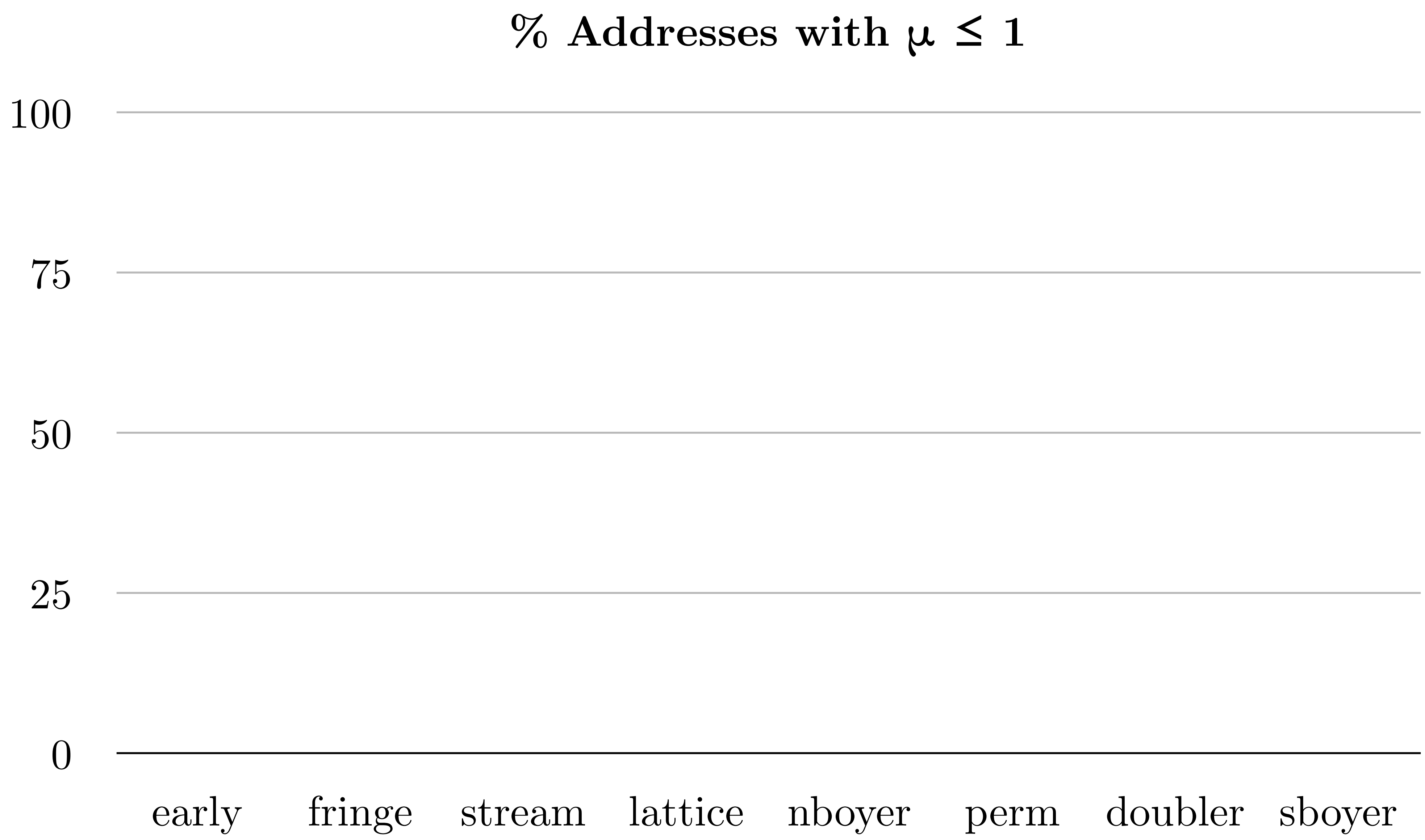
- Simple
- Correct

Summary: Counting

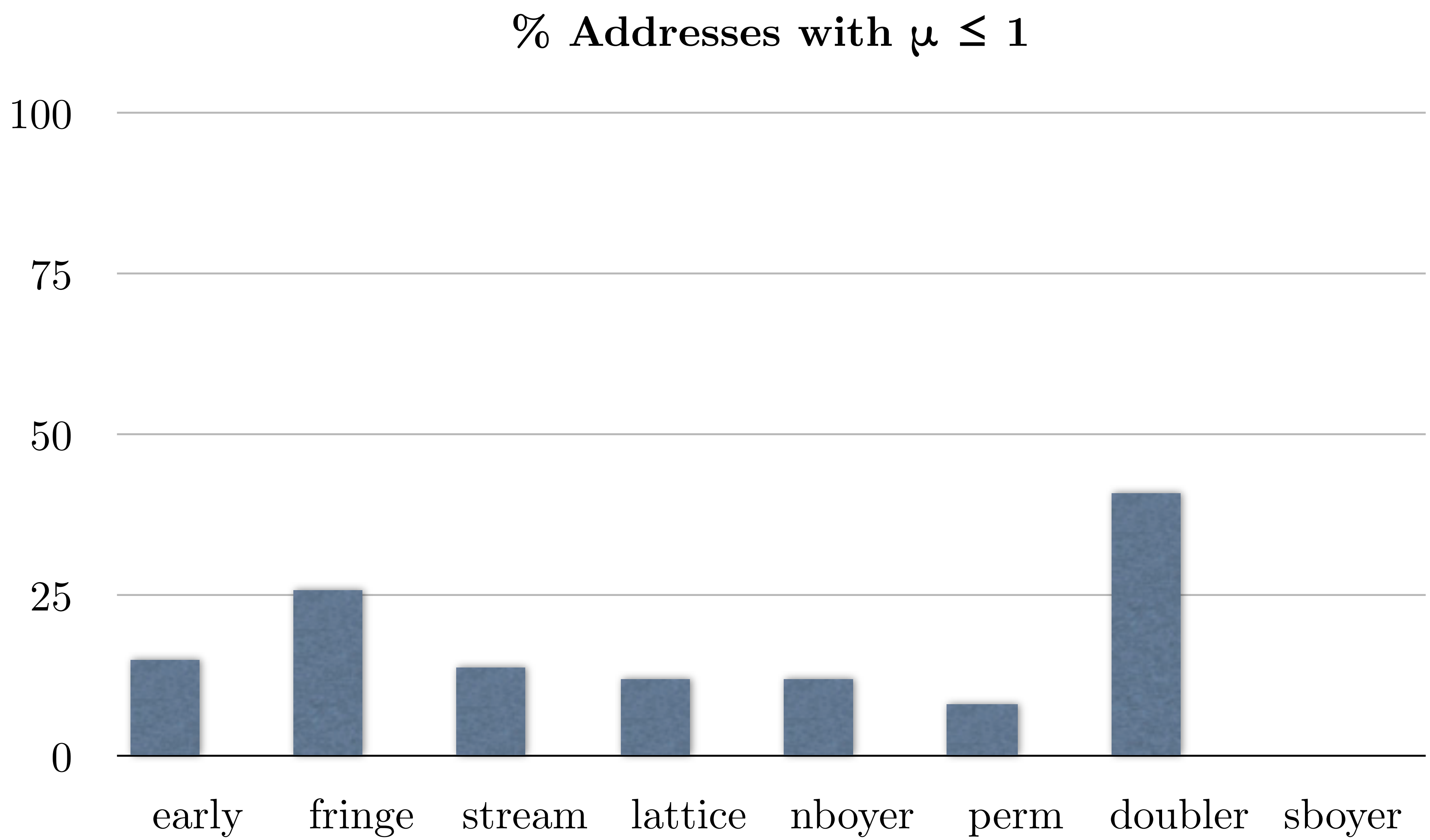
- Simple
- Correct
- Worthless

Results: Counting

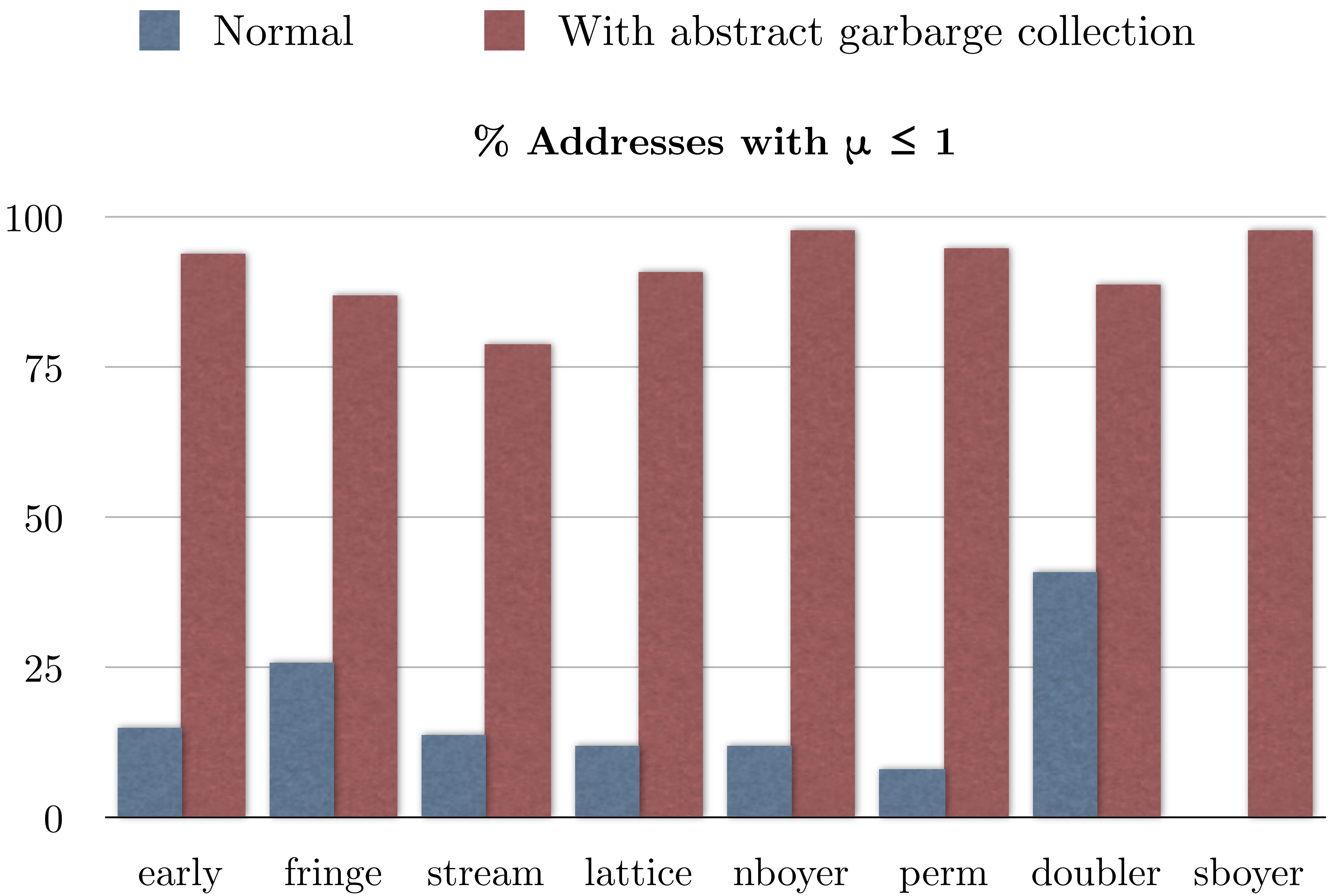
Results: Counting



Results: Counting



Results: Counting



Abstract garbage collection

Before an abstract transition,
discard unreachable structure
from the heap.

Example: Zombies & GC

mem $\overline{mem}$

Example: Zombies & GC

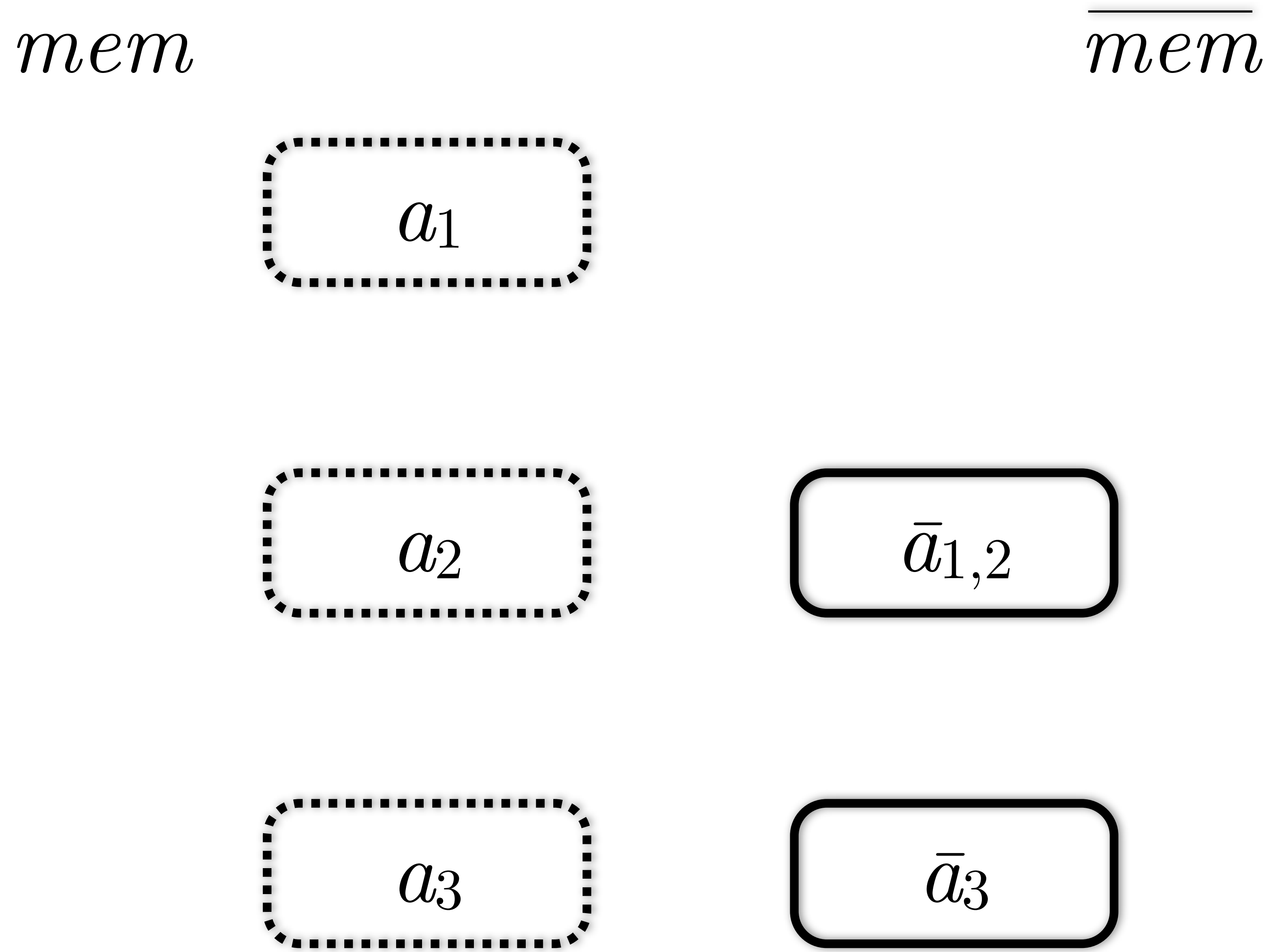
mem $\overline{mem}$

$\boxed{a_1}$

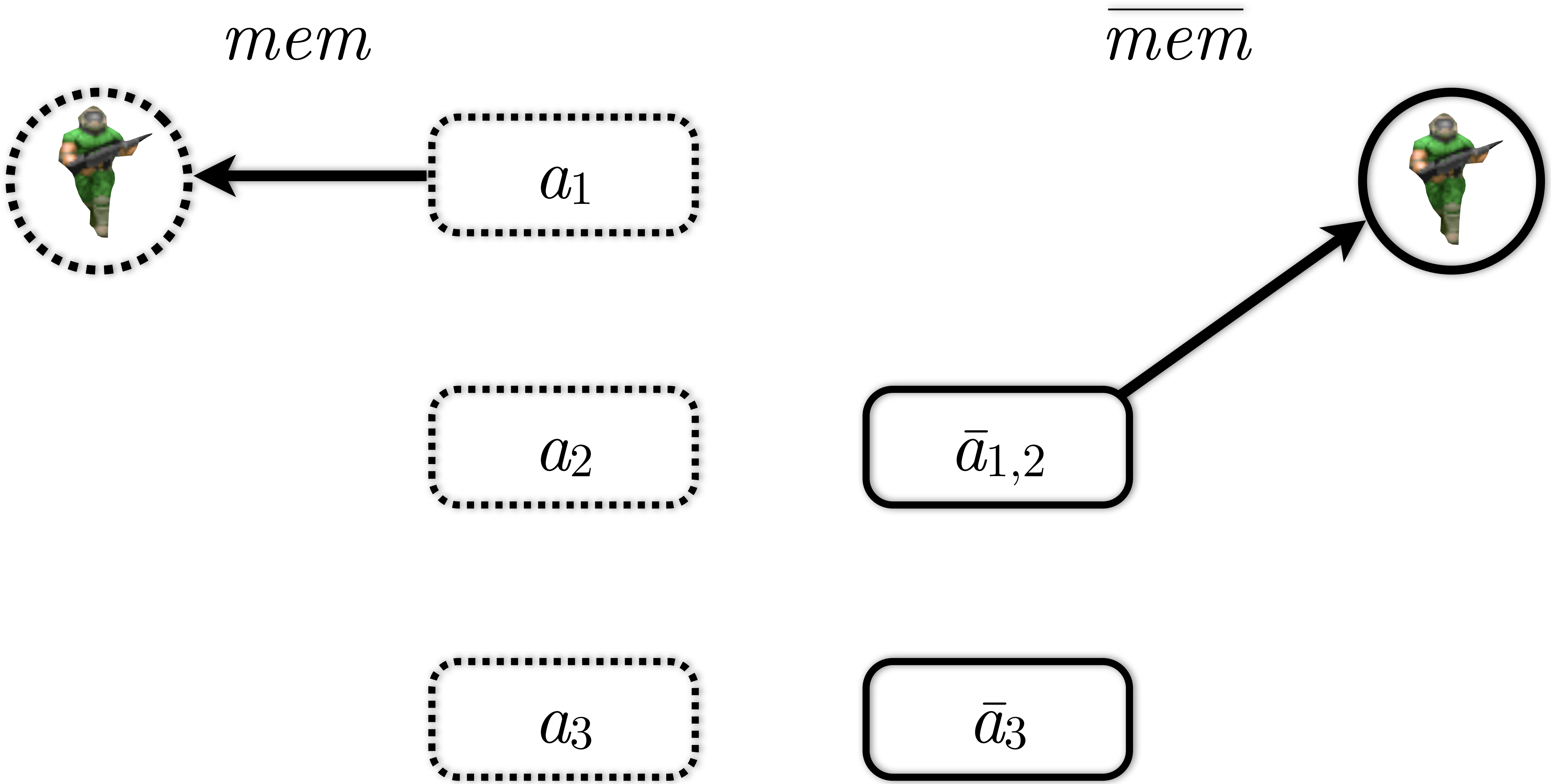
$\boxed{a_2}$

$\boxed{a_3}$

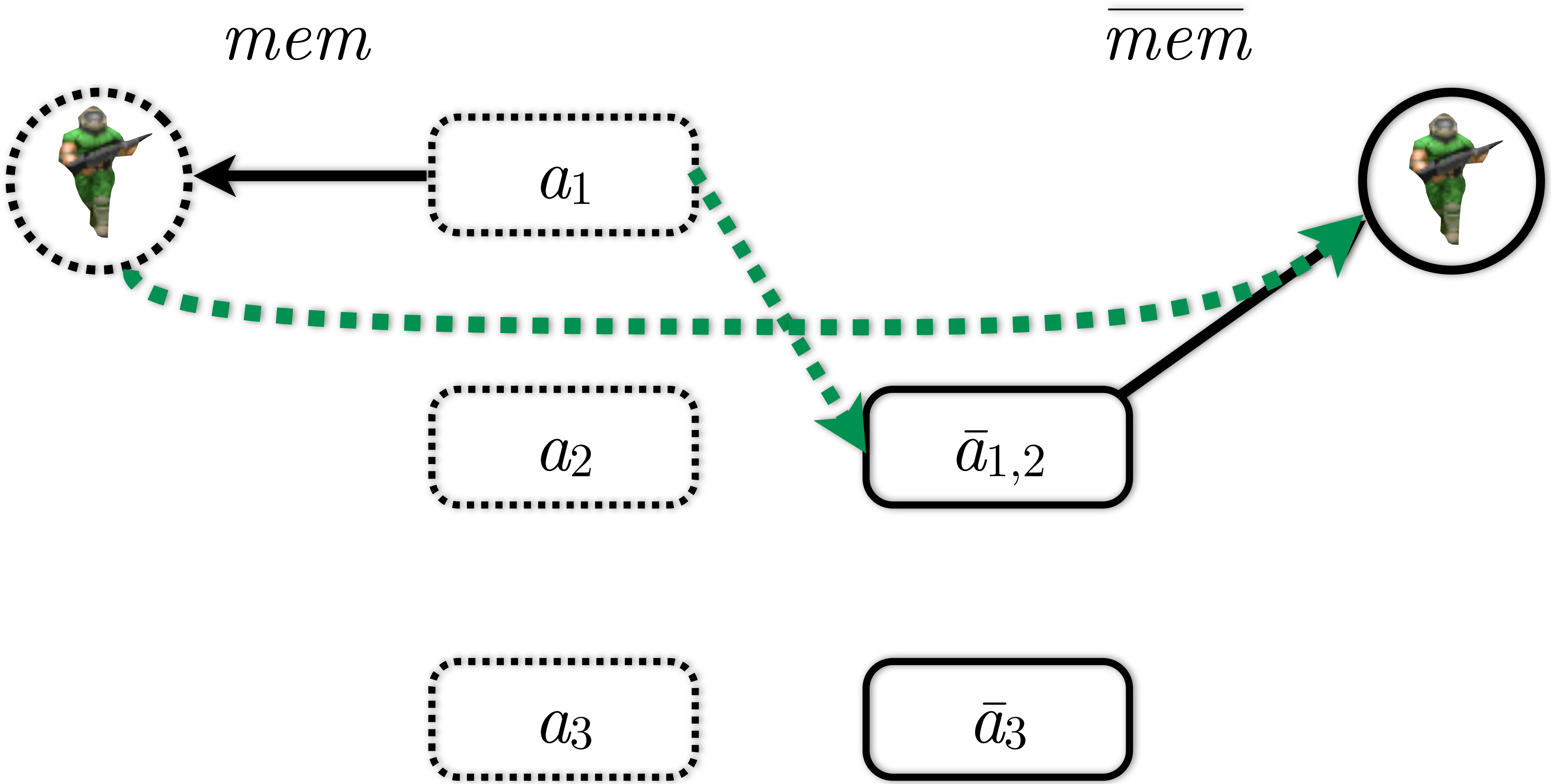
Example: Zombies & GC



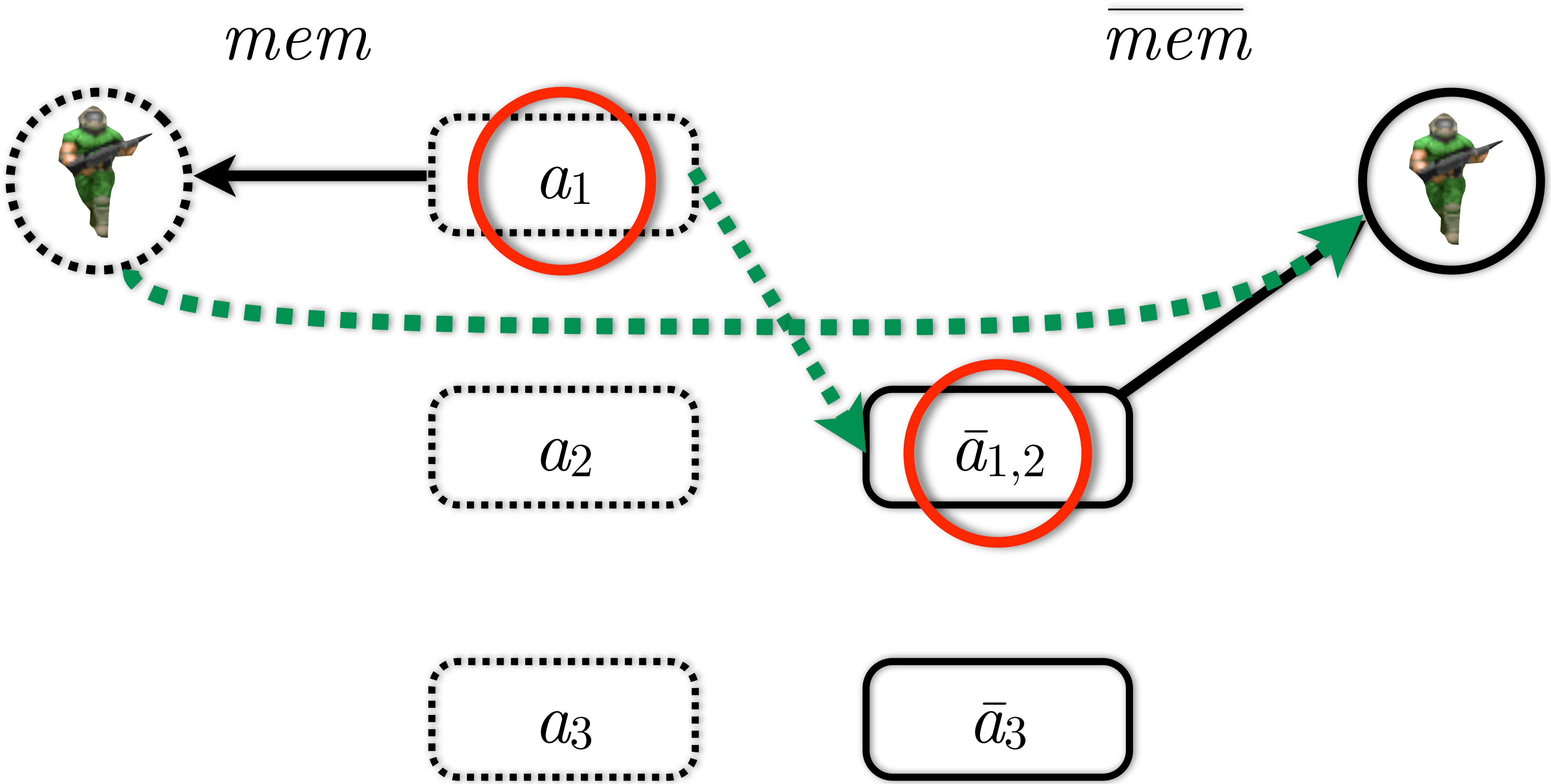
Example: Zombies & GC



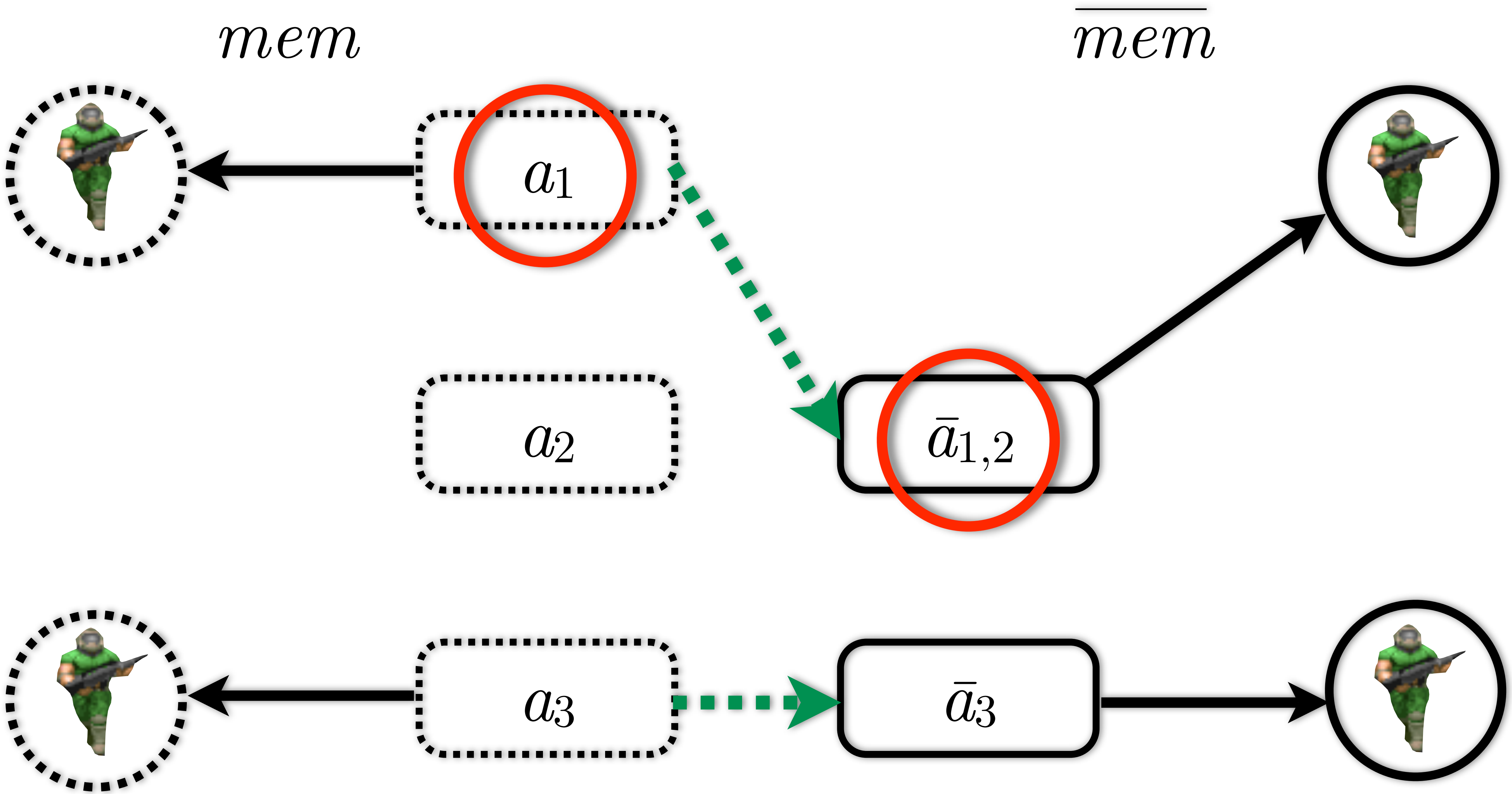
Example: Zombies & GC



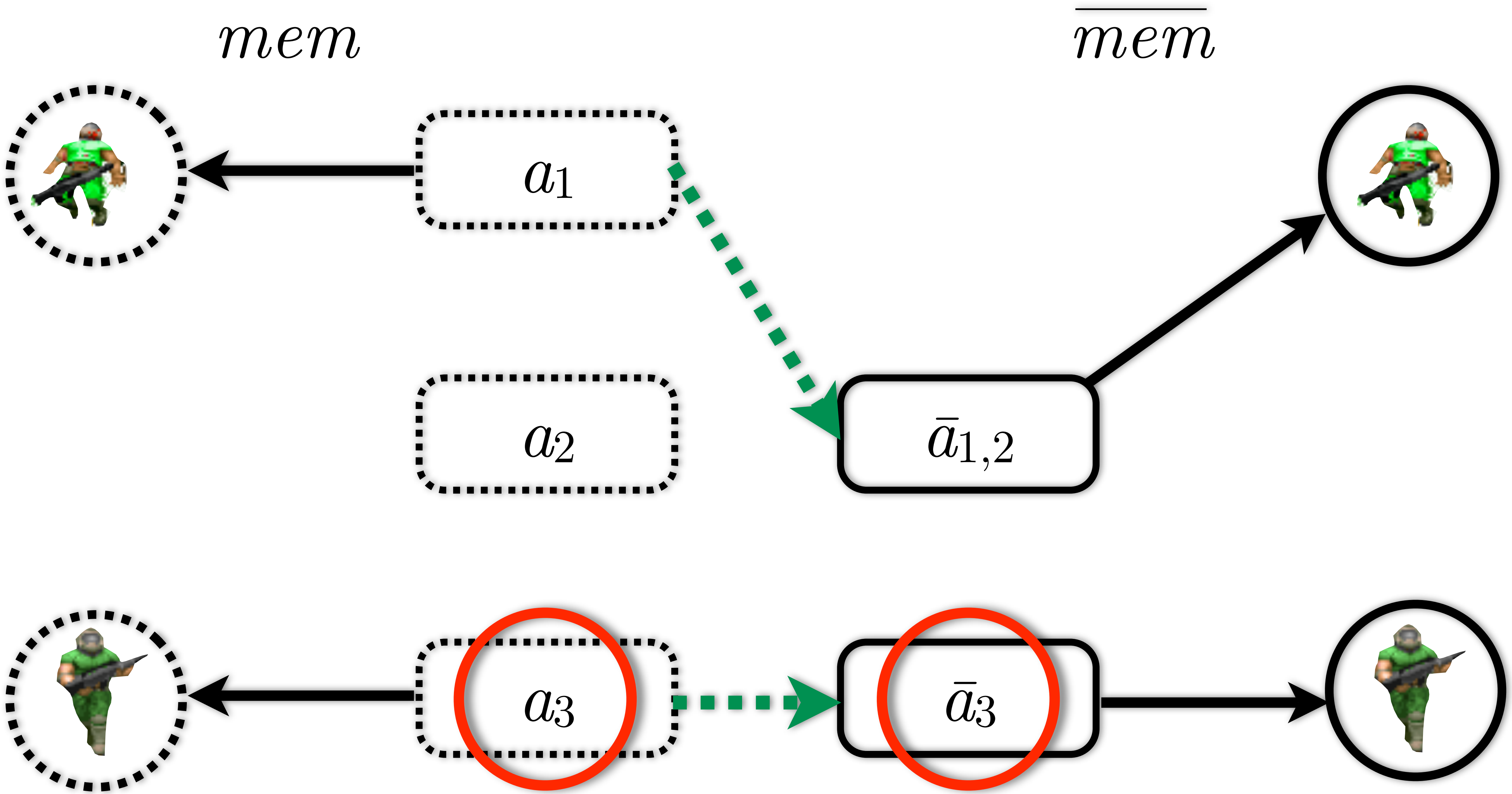
Example: Zombies & GC



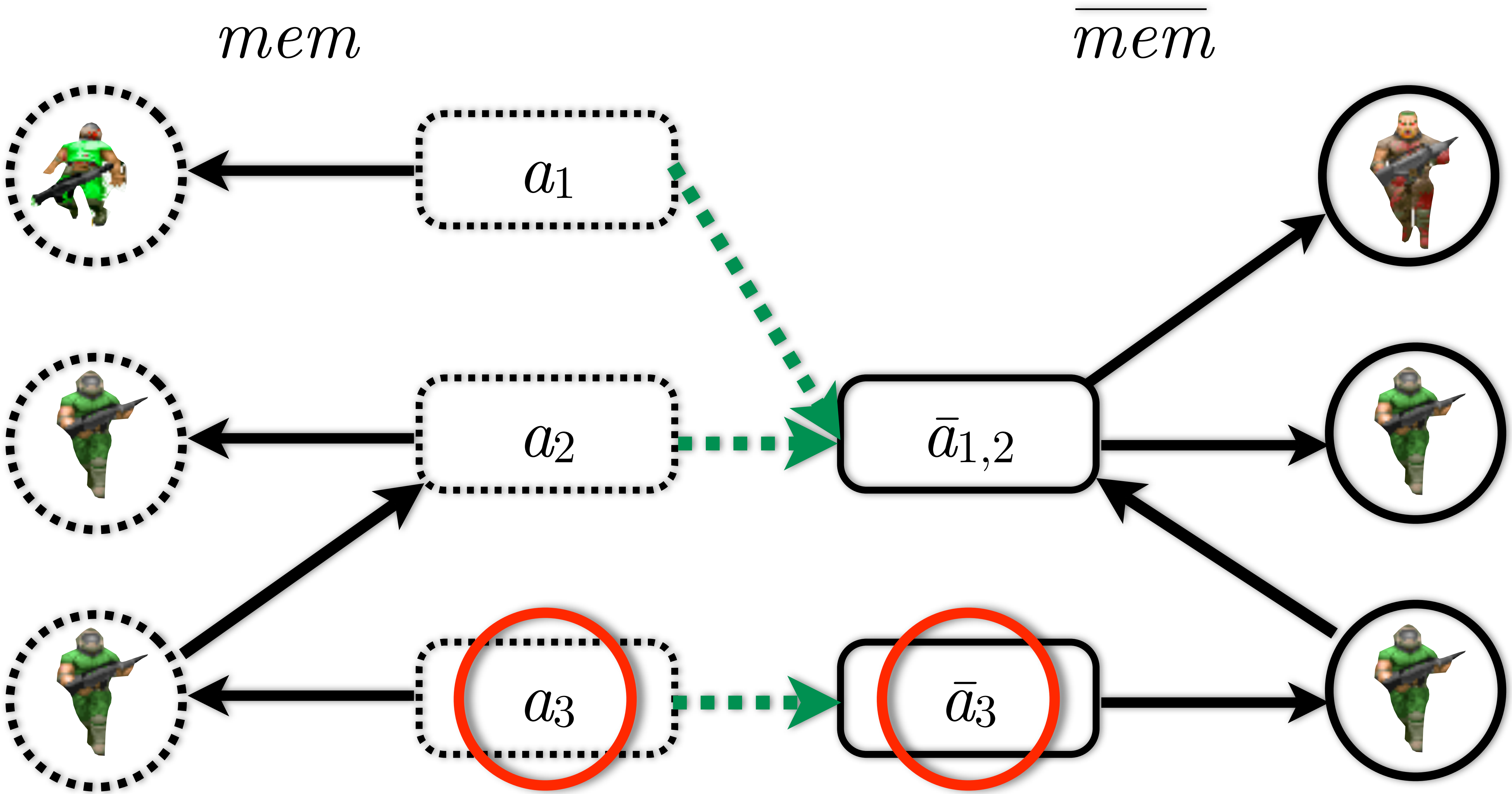
Example: Zombies & GC



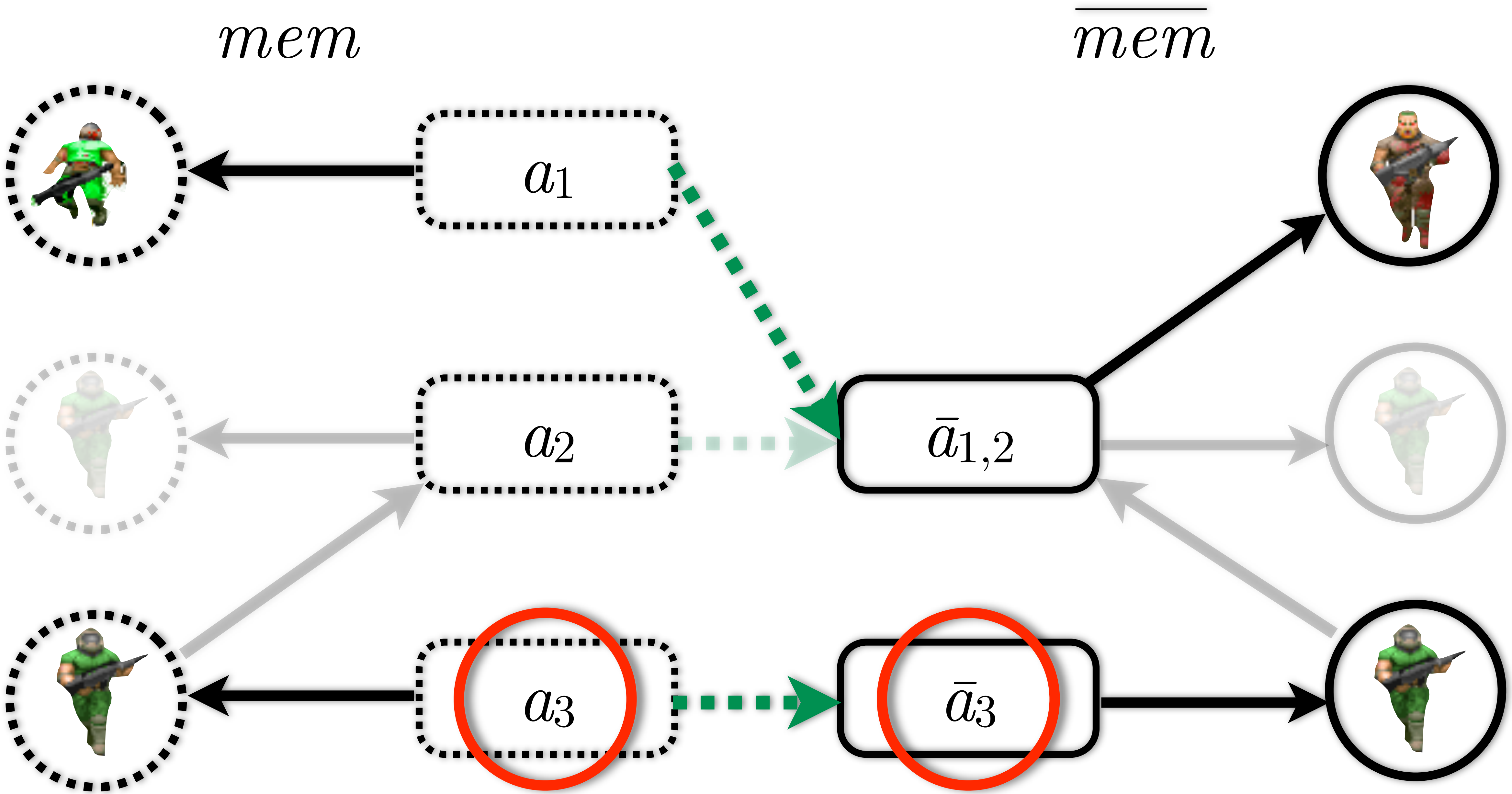
Example: Zombies & GC



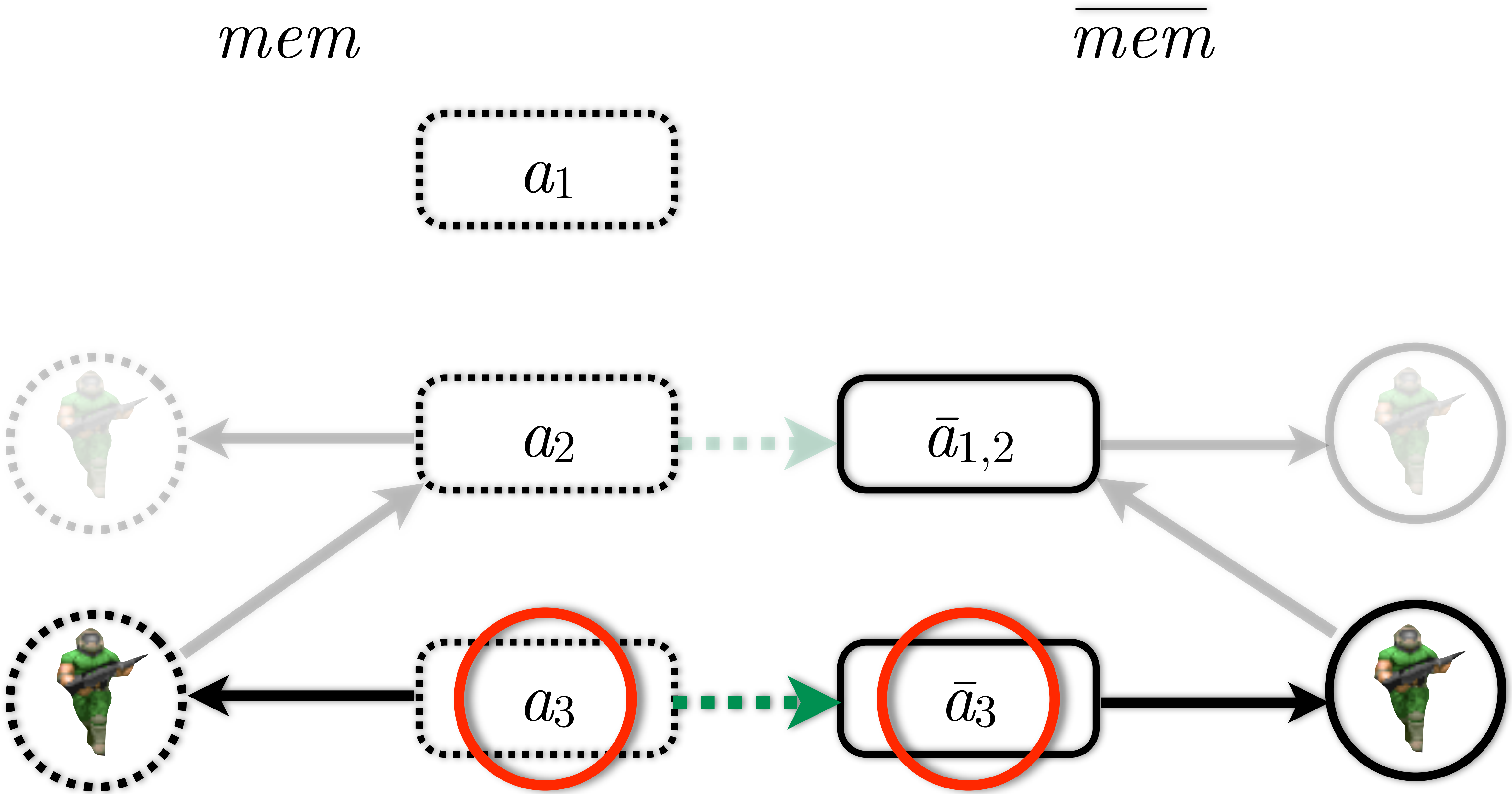
Example: Zombies & GC



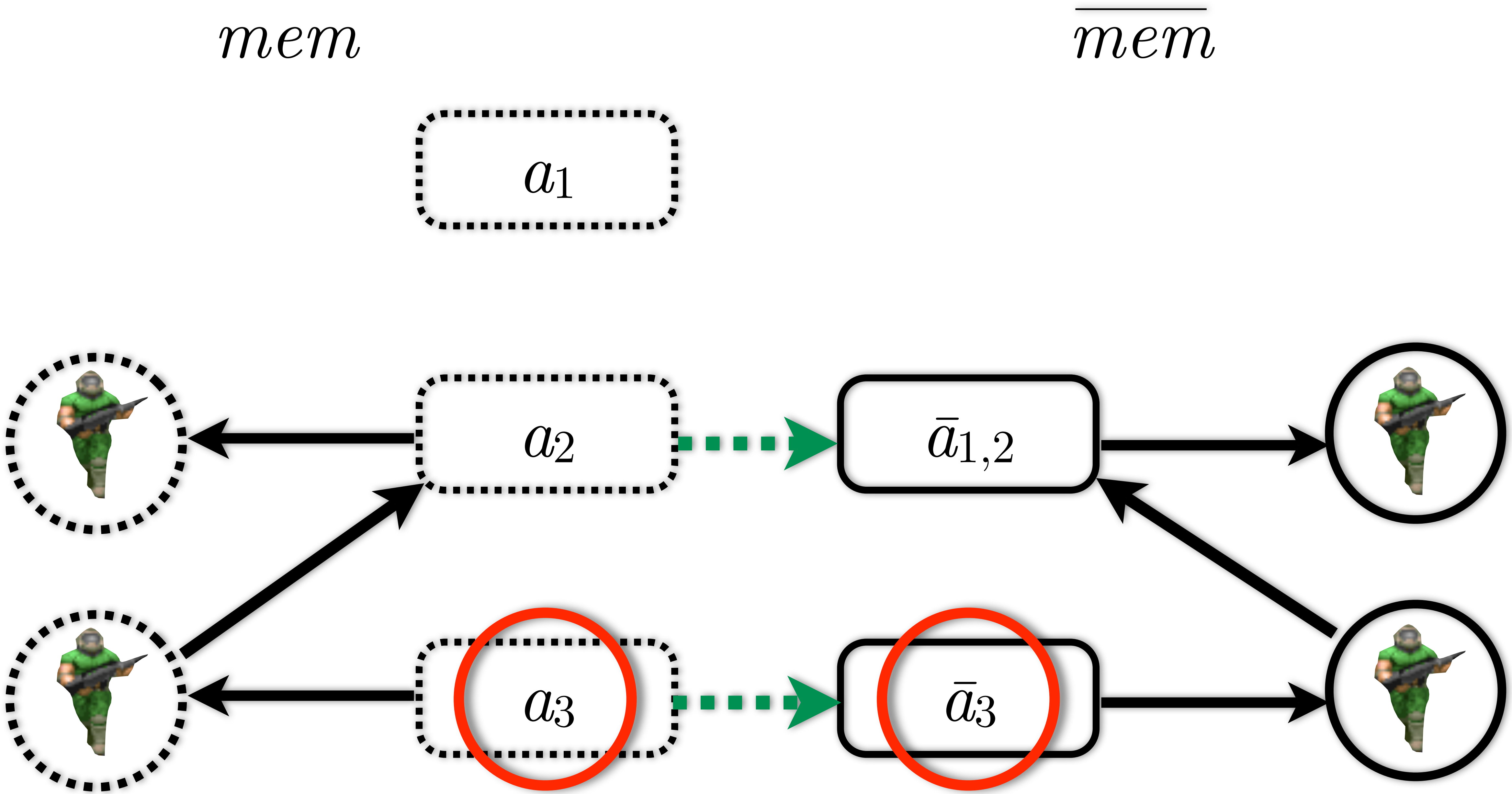
Example: Zombies & GC



Example: Zombies & GC

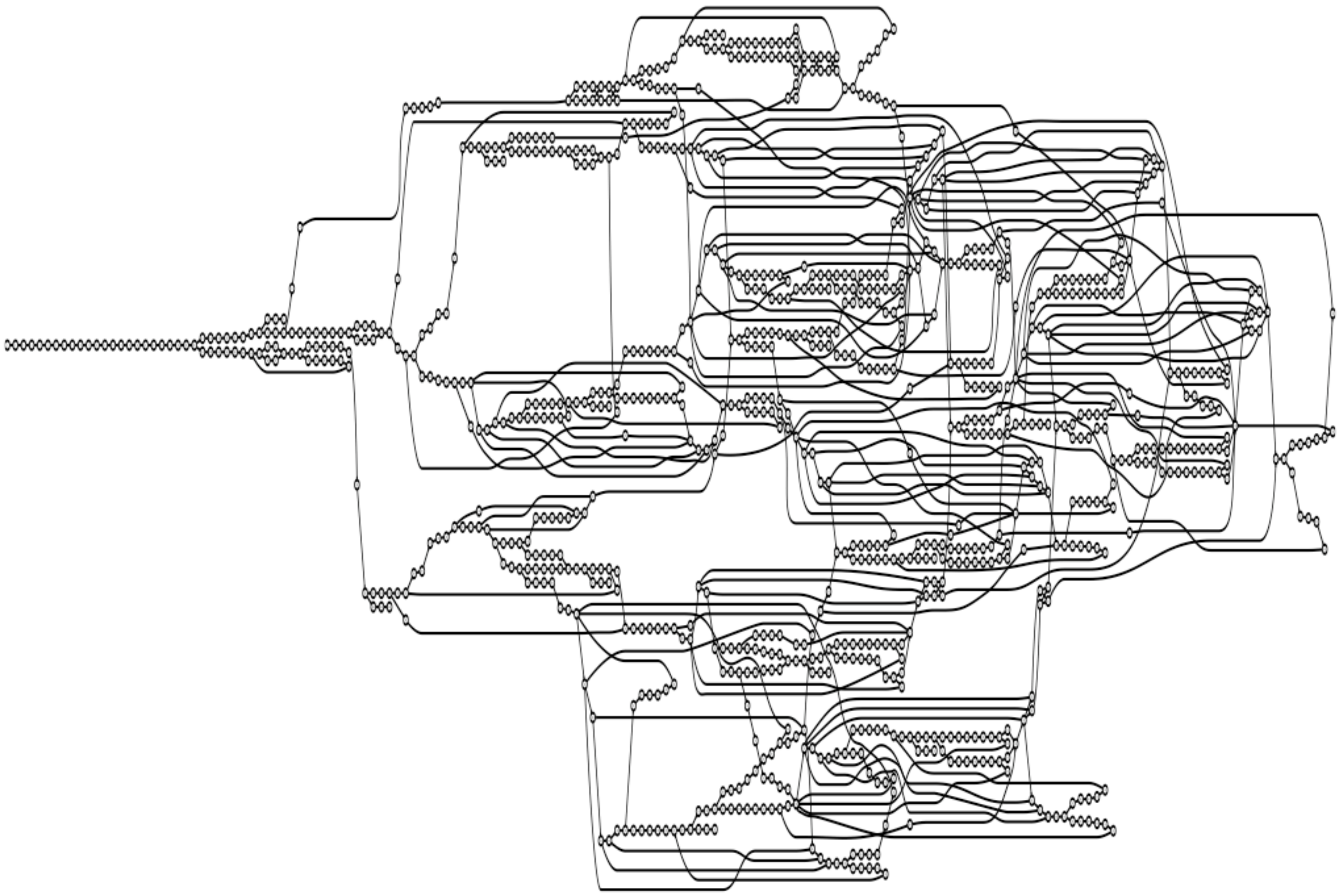


Example: Zombies & GC

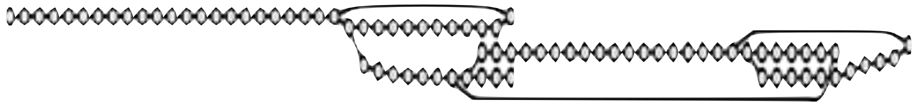


Effect of Abstract GC

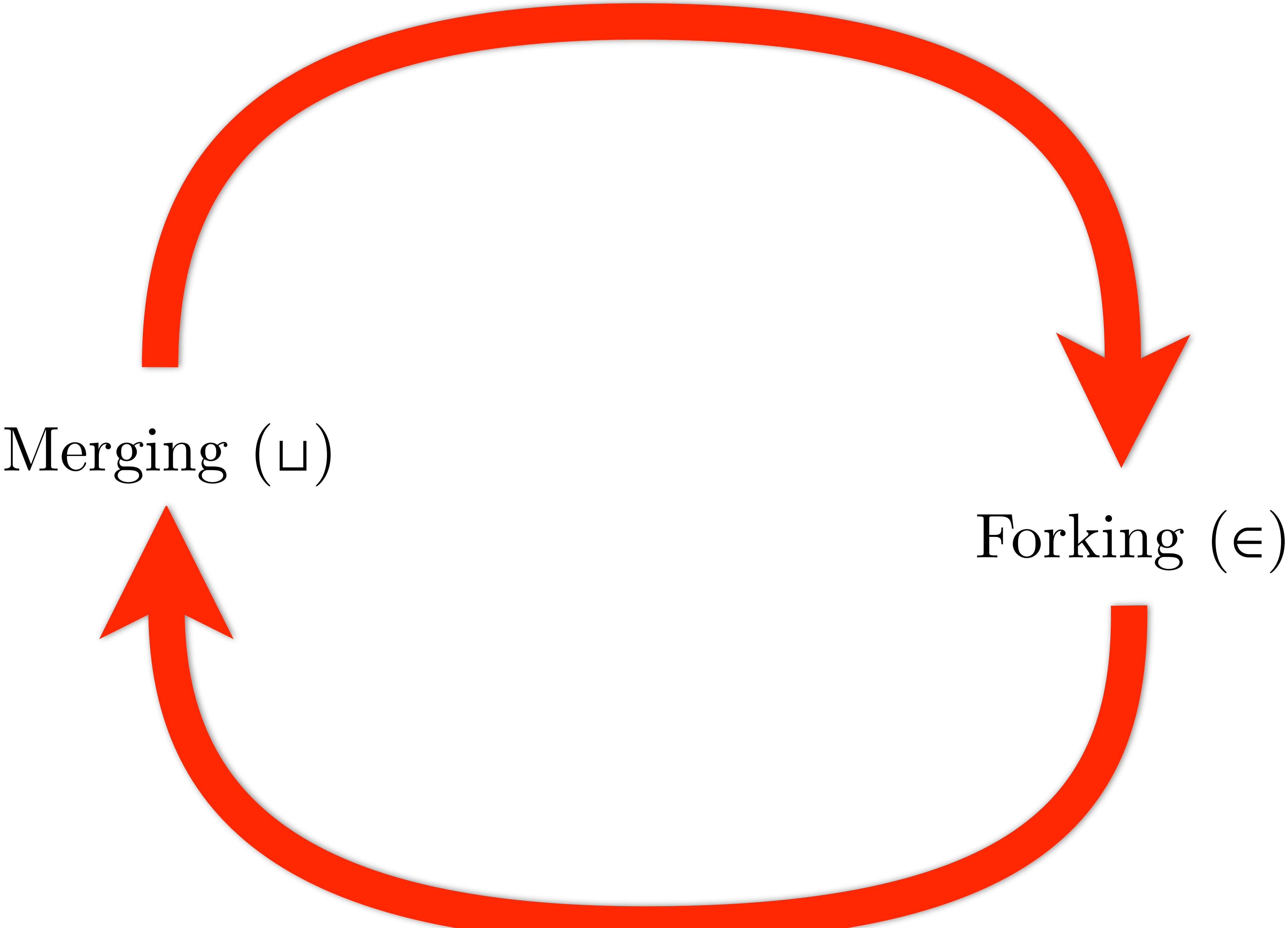
Effect of Abstract GC



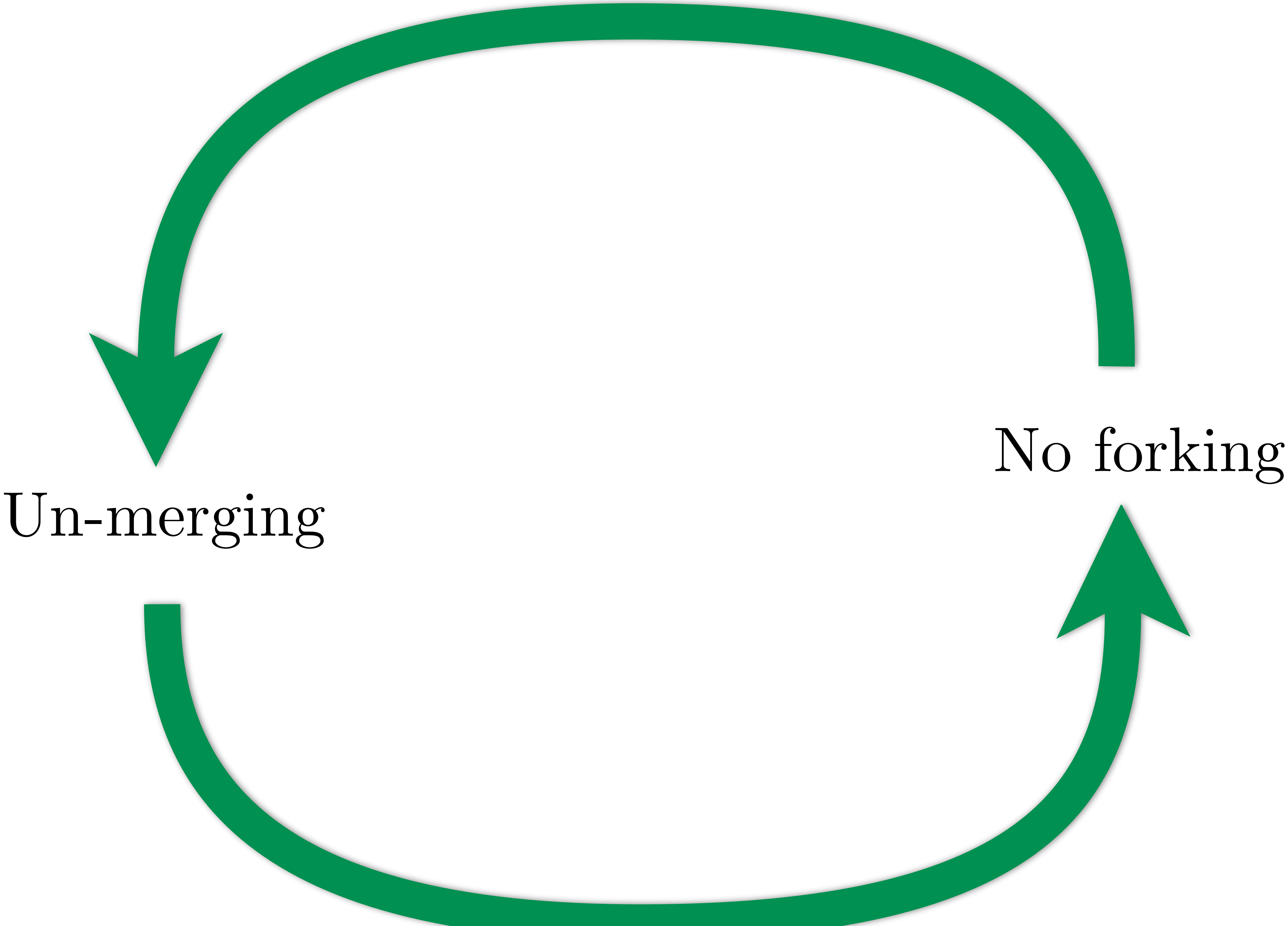
Effect of Abstract GC



Vicious cycle



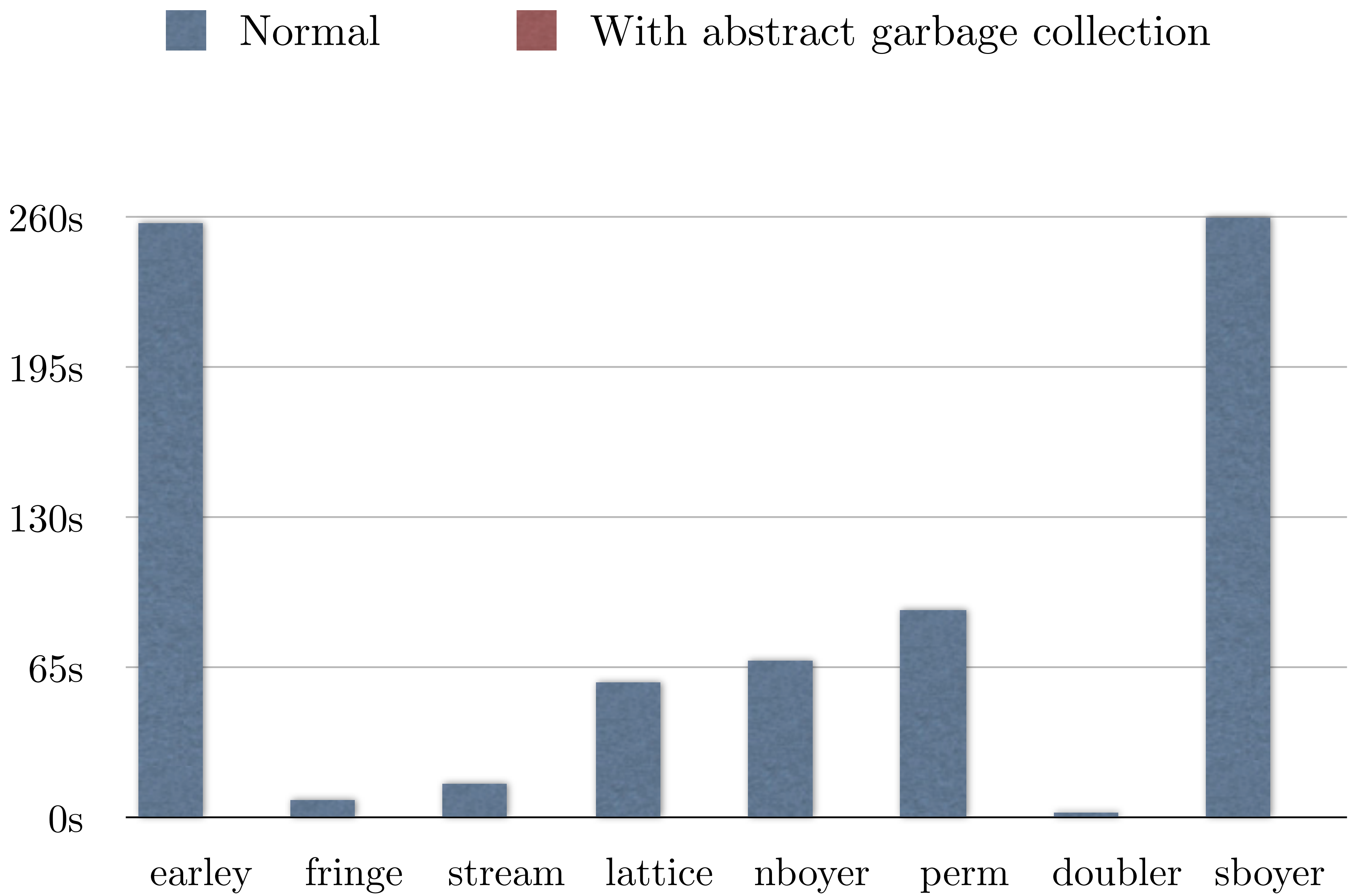
Virtuous cycle



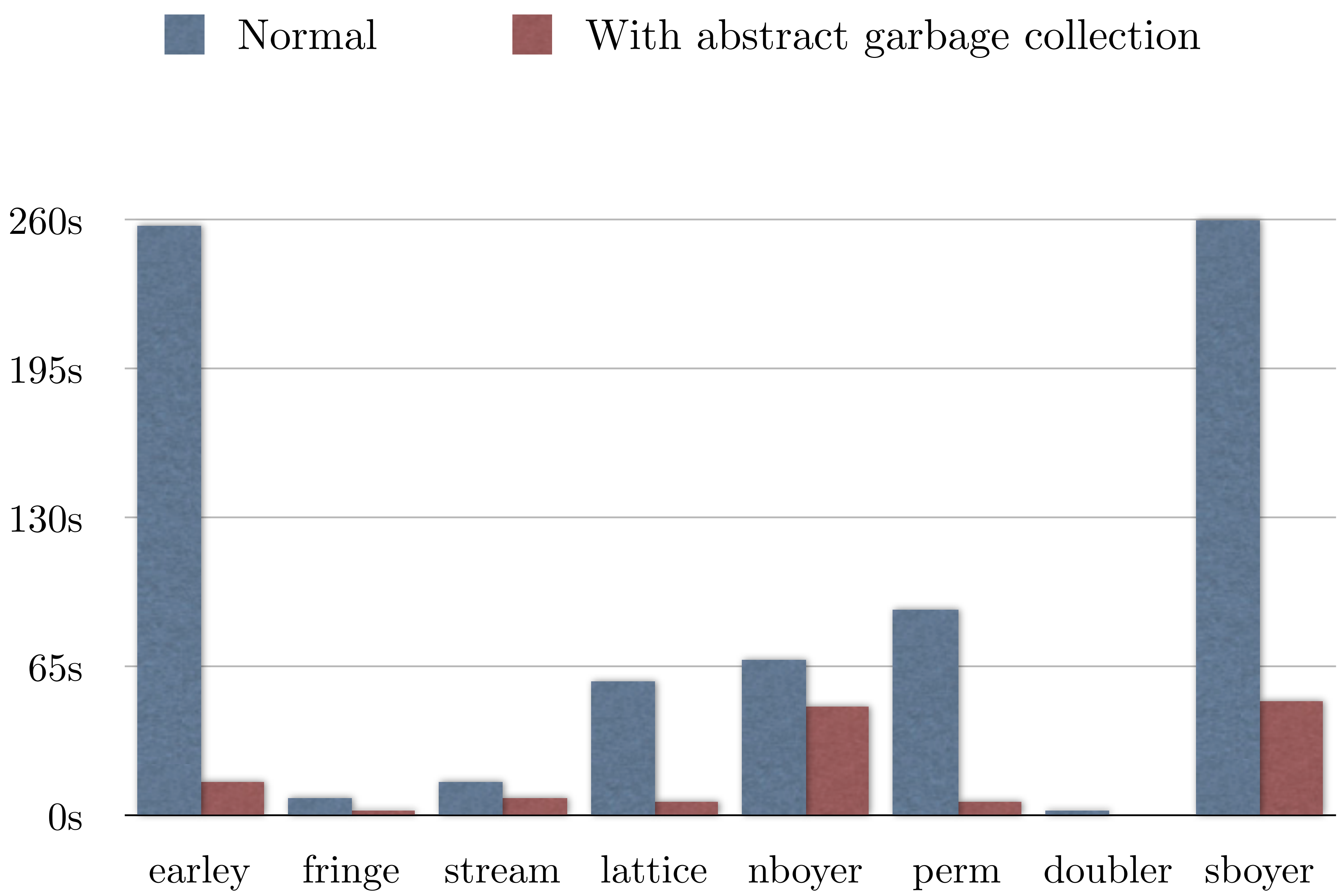
Results: Analysis time

Normal   With abstract garbage collection

Results: Analysis time



Results: Analysis time



Environment analysis is not enough.

Example: Overflow

```
a[i]
```

Example: Overflow

`a[i]`

Can we prove that `i` is in bounds?

Example: Overflow

```
let loop i =
 if i < length a
 then f(a[i]) ;
      loop (i+1)
 else ()
in loop 0
```

Can we prove that `i` is in bounds?

Example: Overflow

```
indices.each(λi.
  f(a[i]))
```

Can we prove that `i` is in bounds?

Example: Overflow

```
λ(a,i).
  f(a[i]))
```

Can we prove that `i` is in bounds?

Example: Overflow

Harder than "what flows here?"

```
λ(a,i)
  f(a[i]))
```

Can we prove that `i` is in bounds?

Logic-flow analysis (POPL 2007)

Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)

s

$\hat{s}$

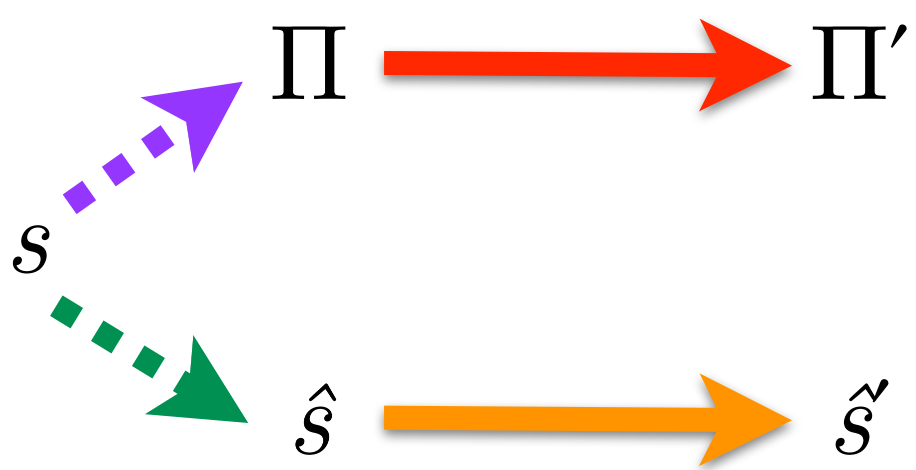
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions (Π)

$$\begin{tikzcd} & \Pi \\ s \arrow[ur, dotted] \arrow[dr, dotted] & \\ & \hat{s} \end{tikzcd}$$

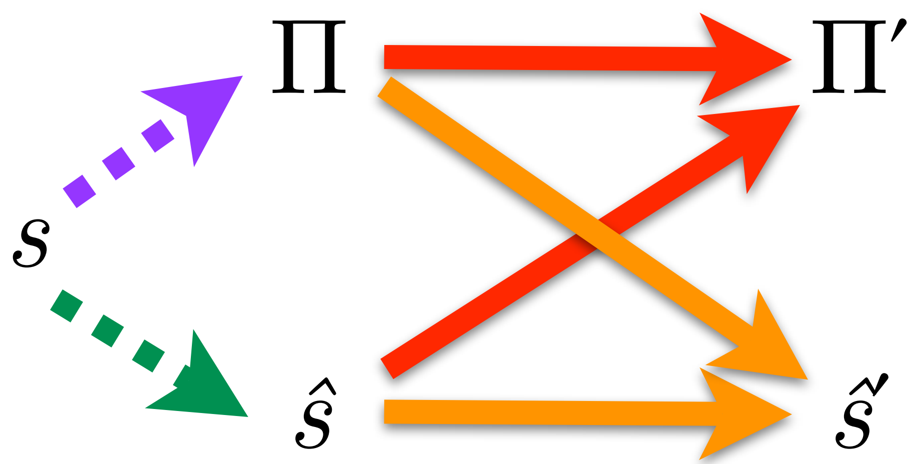
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions (Π)
- Conduct joint abstract interpretation



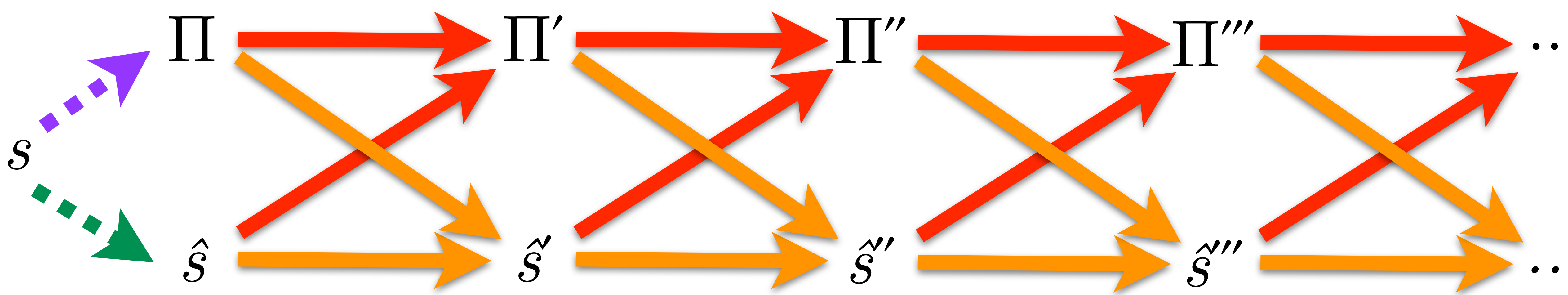
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions (Π)
- Conduct joint abstract interpretation



Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions (Π)
- Conduct joint abstract interpretation



The Palsberg paradigm

"Higher-order analysis =
first-order analysis + control-flow analysis."

Jens Palsberg

The post-Palsberg paradigm

"Higher-order analysis $=$
first-order analysis $\times$ control-flow analysis."

My research in the wild

- Optimization: Scheme 48 (Knauel)
- Optimization: Waterloo (Zwarich)
- Coroutines: MLton (Chambers, Harvey)
- Security: LLVM, C/C++ (Diagis)

Thank you

Ongoing work

- Automatic parallelization
- Polyvariance completeness theorem
- Anodizing semantics
- Abstractions of Peano pointer arithmetic
- Soundness modulo congruence