Static analysis of modern software systems: Taming control-flow

Matt Might

University of Utah

matt.might.net

ucombinator.org

Problem

Software fails.

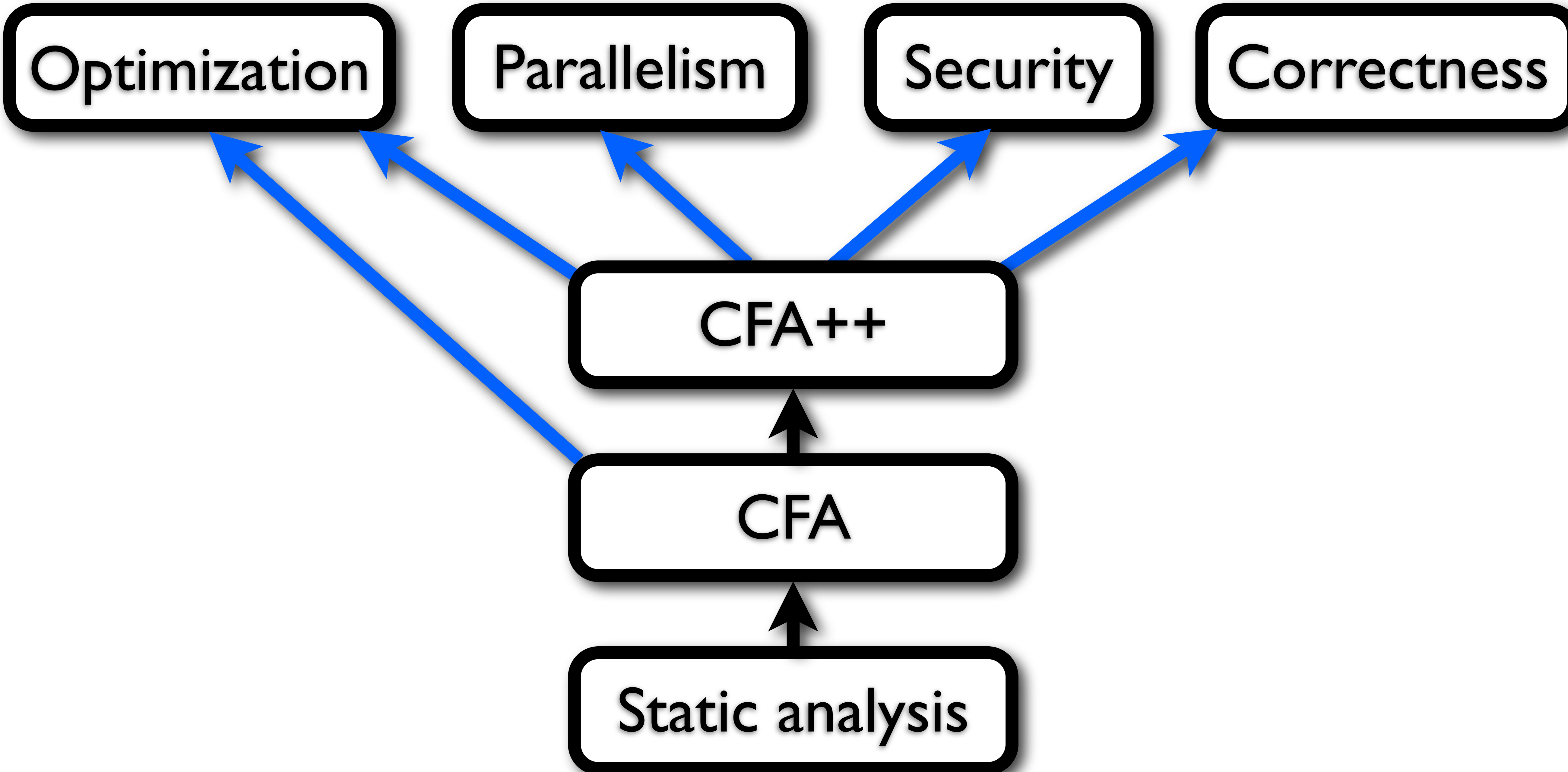
Problem

- Software fails because we can’t engineer it.
- We can’t engineer what we can’t predict.
- We can’t predict the behavior of software.

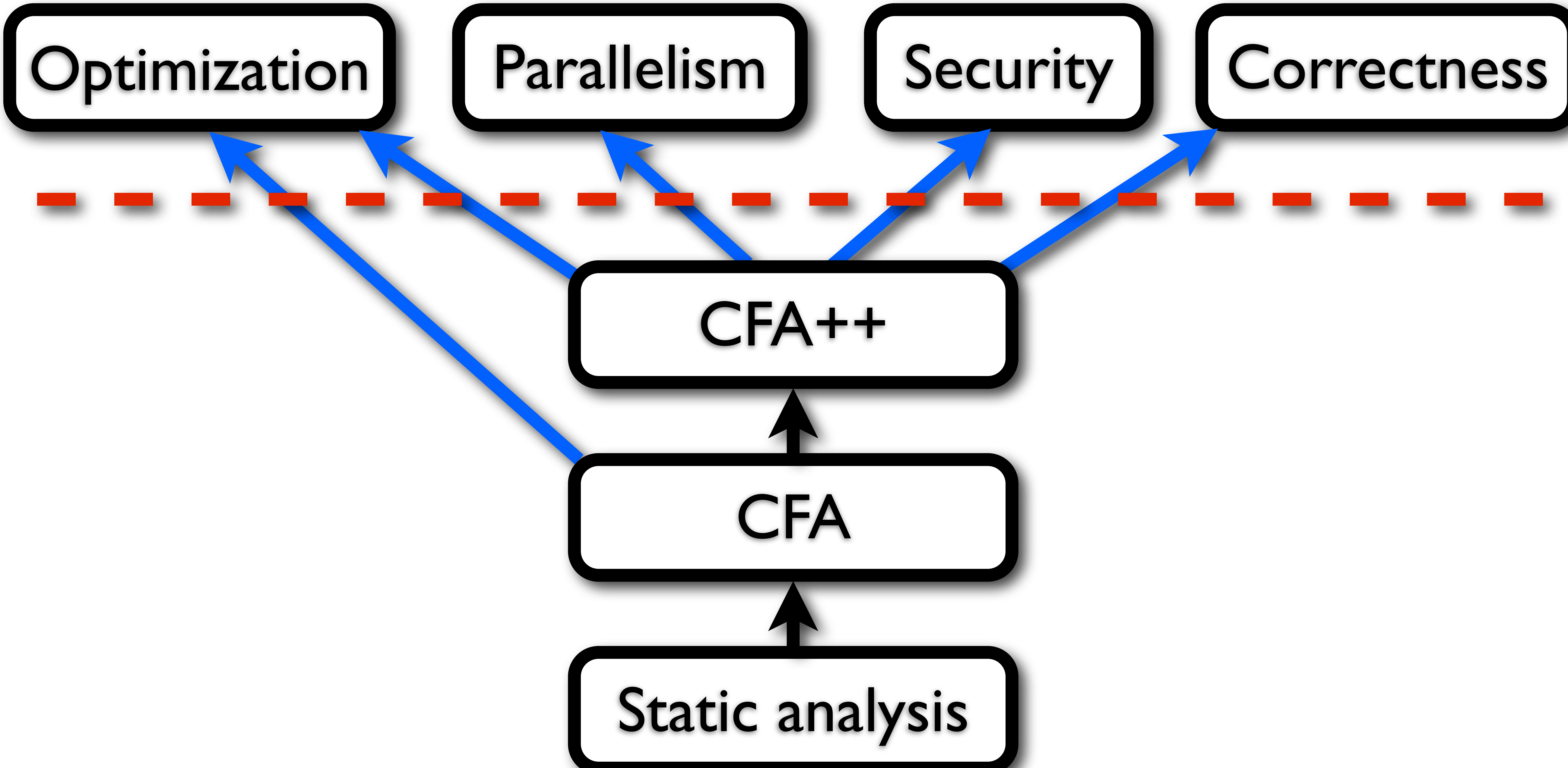
Message

- Static analysis of modern software is hard!
- Control-flow analysis is the gatekeeper.
- Yet, precise control-flow analysis is possible.

Goal

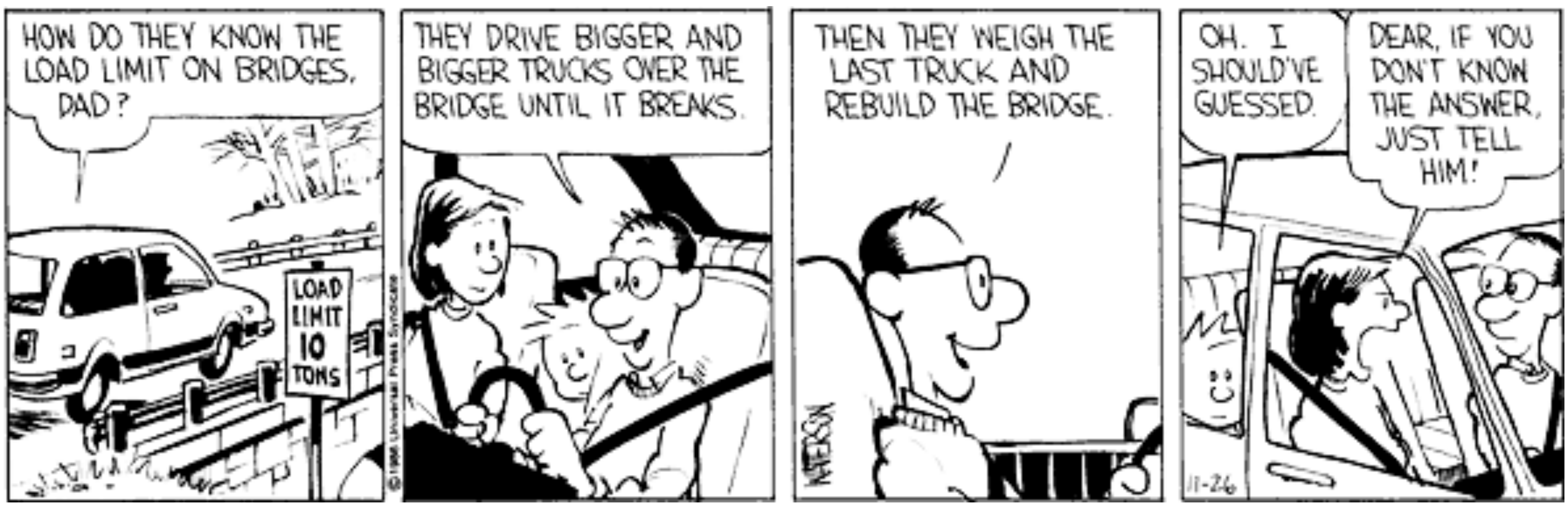

Optimization
Parallelism
Security
Correctness
CFA++
CFA
Static analysis

Goal



Optimization
Parallelism
Security
Correctness
CFA++
CFA
Static analysis

Software “engineering”



Why we need software *engineering*

Security vulnerabilities

$80 billion in cyber-crime each year.

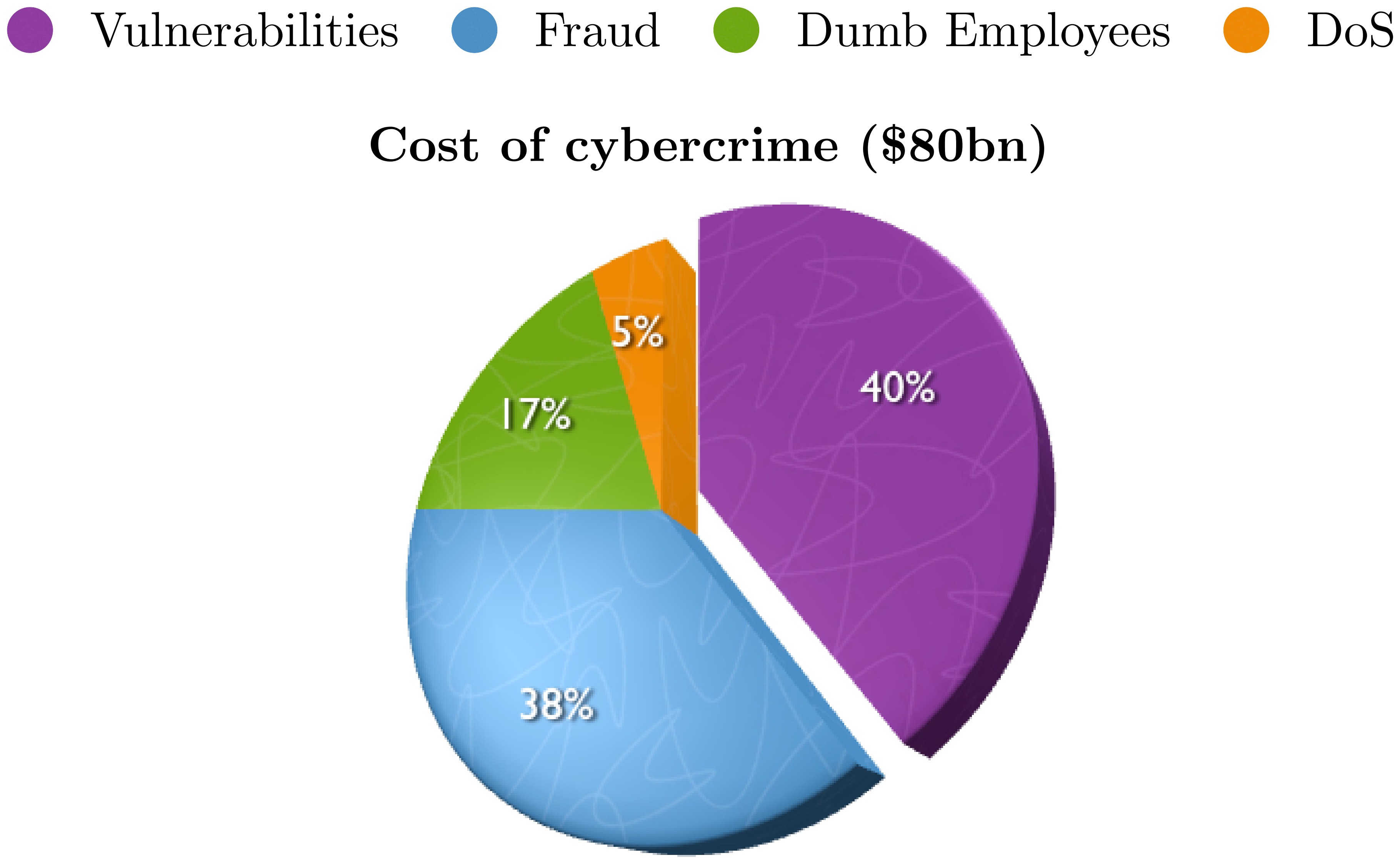
FBI

Cost and cause of insecurity

Vulnerabilities Fraud Dumb Employees DoS

Cost of cybercrime ($80bn)

Cost and cause of insecurity

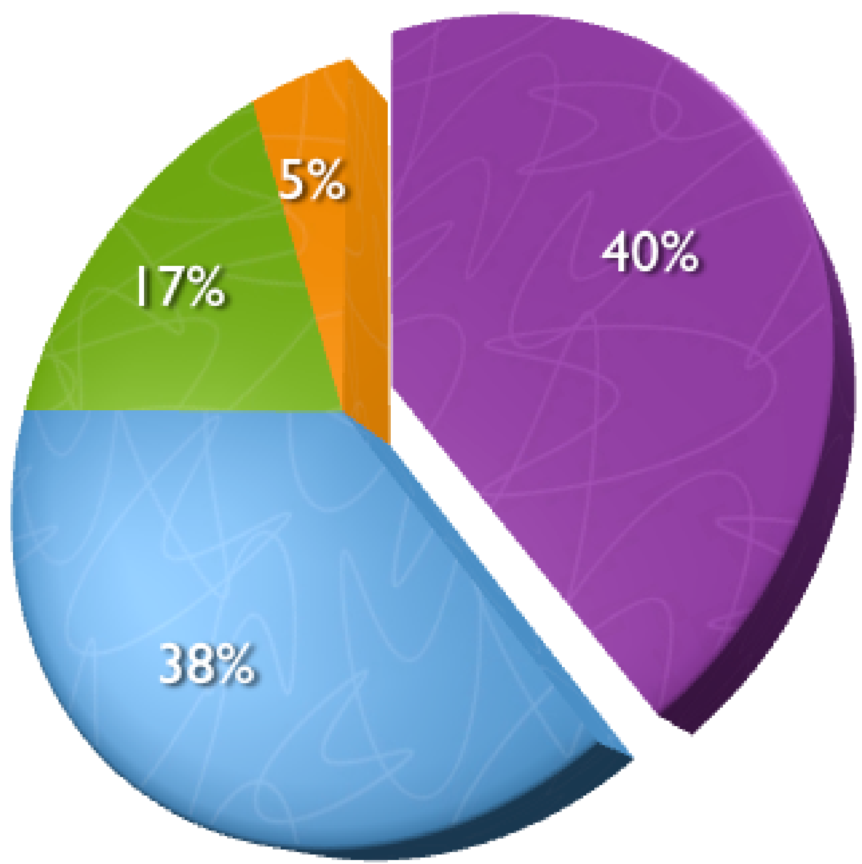


Source: CSI/FBI Survey 2007

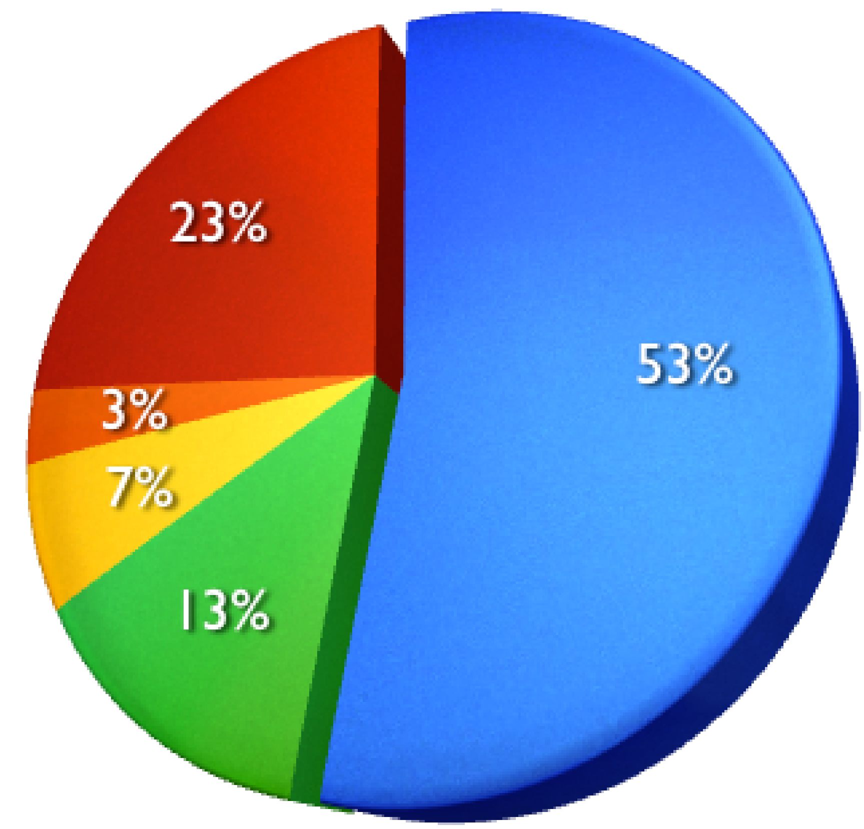
Cost and cause of insecurity

● 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

Software bugs

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

NIST

Why bugs are bad

`<exploding-rocket-video />`



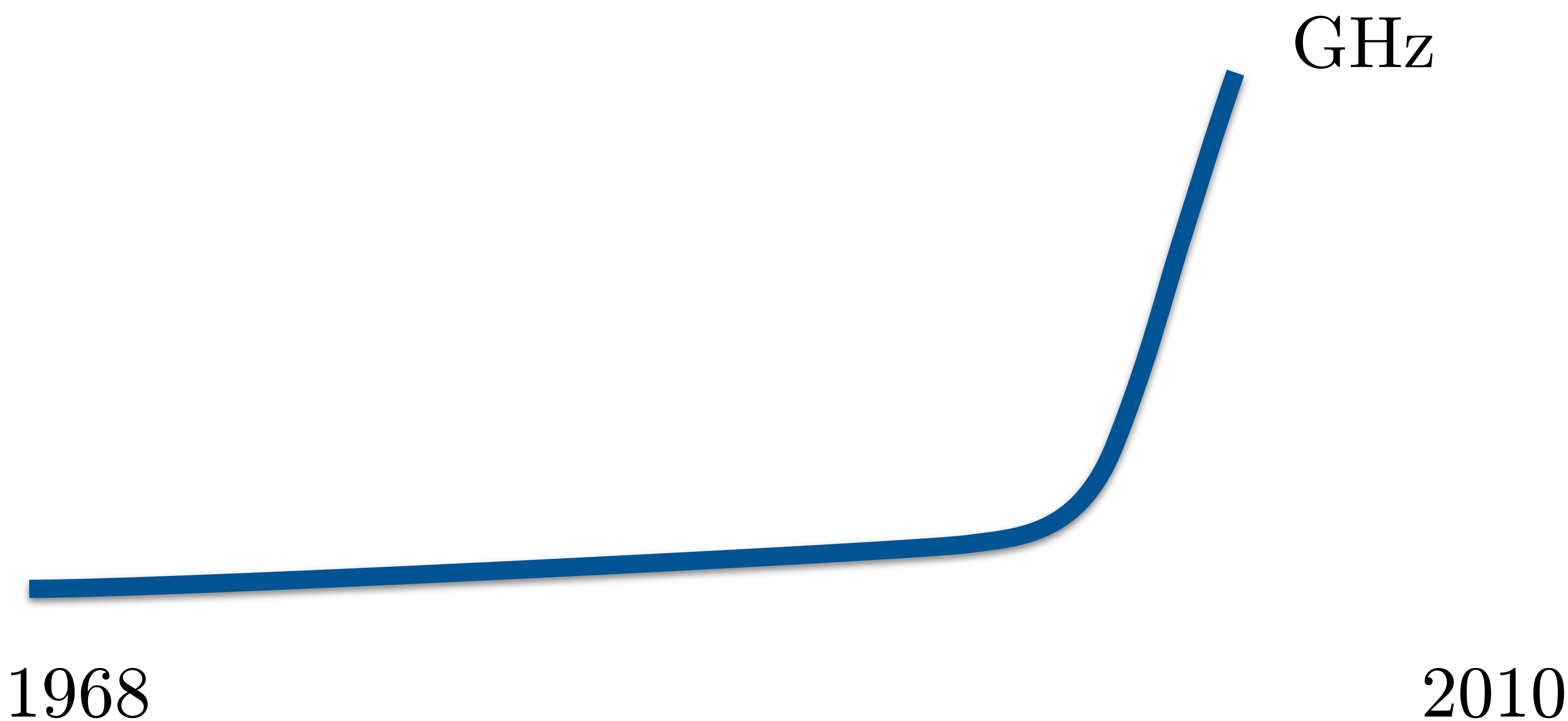
$10 billion




Parallelism is here

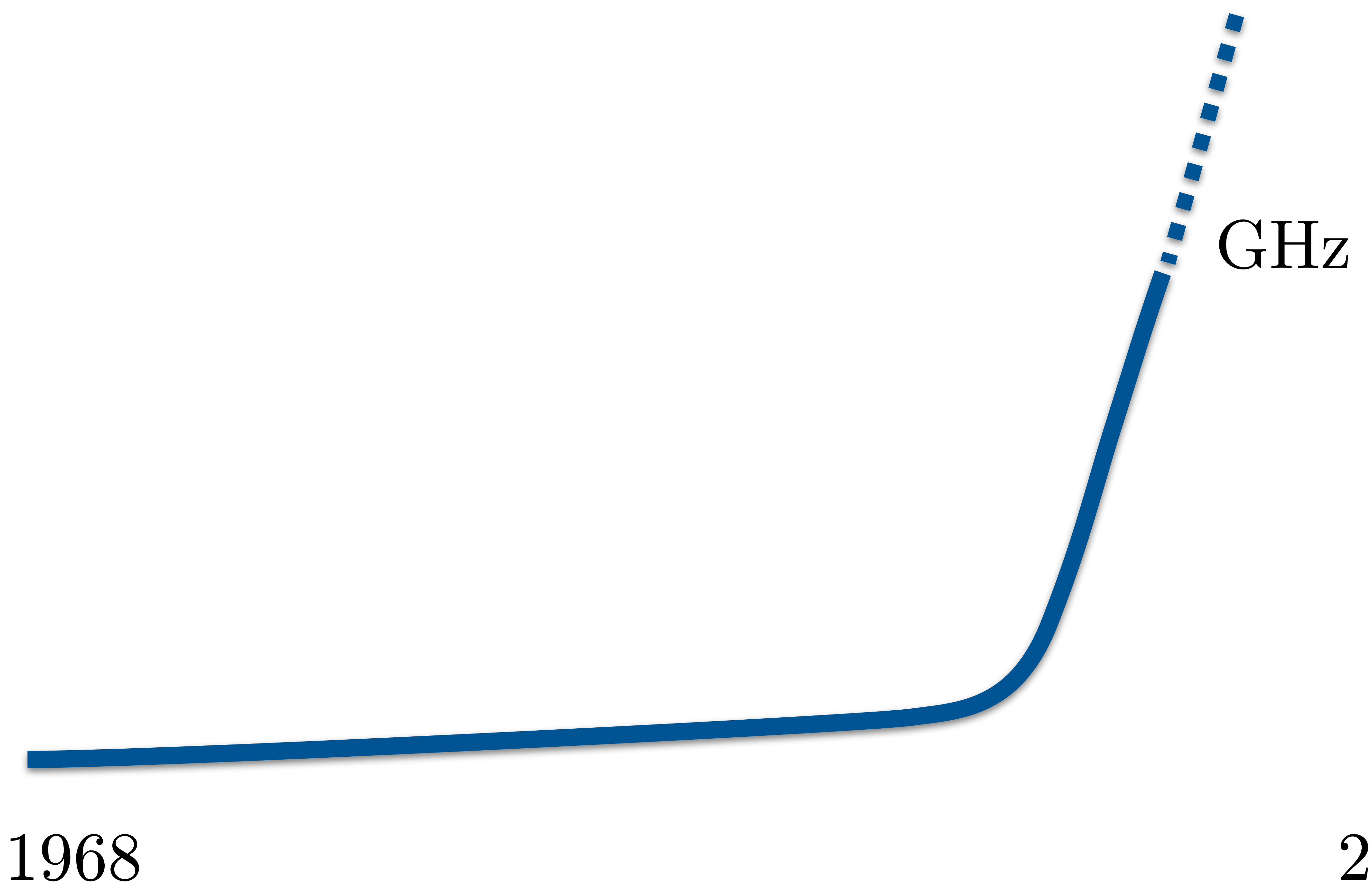
Source: Intel

Parallelism is here



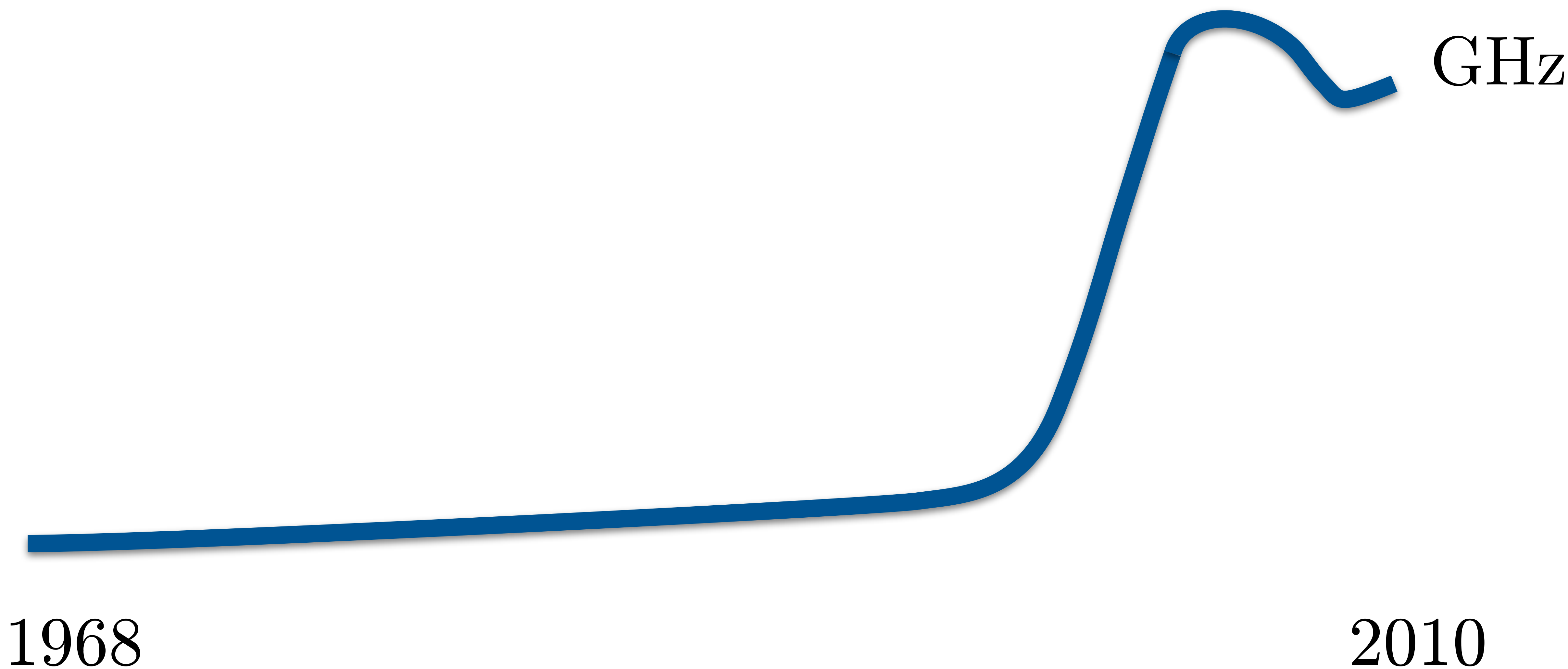
Source: Intel

Parallelism is here



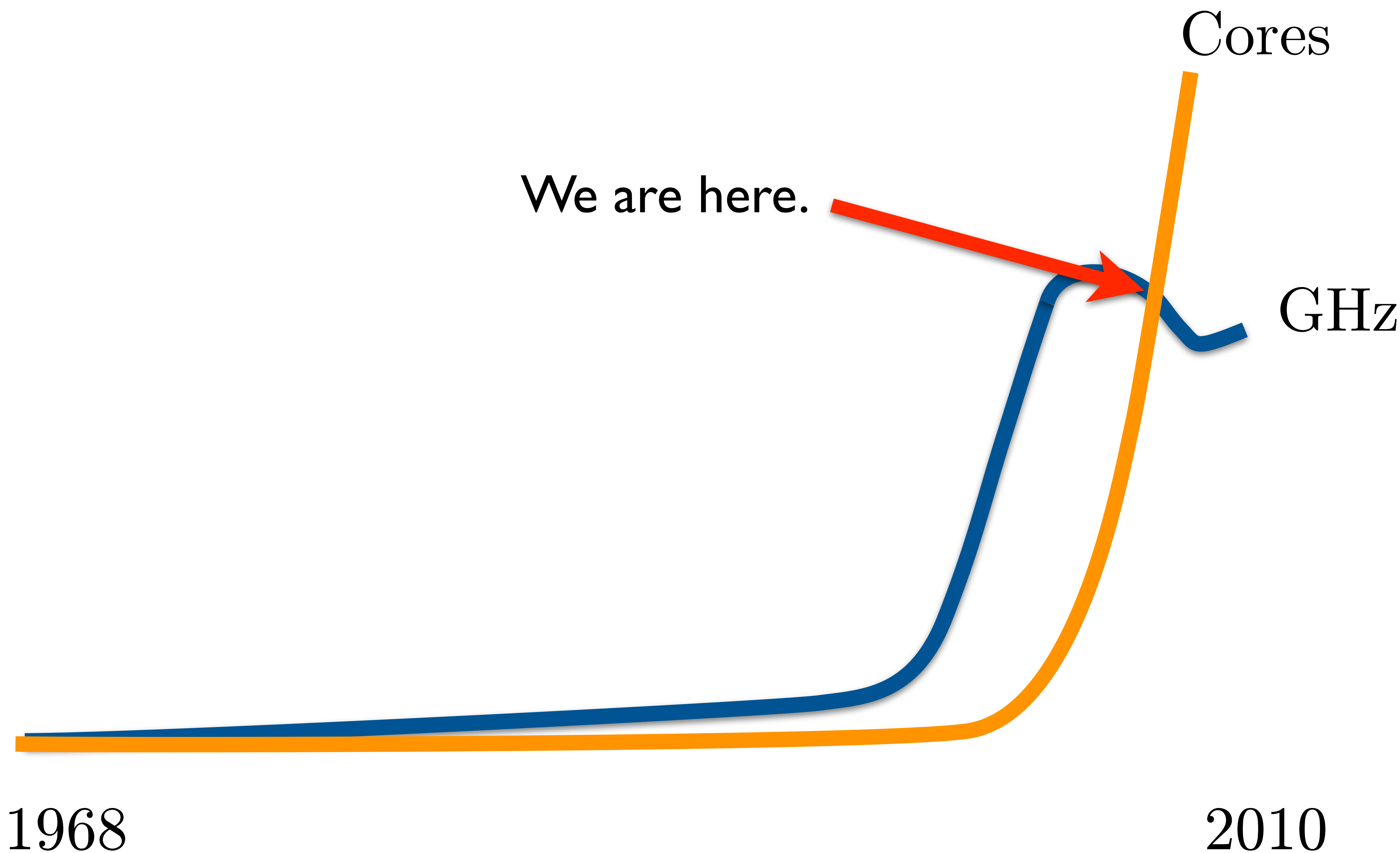
Source: Intel

Parallelism is here



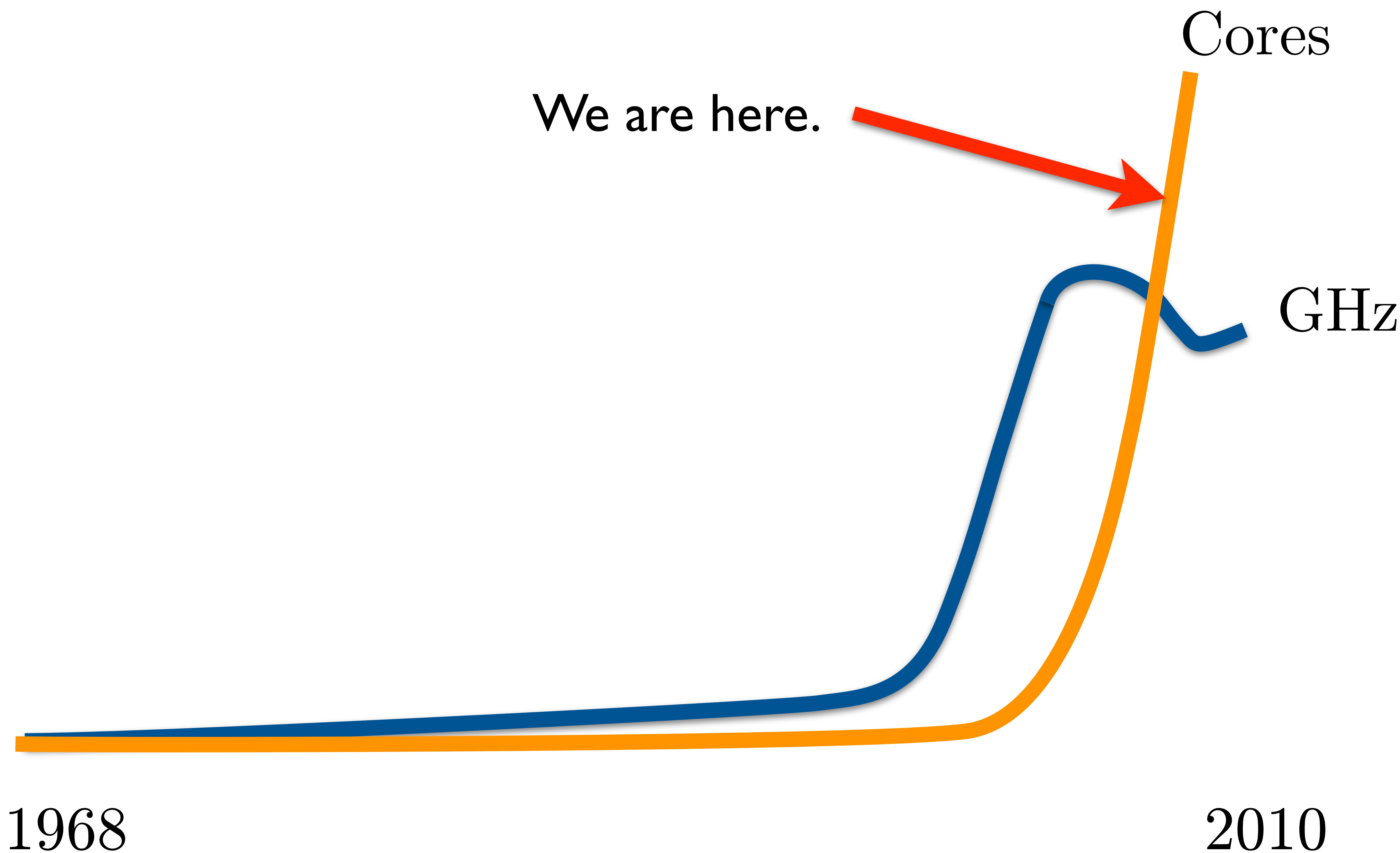
Source: Intel

Parallelism is here



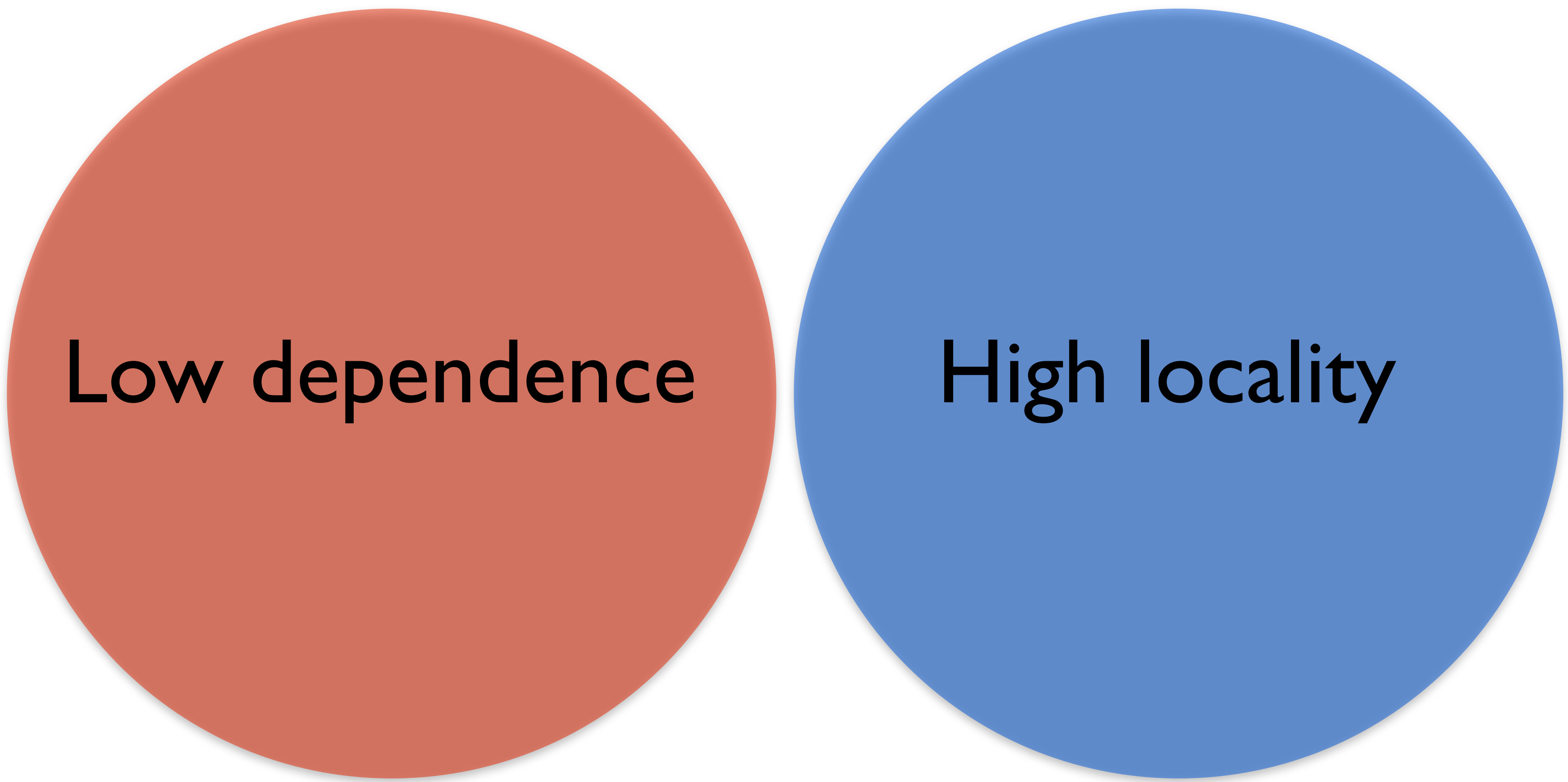
Source: Intel

Parallelism is here

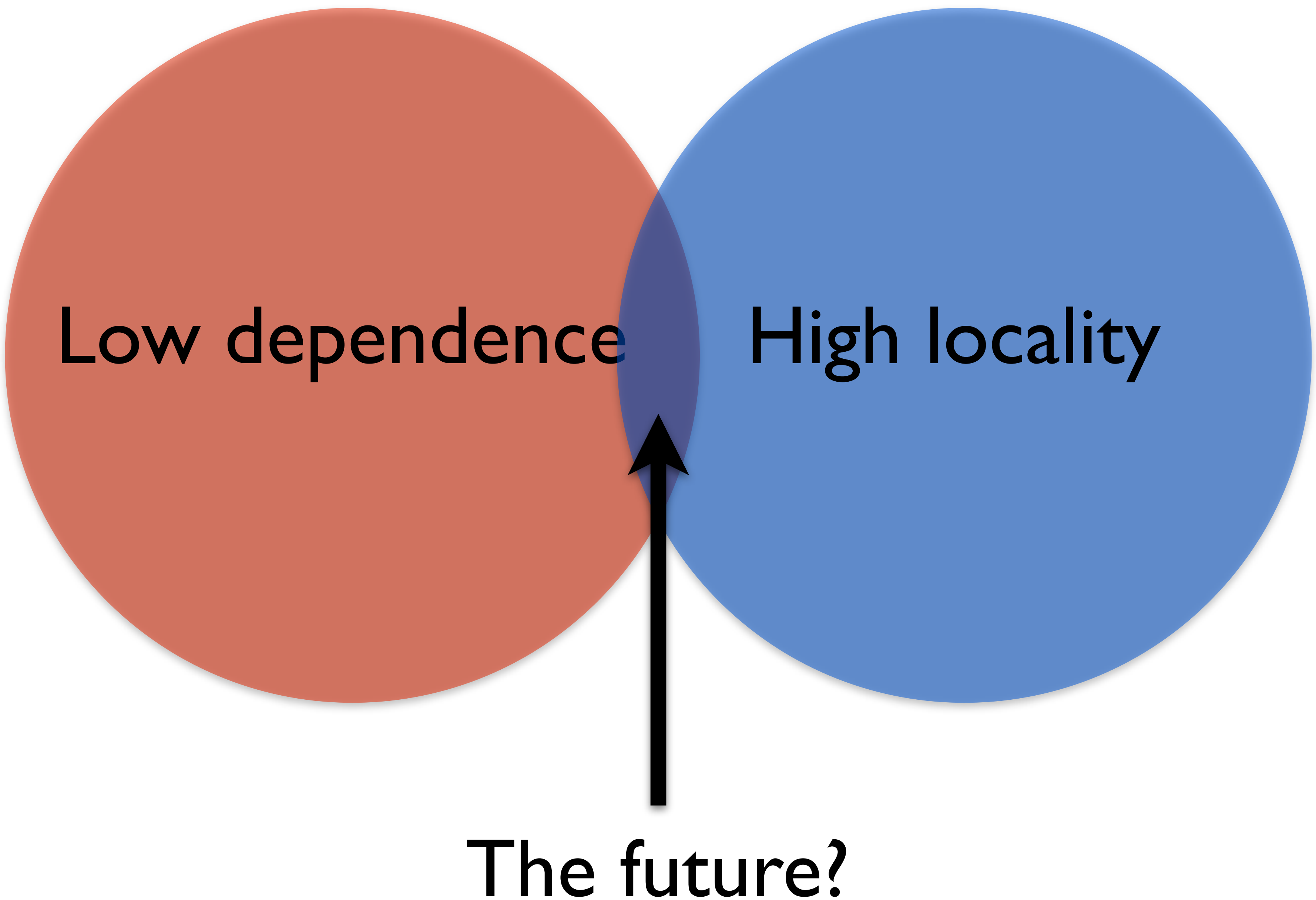


Source: Intel

Tomorrow's software



Tomorrow's software



Bottom line

If we want software that is...

- ...more parallel,
- ...more correct,
- ...more secure,

then we need engineering.

Why can’t we predict what software will do?

Why can’t we predict what software will do?

Because Alan Turing said we can’t.

“Thou shalt not write a program which determines whether a program halts.”

Banned by corollary

- Will a program eventually do **X**?
- Will a program never do **Y**?

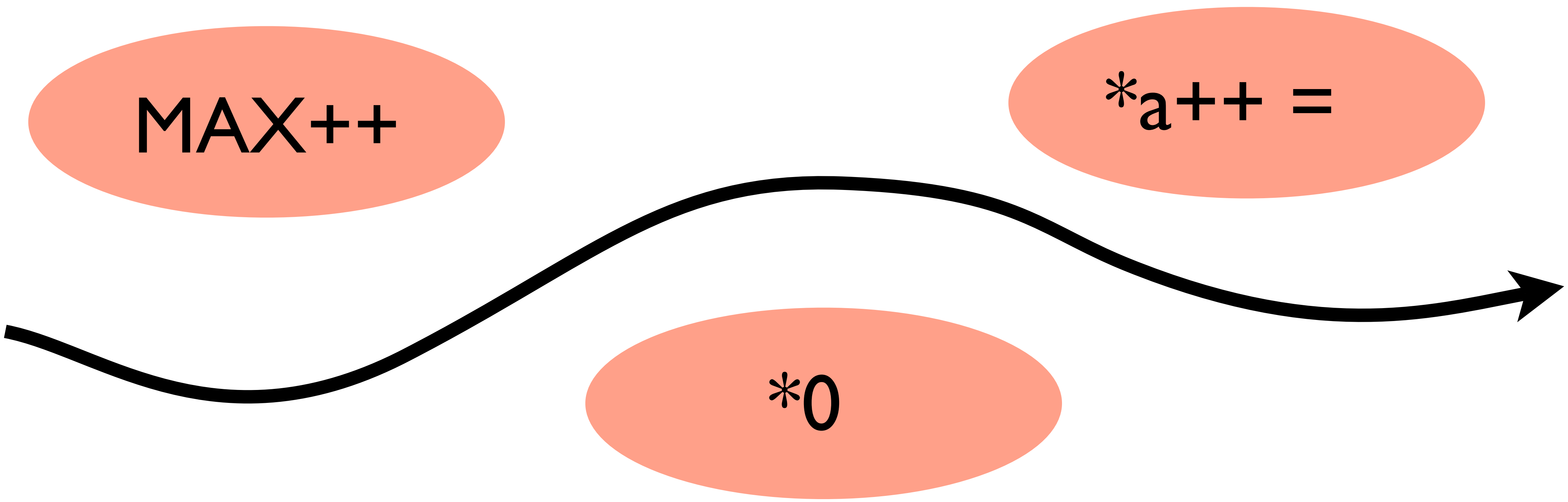
A “loop” hole

- Always answering “yes” or “no” is impossible.
- Answering “yes,” “no” or “maybe” is allowed.

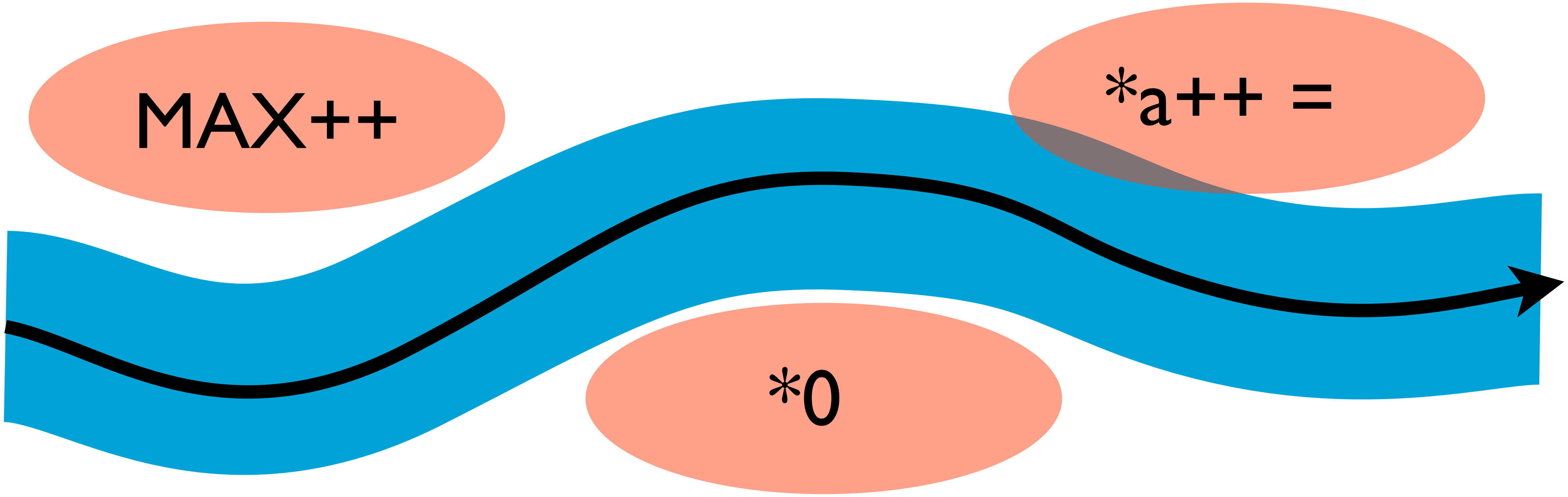
The static analysis game

The static analysis game

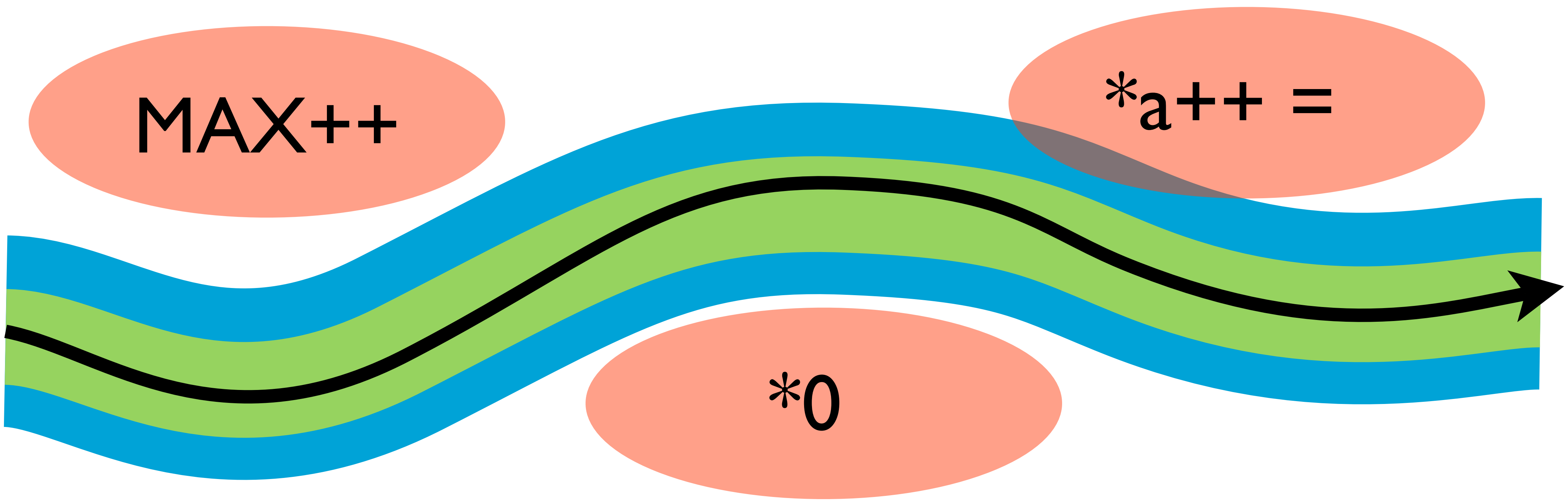
The static analysis game



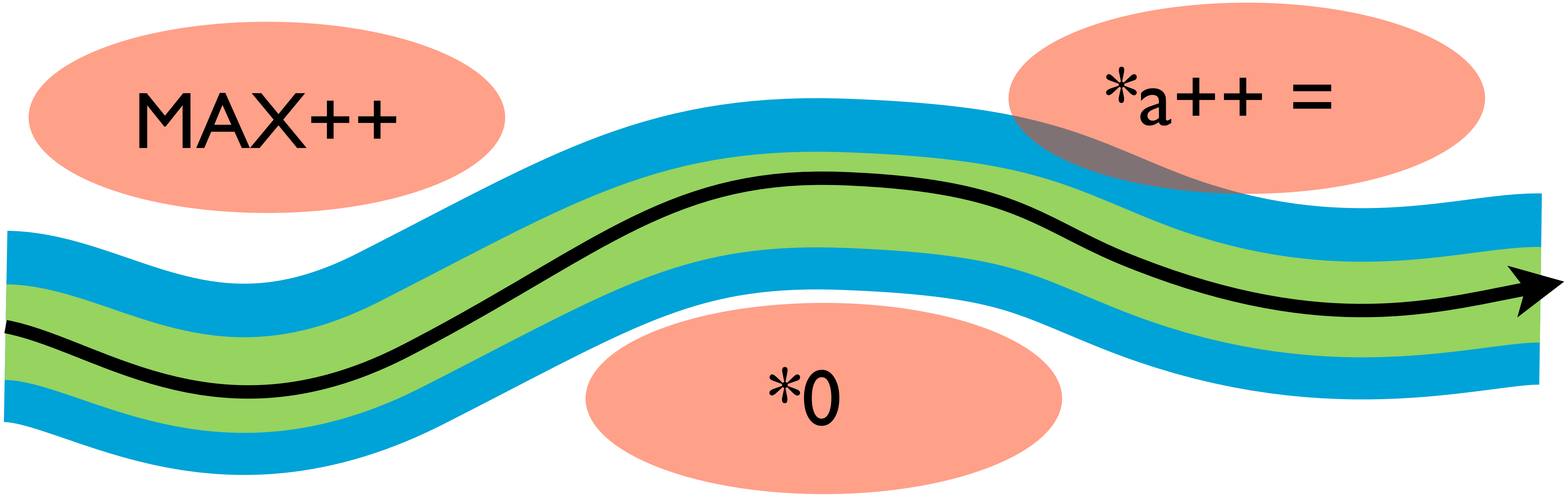
The static analysis game



The static analysis game



The static analysis game



“Full employment theorem.” -Appel

Why analyzing modern software is hard

What happens here?

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

What happens here?

What is `animal`?

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

What is `food`?

What happens here?

```
void process (Animal animal) {
    food = world.gather() ;
    animal.eat(food);
}
```

What happens here?

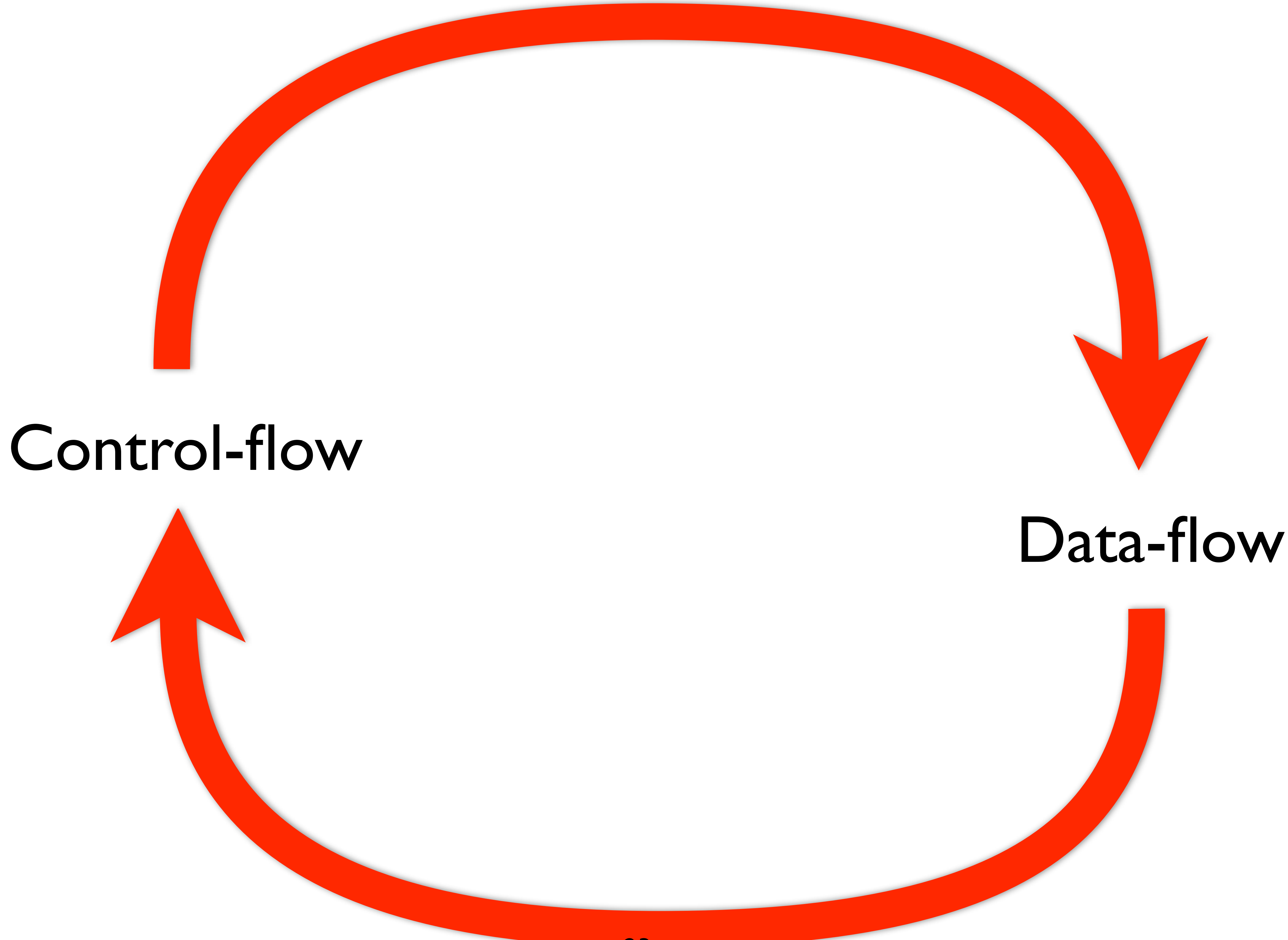
Who calls `process`?

```
void process (Animal animal) {
    food = world.gather() ;
    animal.eat(food);
}
```

What is `world`?

The control-flow problem

The control-flow problem



Gatekeeper

Before we can do anything interesting,
we must bound interprocedural control-flow.

Essence of the problem

Value = Object

Essence of the problem

Value = Object

= Class + Record

Essence of the problem

$$
\begin{aligned}
\text{Value} &= \text{Object} \\
&= \text{Class} + \text{Record} \\
&\subseteq \text{Code} + \text{Data}
\end{aligned}
$$

Which language
is the paragon of
value = code + data?

λ-calculus

Assertion

If we can analyze λ-calculus expressions,
we can analyze object-oriented programs.

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

Lisp and Scheme

- v ≡ `v`
- $f(e)$ ≡ `(f e)`
- $\lambda v.e$ ≡ `(lambda (v) e)`

λ-fortified

- Lisp
- SML
- Haskell
- Scala
- Java
- C#
- C++ (Boost)
- Python
- Ruby
- Smalltalk
- JavaScript
- PHP(!)

One rule: β-reduction

$$(\lambda v.v^2)(3)$$

One rule: β-reduction

$$3^2$$

Another interpretation

- Functions ≡ Closures
- $Closure = \lambda \times Env$
- $Env = Var \rightarrow Value$
- Ex: $(\lambda x.x + z, [z \mapsto 1])$

Essence of the essence

- Value = Code + Data
- Closure = λ + Environment

Control-flow question

Given a call site `f(x)`, what could `f` be?

Control-flow scenarios

`f(x)`

Control-flow scenarios

```
let f = λz.z
    in f(x)
```

Control-flow scenarios

```
λf.f(x)
```

Control-flow analysis

A **control-flow analysis** conservatively approximates the procedures which may be invoked at a given call site.

Control-flow analysis

A **control-flow analysis** conservatively approximates the procedures which may be invoked at a given call site.

A **value-flow analysis** conservatively approximates the values to which an expression may evaluate.

Techniques for CFA

- *Ad hoc* techniques
- Constraint-solving
- Type-based analysis
- Abstract interpretation

Techniques for CFA

- *Ad hoc* techniques
- Constraint-solving
- Type-based analysis
- Abstract interpretation

Techniques for CFA

- *Ad hoc* techniques
- Constraint-solving
- Type-based analysis
- Abstract interpretation

Constraint-based 0CFA

What is 0CFA?

Lambda-flow analysis.

The 0CFA approximation

- Value = Code x Data
- Closure = Lambda x Env
- Object = Class x Record

The 0CFA approximation

- Value = Code
- Closure = Lambda
- Object = Class

0CFA

$$e_1(e_2)$$

0CFA

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

$e_1(e_2)$

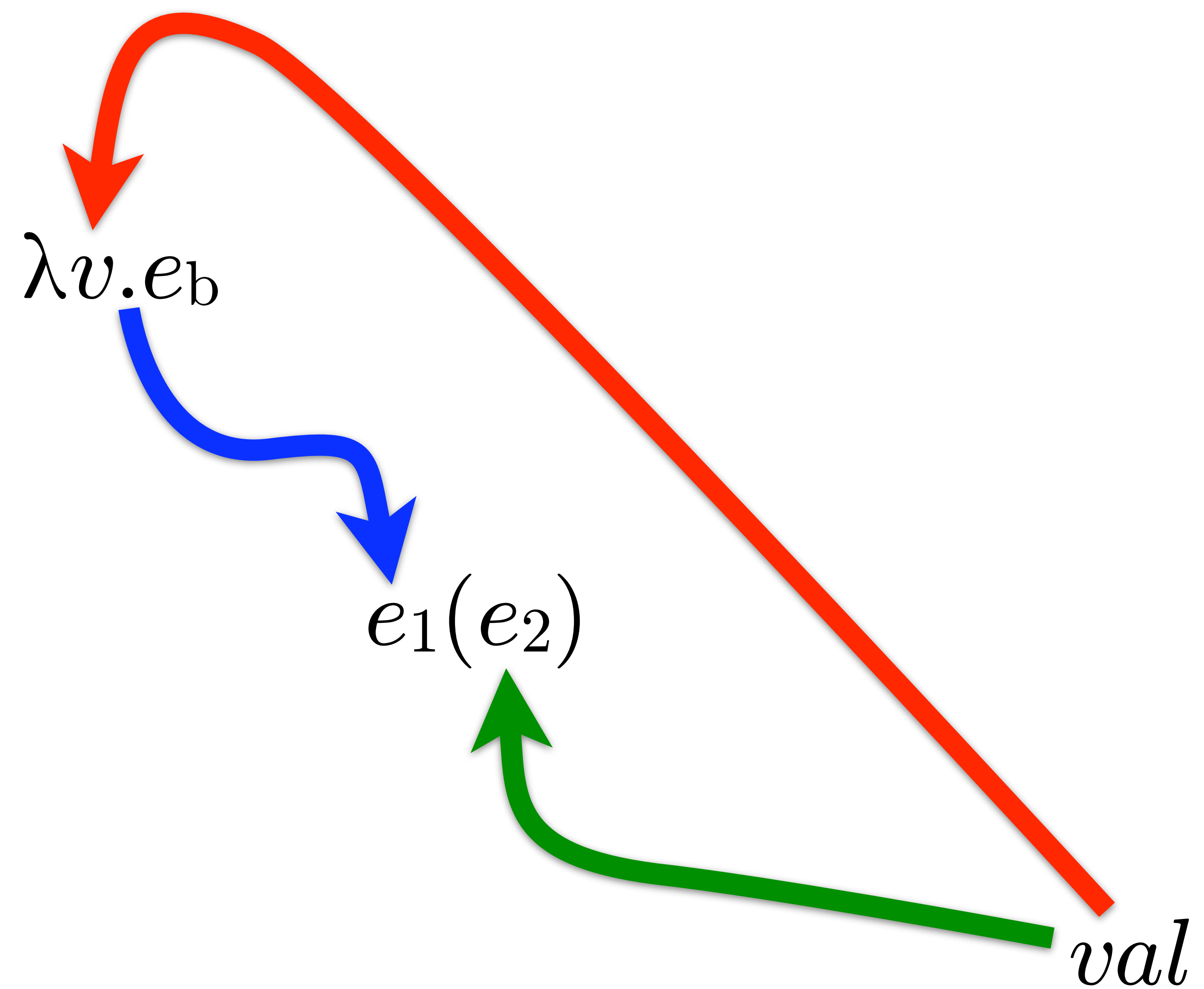
0CFA

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

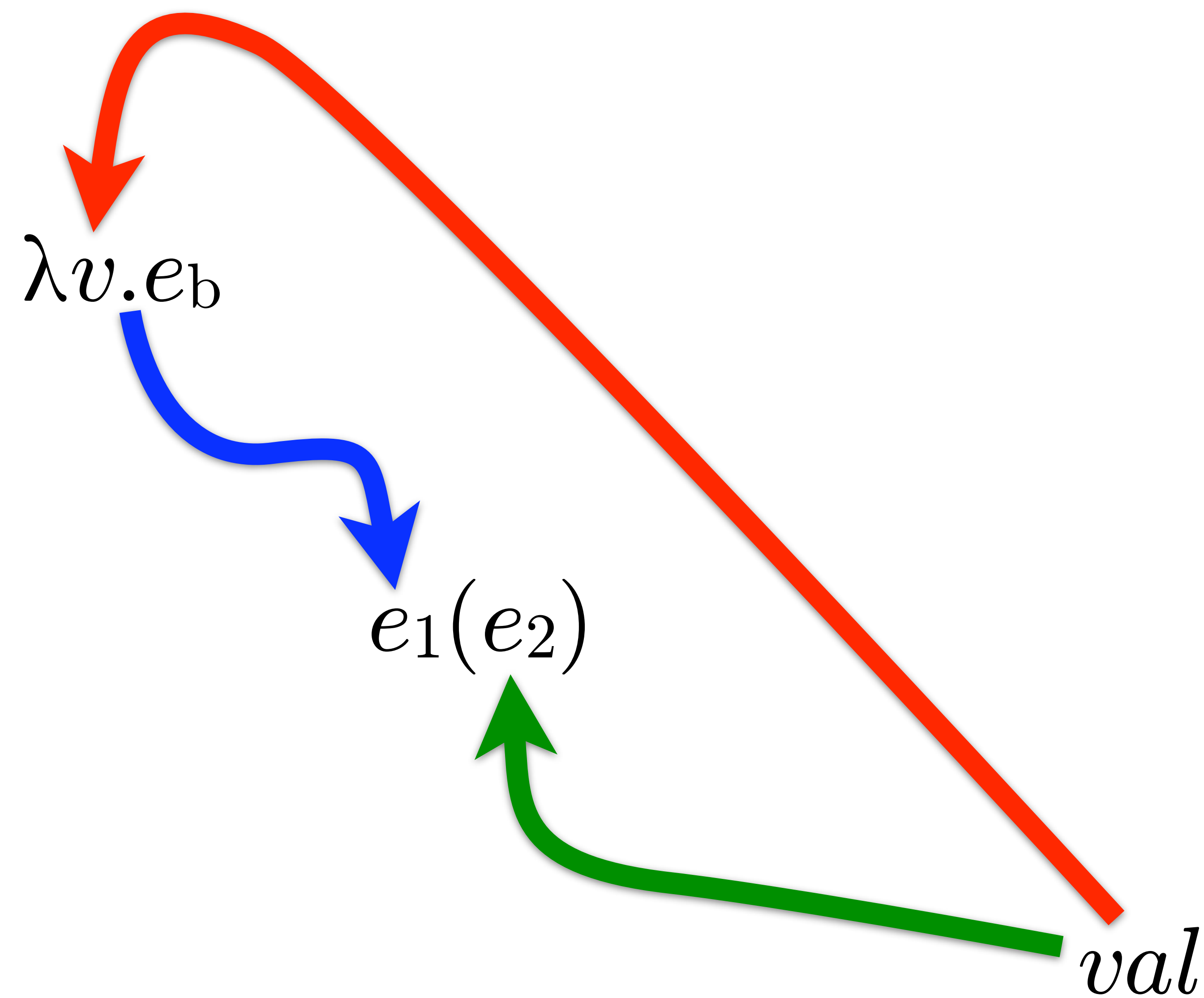
$e_1(e_2)$

val

0CFA



0CFA



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

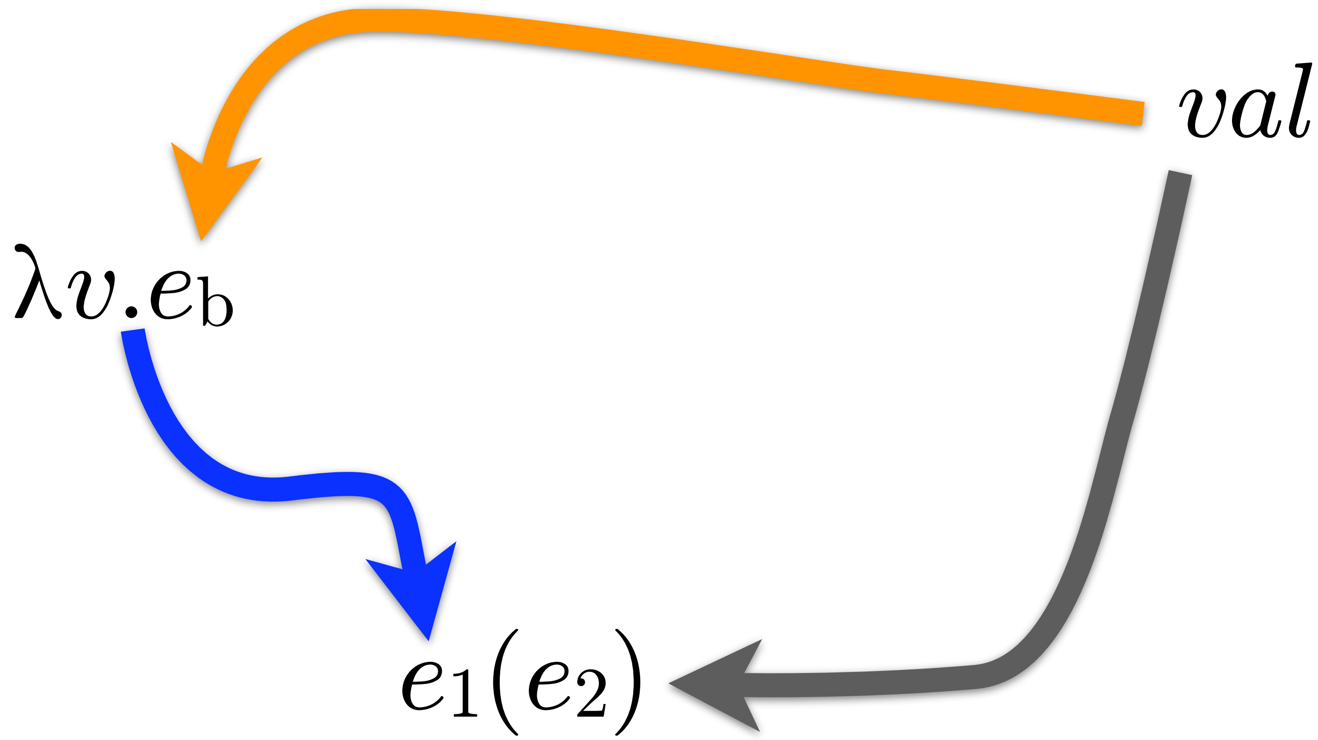
0CFA

val

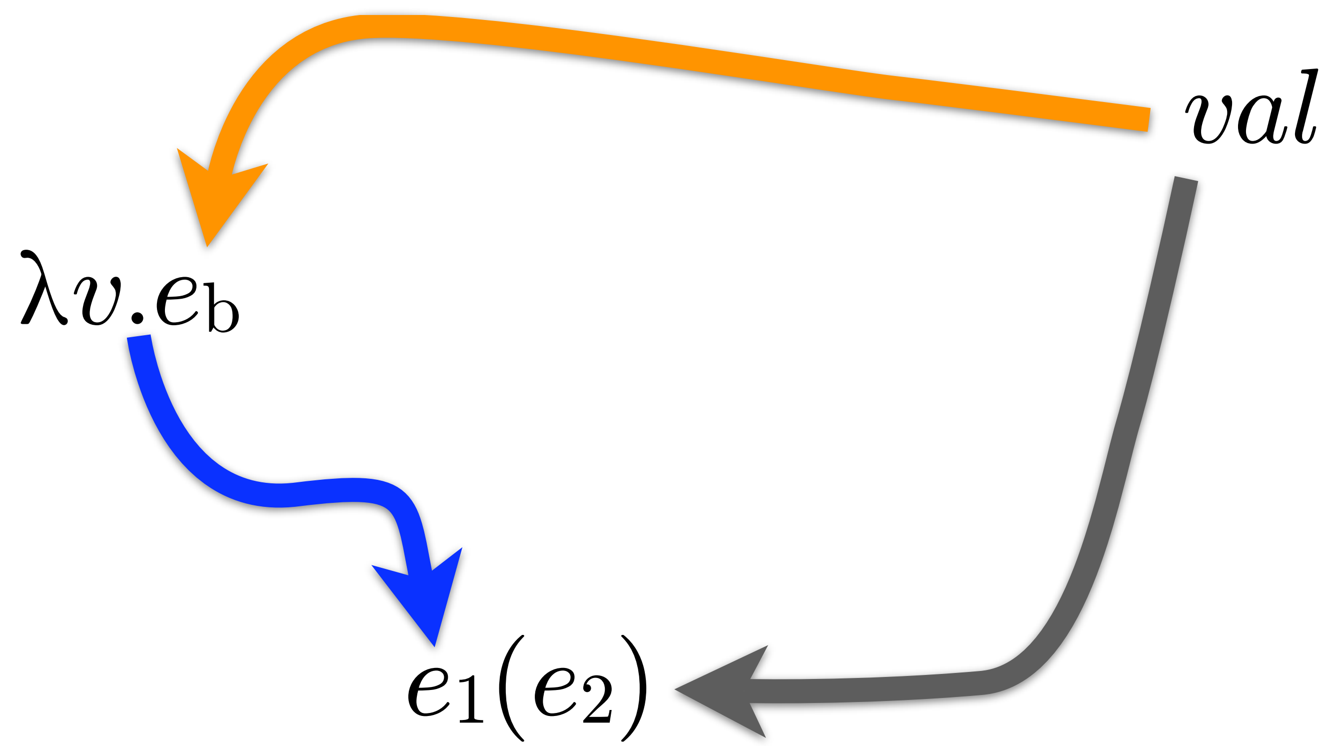
$\lambda v.e_{\mathrm{b}}$

$e_1(e_2)$

0CFA



0CFA



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

0CFA

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

0CFA

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

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

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

0CFA (Palsberg, 1995)

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

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

$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1]}{\mathrm{FlowsTo}[e_2] \subseteq \mathrm{FlowsTo}[v]}$$

0CFA (Palsberg, 1995)

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

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

$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1]}{\mathrm{FlowsTo}[e_2] \subseteq \mathrm{FlowsTo}[v]}$$

0CFA (Palsberg, 1995)

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

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

+ Constraint Solver

$$\frac{\lambda v.e_{\mathrm{b}} \in \mathrm{FlowsTo}[e_1]}{\mathrm{FlowsTo}[e_2] \subseteq \mathrm{FlowsTo}[v]}$$

= Control-flow analysis

Applications

- Classic data-flow analysis
- Data-flow optimizations
- Global register allocation
- Defunctionalization
- Static method resolution
- Global constant propagation
- Global copy propagation
- Loop detection/optimization
- Escape analysis
- Constant folding

A problem with the "CFA-first" approach

Problem: Cross-flow

```
map f list
```

Problem: Cross-flow

```
fireMissile(n)          []

          map f list

petBunny(n)          [1,2,3]
```

Problem: Cross-flow

`fireMissile(n)` `[]`

`map f list`

`petBunny(n)` `[1,2,3]`

Problem: Cross-flow

```
fireMissile(n)                    []

              map f list

petBunny(n)                [1,2,3]
```

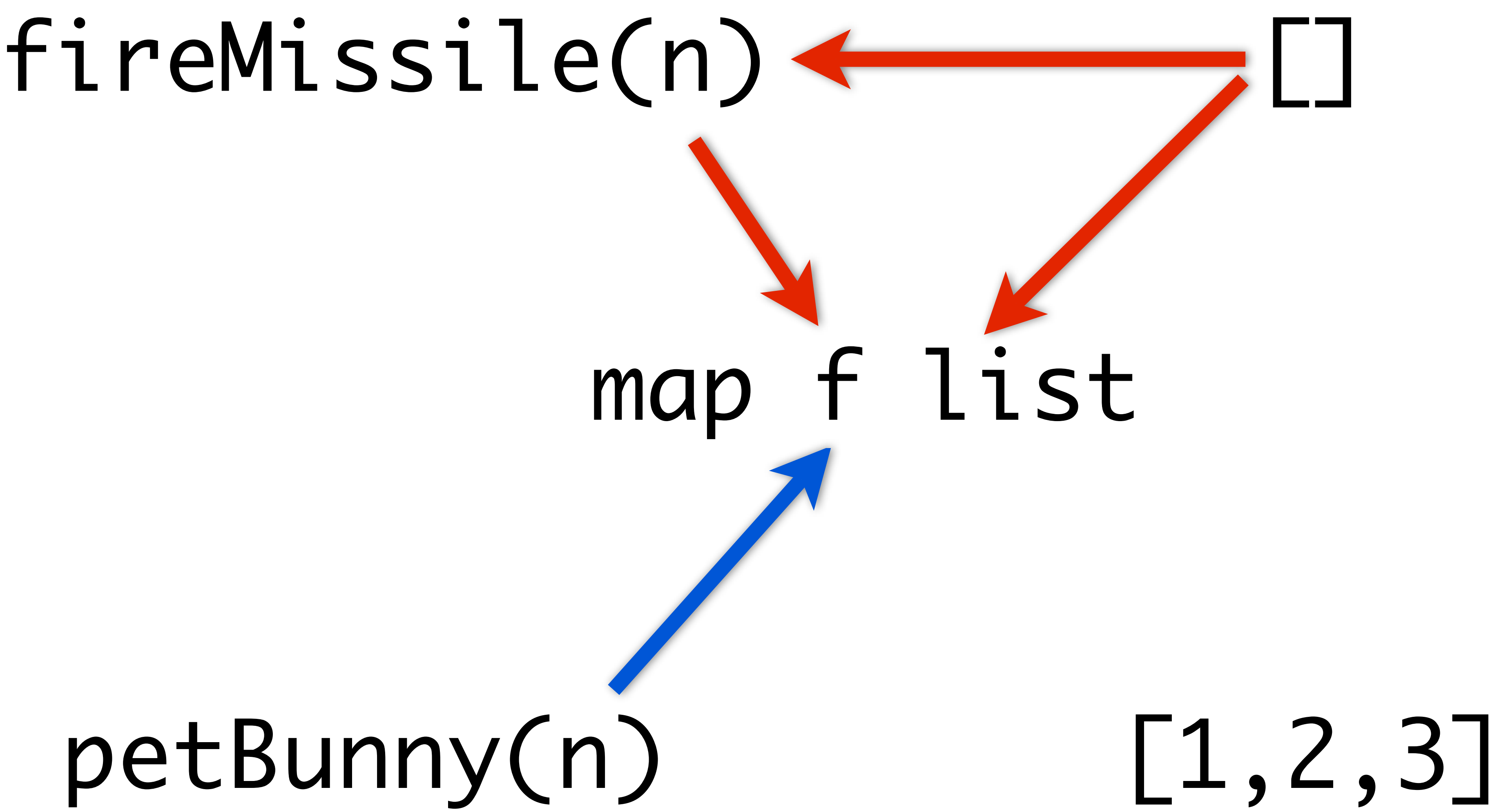
Problem: Cross-flow

`fireMissile(n)` `[]`

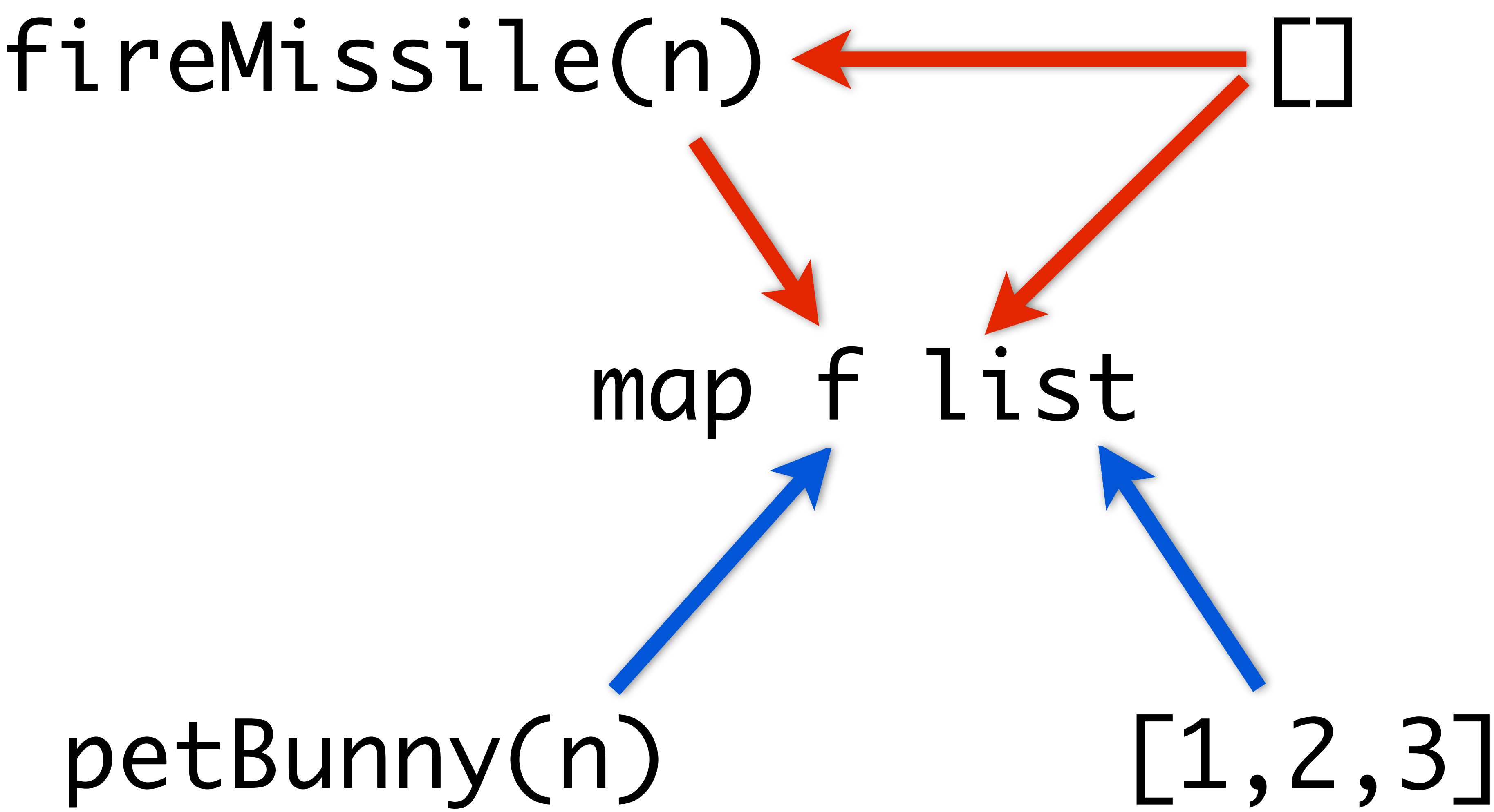
`map f list`

`petBunny(n)` `[1,2,3]`

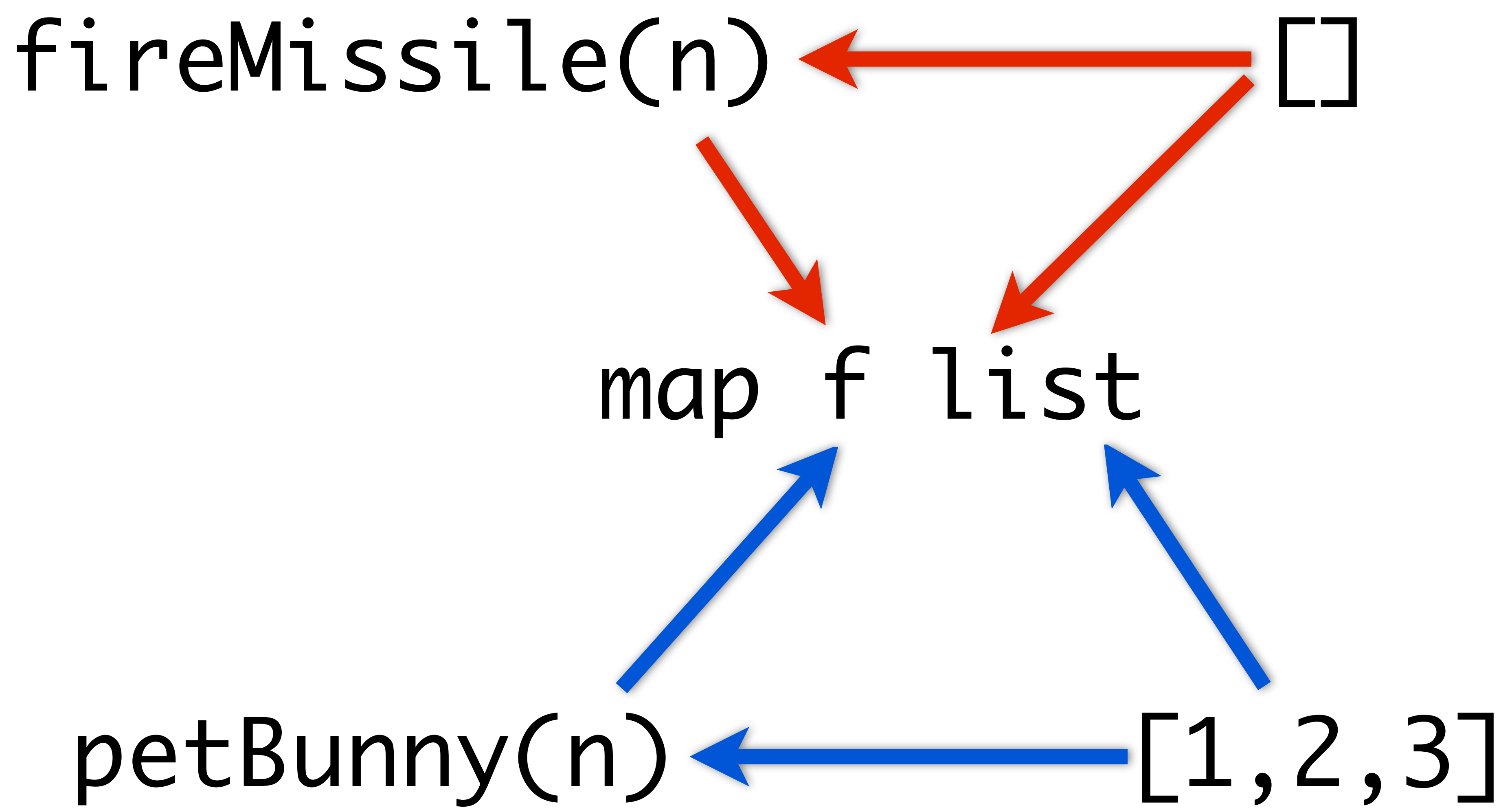
Problem: Cross-flow



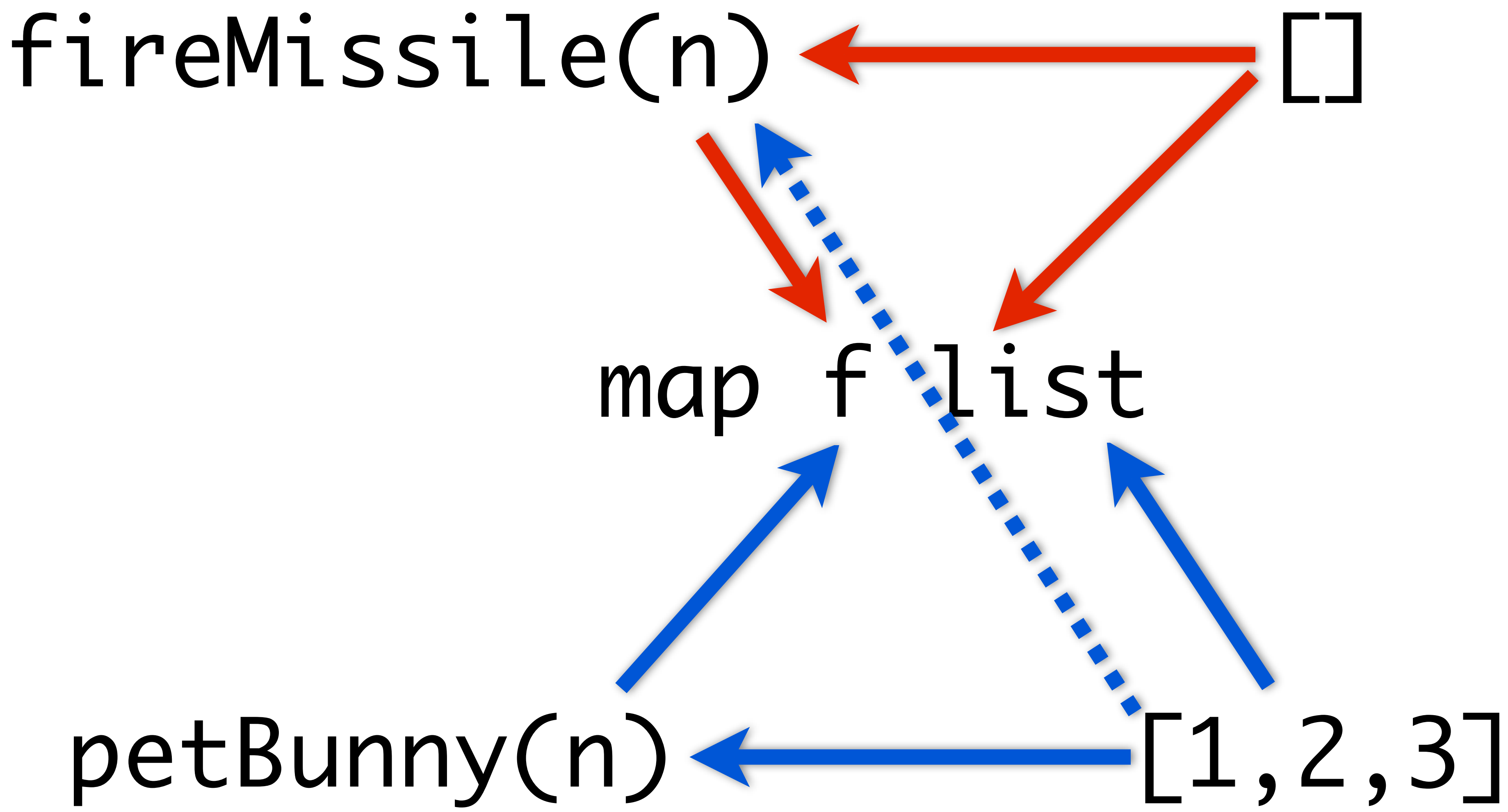
Problem: Cross-flow



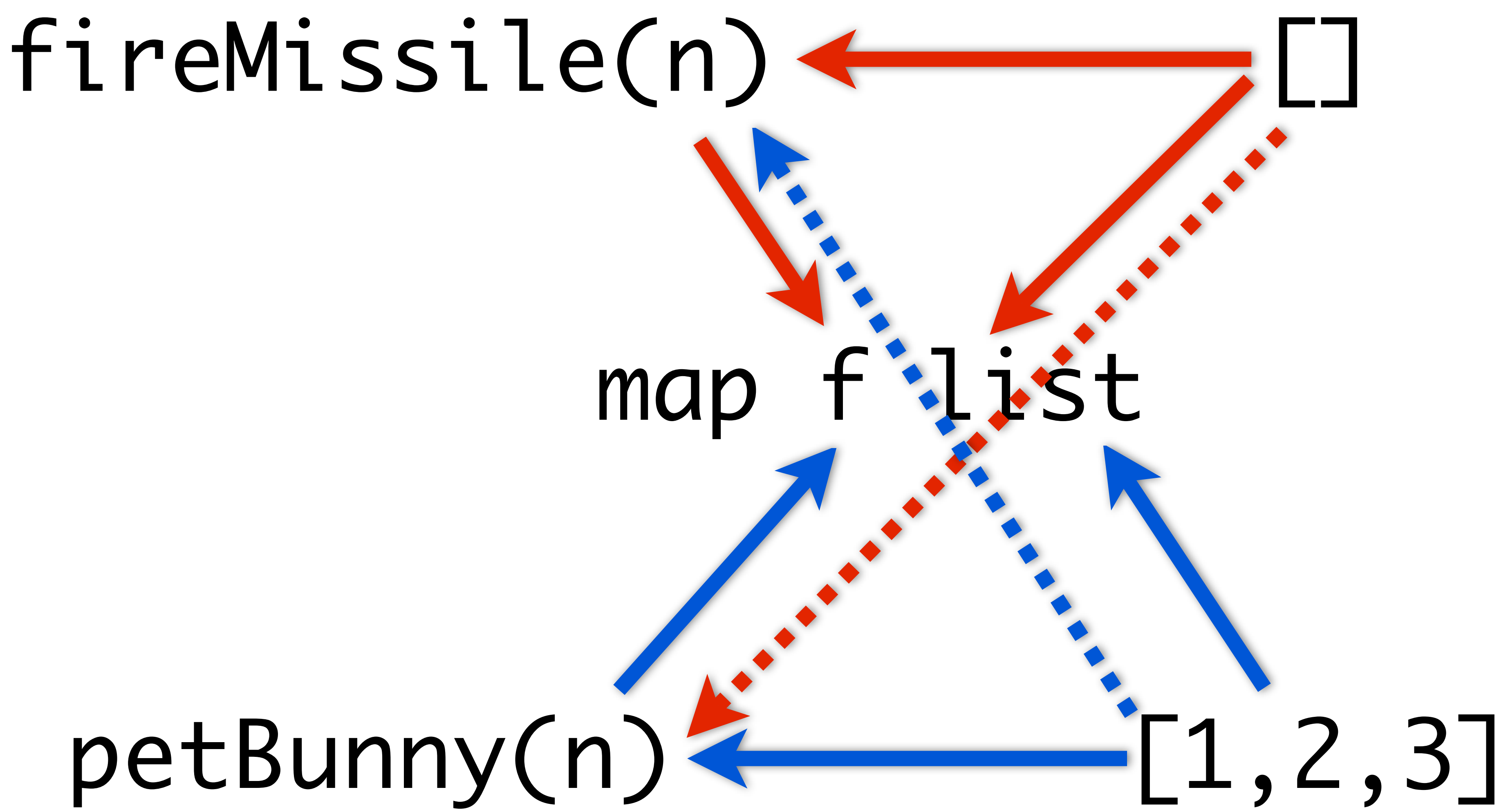
Problem: Cross-flow



Problem: Cross-flow



Problem: Cross-flow



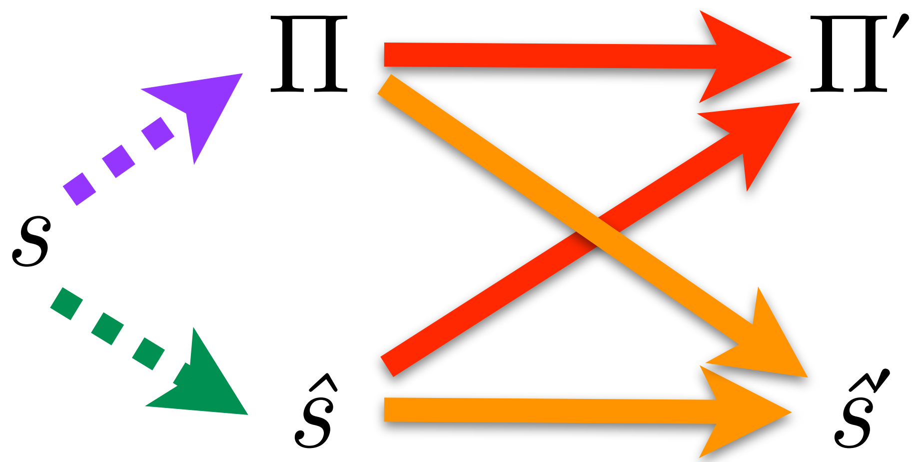
Solution platform: Small-step abstract interpretation

Small-step advantage

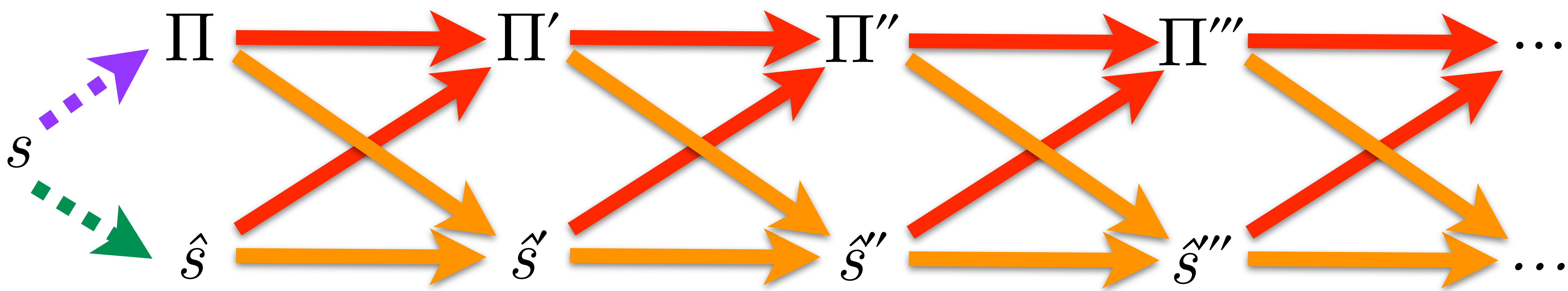
Small-step advantage

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

Small-step advantage



Small-step advantage



Small-step strategy

- Model program as infinite-state machine
- Approximate program with finite-state machine

Small-step machine

Small-step machine

- Convert program e into machine state s_0

$$\begin{array}{c} e \\ \big\downarrow \\ s_0 \end{array}$$

Small-step machine

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

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

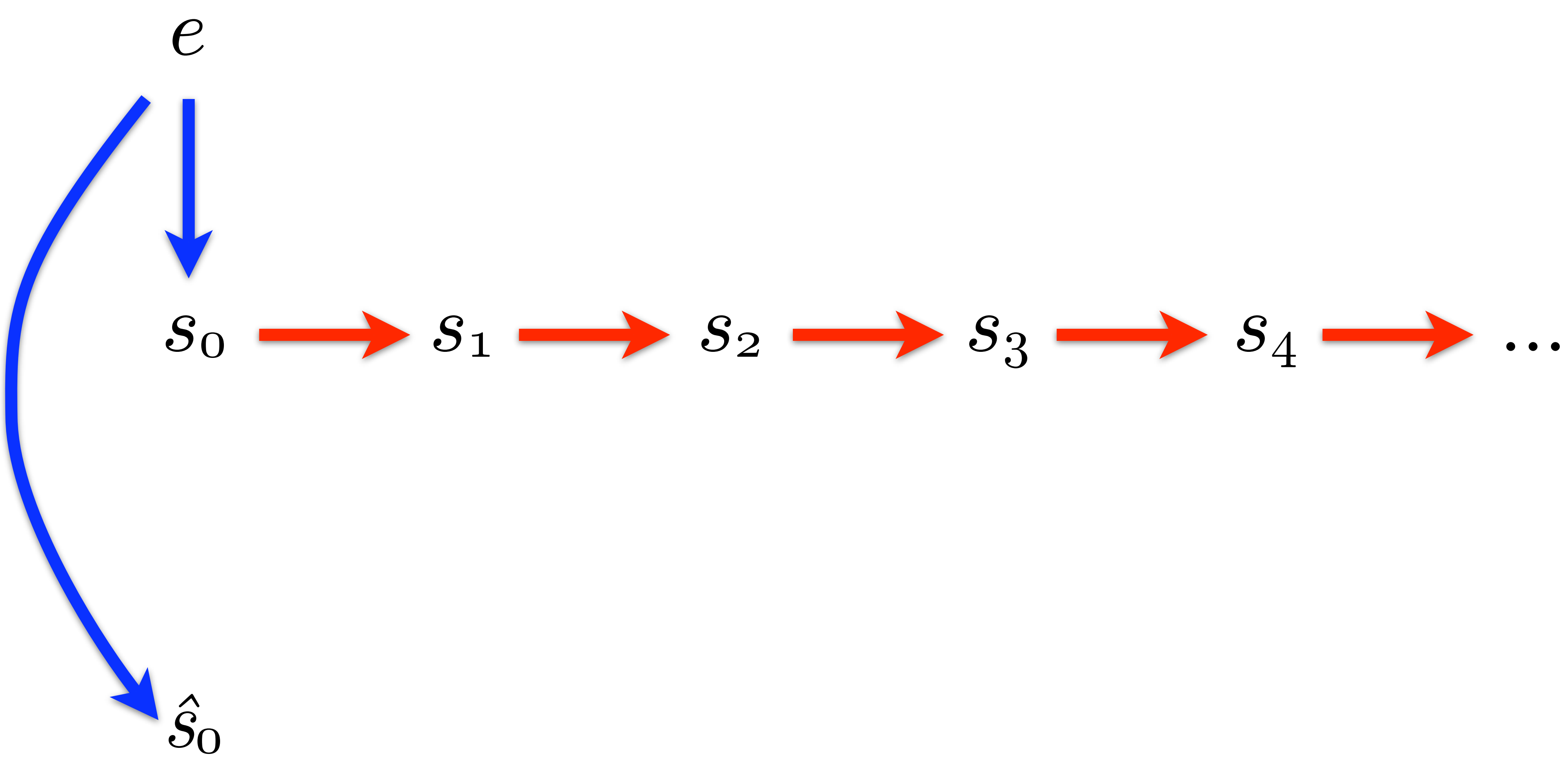
Abstract machine

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

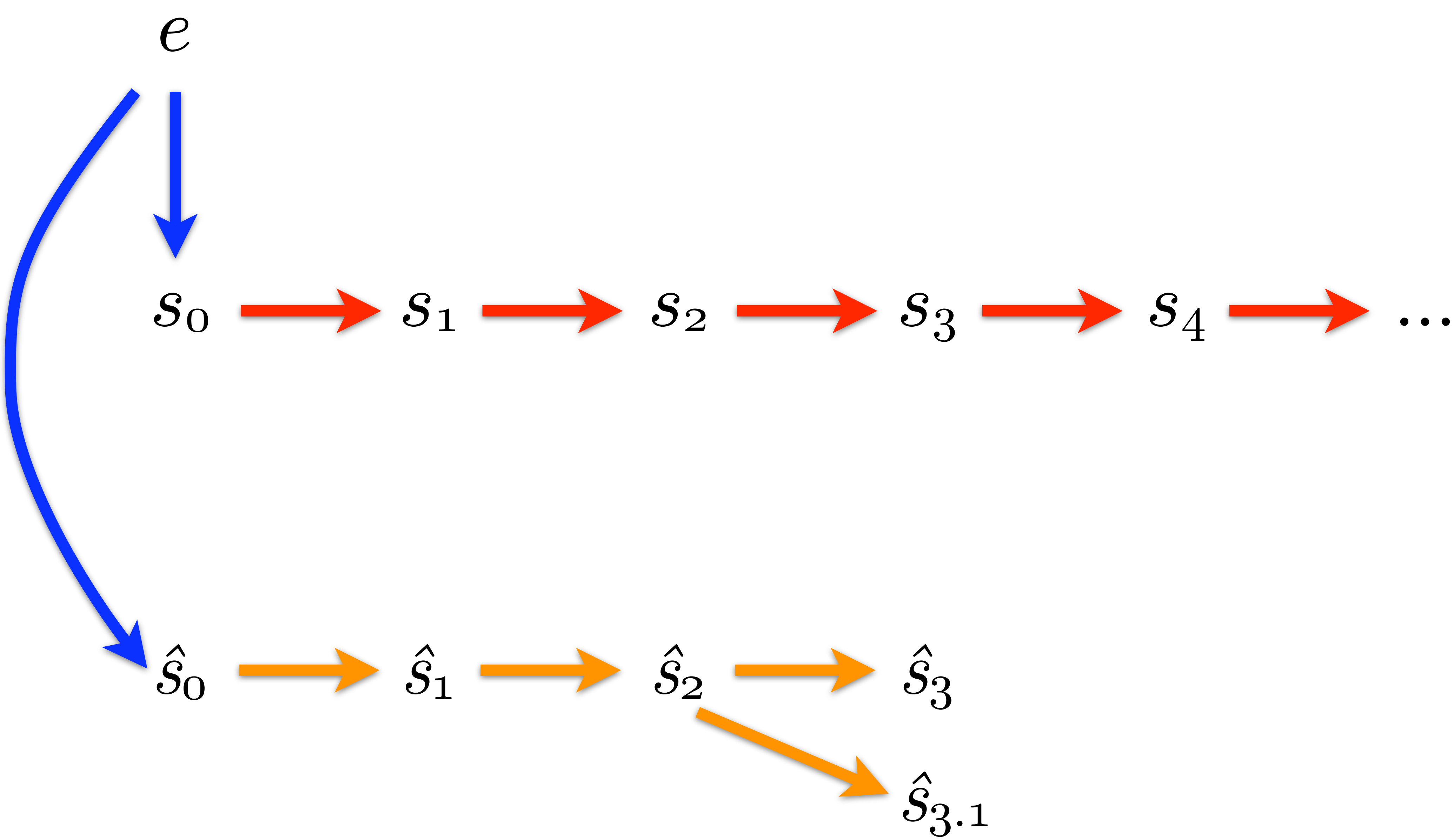
Abstract machine

$$\begin{array}{l} e \\ \downarrow \\ s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow s_4 \longrightarrow \ldots \end{array}$$

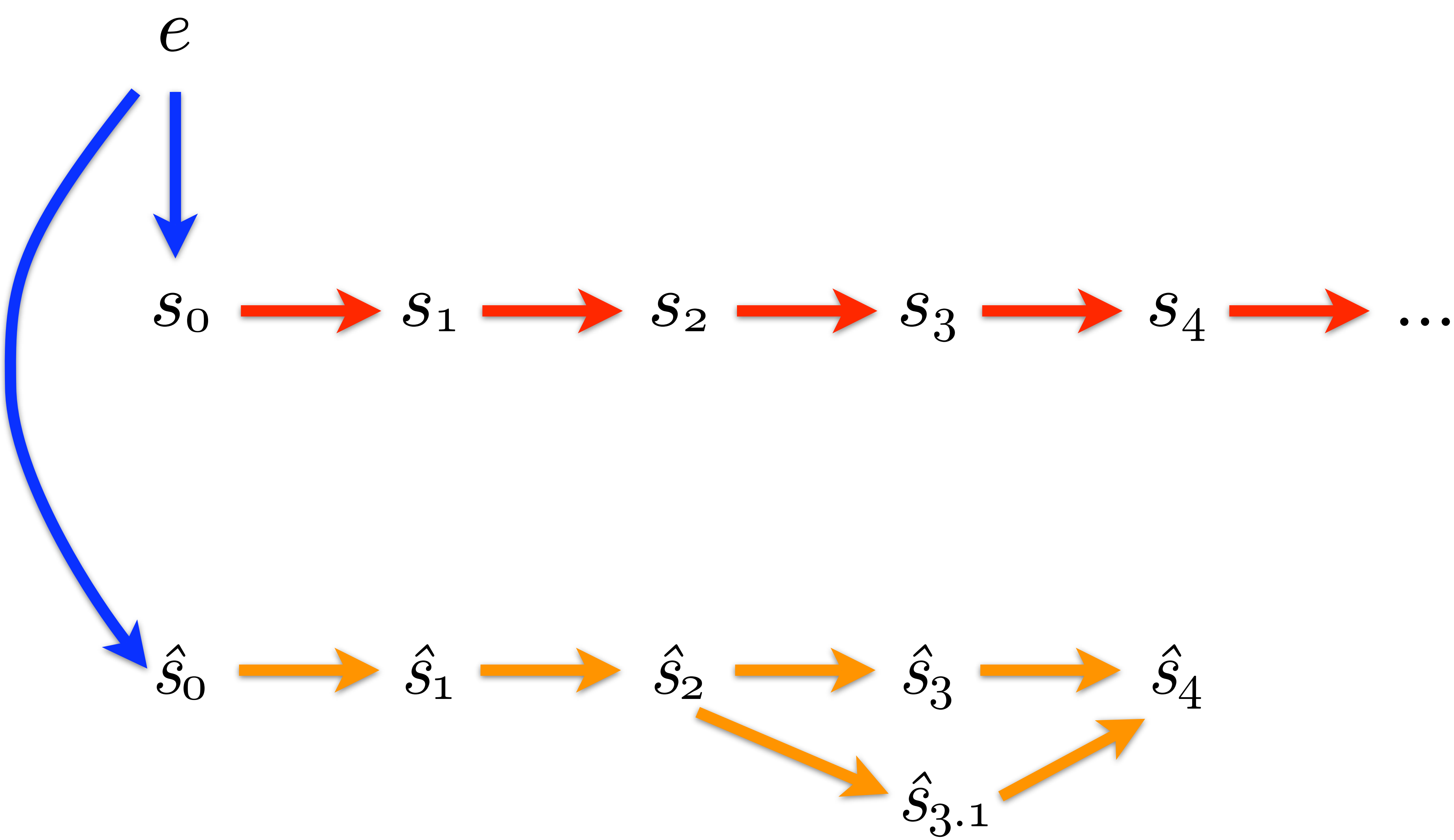
Abstract machine



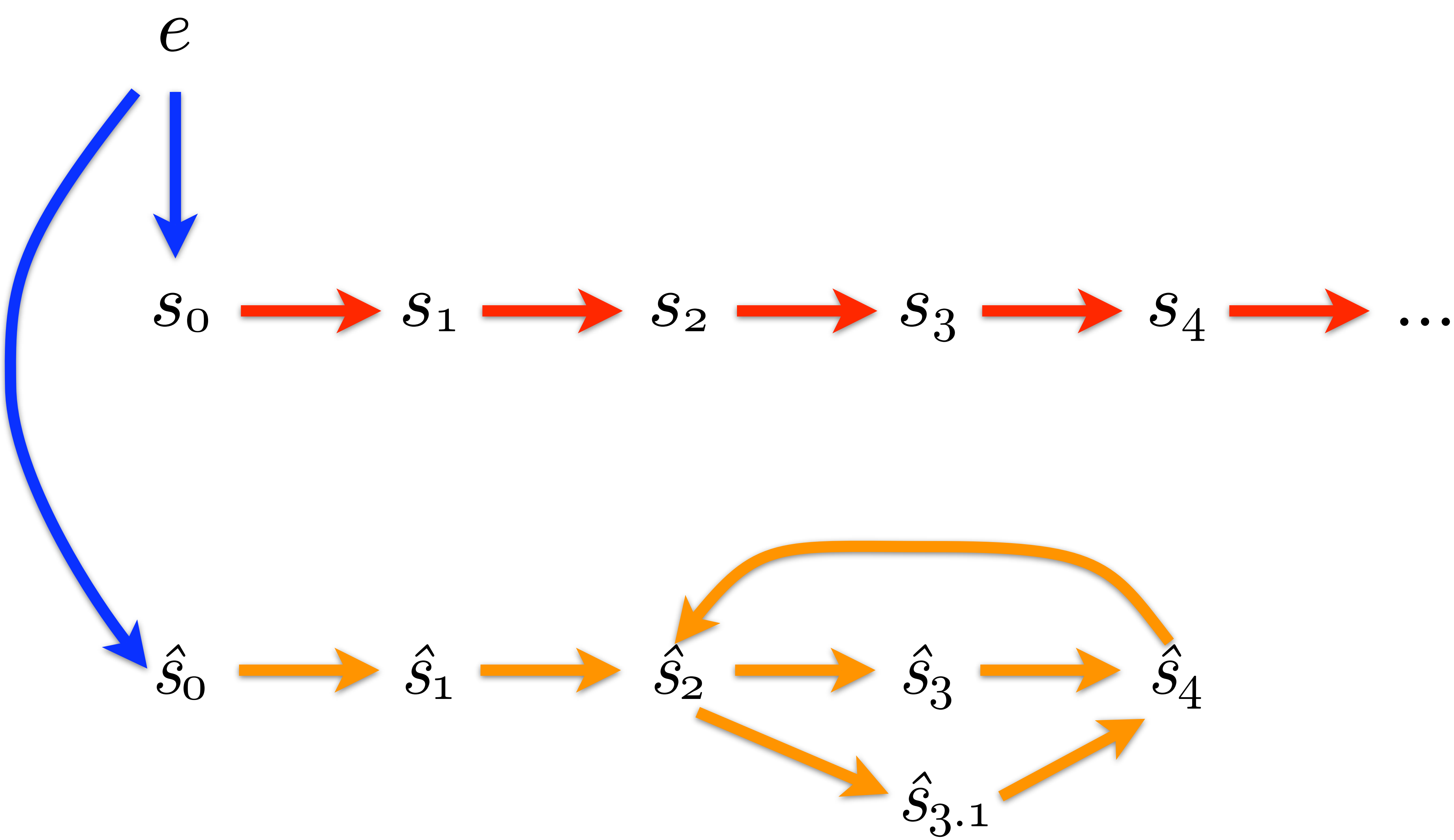
Abstract machine



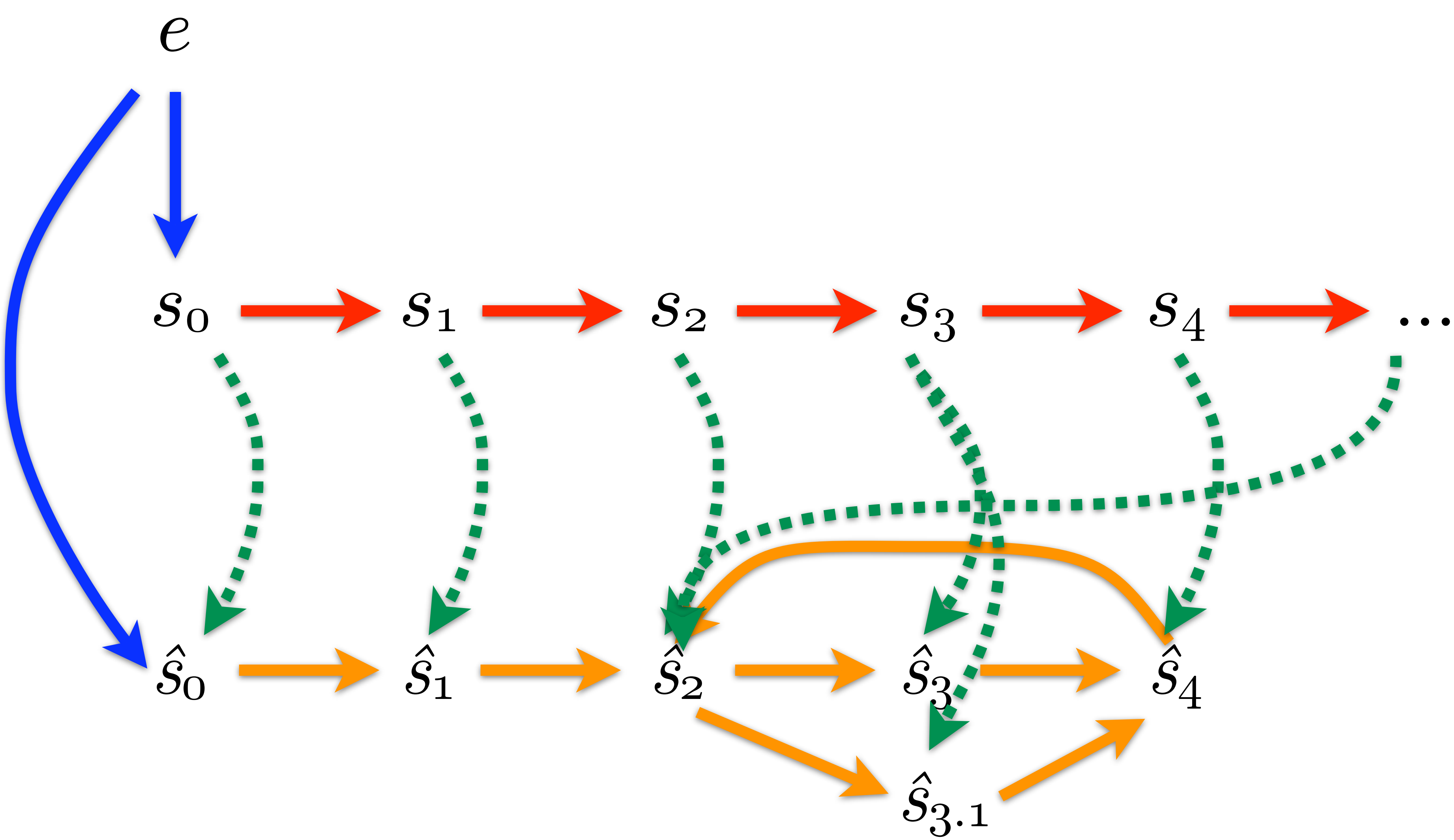
Abstract machine



Abstract machine



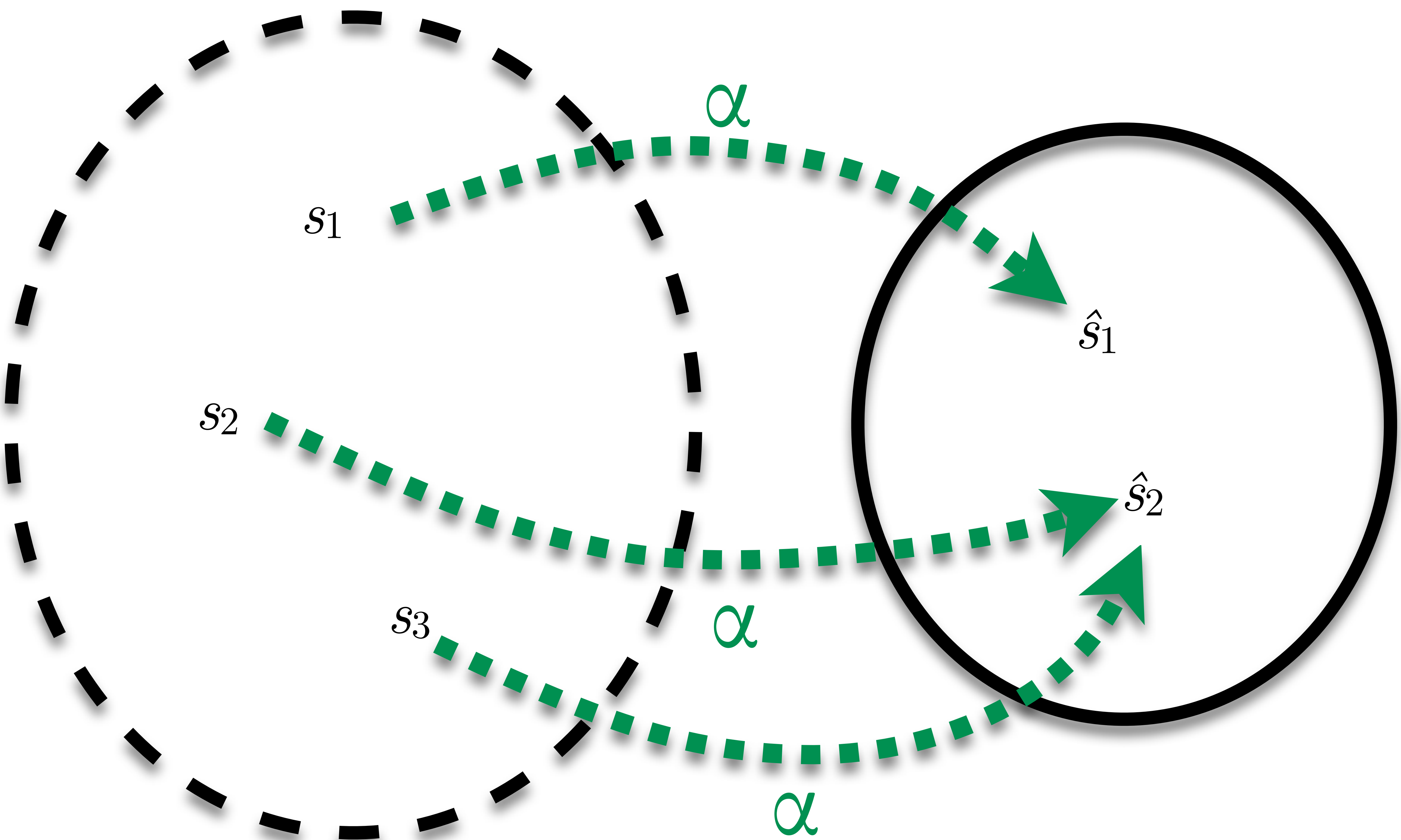
Abstract machine



Theorem: The abstract simulates the concrete.

Abstraction

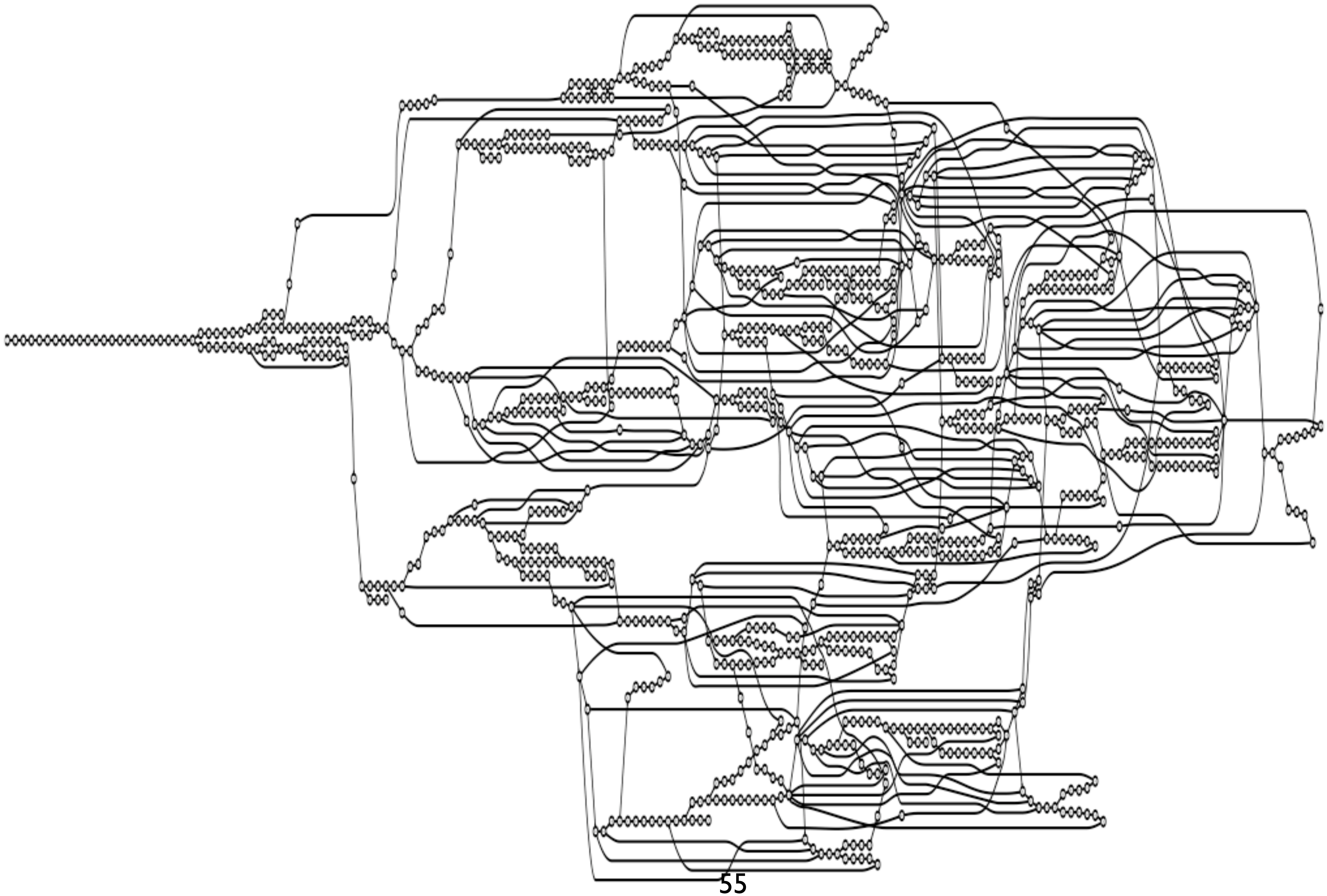
Abstraction



Example: Abstract graph

```
(letrec ((lp1 (λ (i x)
                (if (= 0 i)  x
                    (letrec ((lp2 (λ (j f y) (if (= 0 j)
                                                  (lp1 (- i 1) y)
                                                  (lp2 (- j 1) f
                                                       (f y))))))
                      (lp2 10 (λ (n) (+ n i)) x))))))
  (lp1 10 0))
```

Example: Abstract graph



Small-step CFA for CPS

Continuation-passing style

$$v \in \mathsf{Var}$$

$$f, e \in \mathsf{Exp} = \mathsf{Var} + \mathsf{Lam}$$

$$lam \in \mathsf{Lam} ::= (\lambda\ (v_1 \ldots v_n)\ call)$$

$$call \in \mathsf{Call} ::= (f\ e_1 \ldots e_n)$$

Concrete state-space

$$\varsigma \in \Sigma = \mathsf{Call} \times BEnv \times Store \times Time$$

$$\beta \in BEnv = \mathsf{Var} \rightharpoonup Addr$$

$$\sigma \in Store = Addr \rightharpoonup Clo$$

$$clo \in Clo = \mathsf{Lam} \times BEnv$$

$a \in Addr$ is a set of addresses

$t \in Time$ is a set of time-stamps

Simpler option

$$\varsigma \in \Sigma = \mathsf{Call} \times Env$$

$$\rho \in Env = \mathsf{Var} \rightharpoonup Clo$$

$$clo \in Clo = \mathsf{Lam} \times Env$$

Concrete state-space

$$\varsigma \in \Sigma = \mathsf{Call} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{Time}$$

$$\beta \in \mathit{BEnv} = \mathsf{Var} \rightharpoonup \mathit{Addr}$$

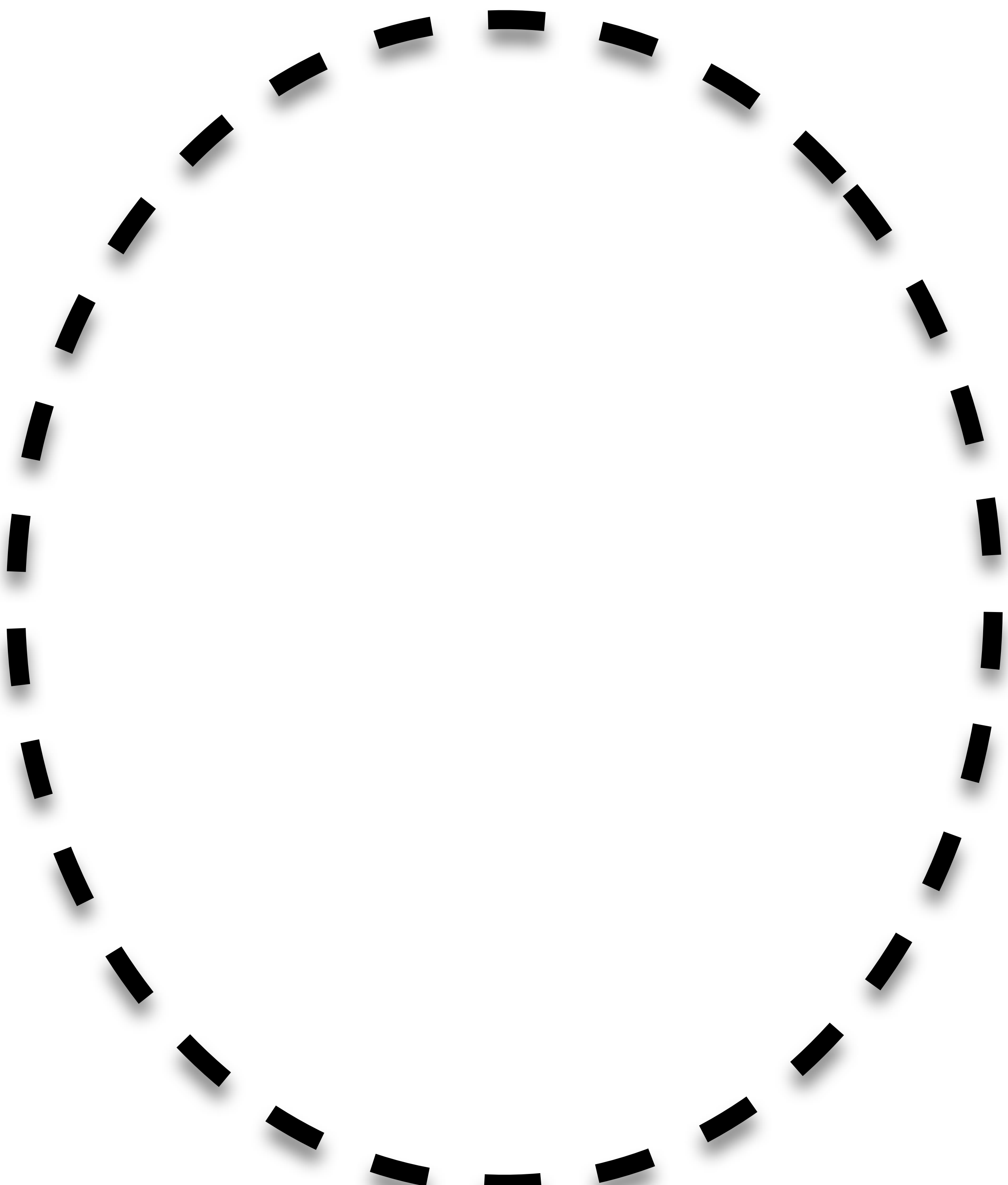
$$\sigma \in \mathit{Store} = \mathit{Addr} \rightharpoonup \mathit{Clo}$$

$$\mathit{clo} \in \mathit{Clo} = \mathsf{Lam} \times \mathit{BEnv}$$

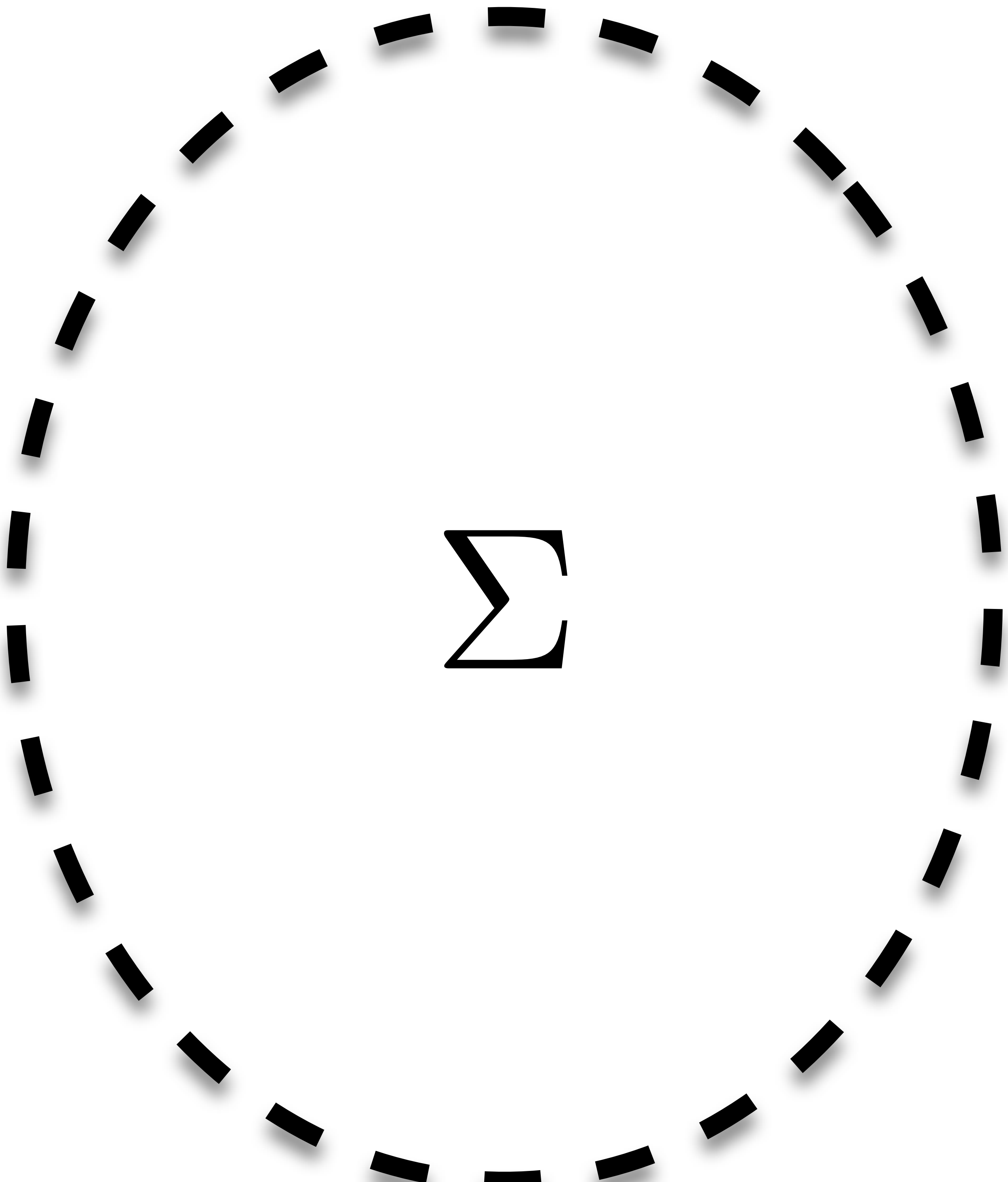
$a \in \mathit{Addr}$ is a set of addresses

$t \in \mathit{Time}$ is a set of time-stamps

State-spaces



State-spaces

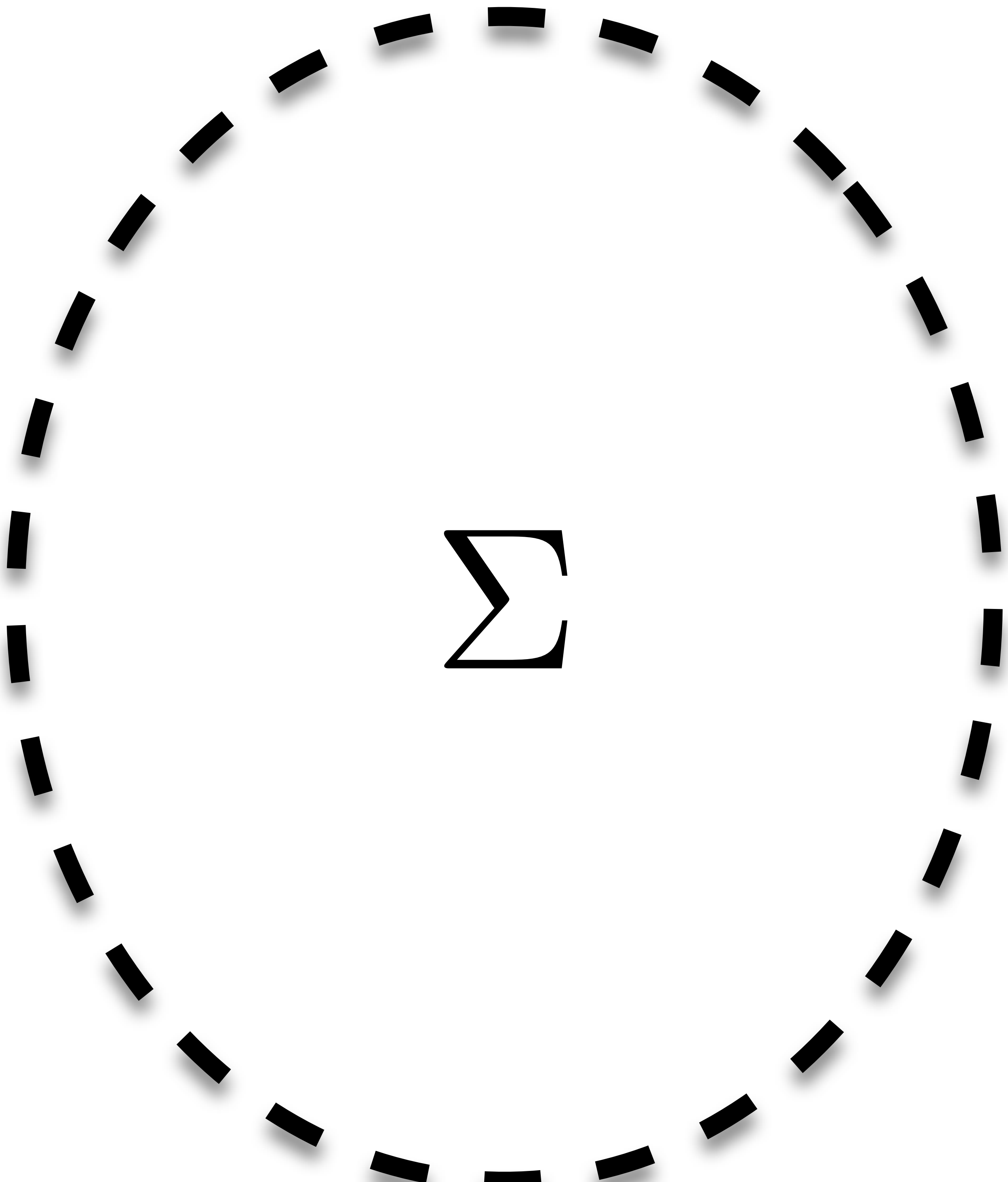


Injector

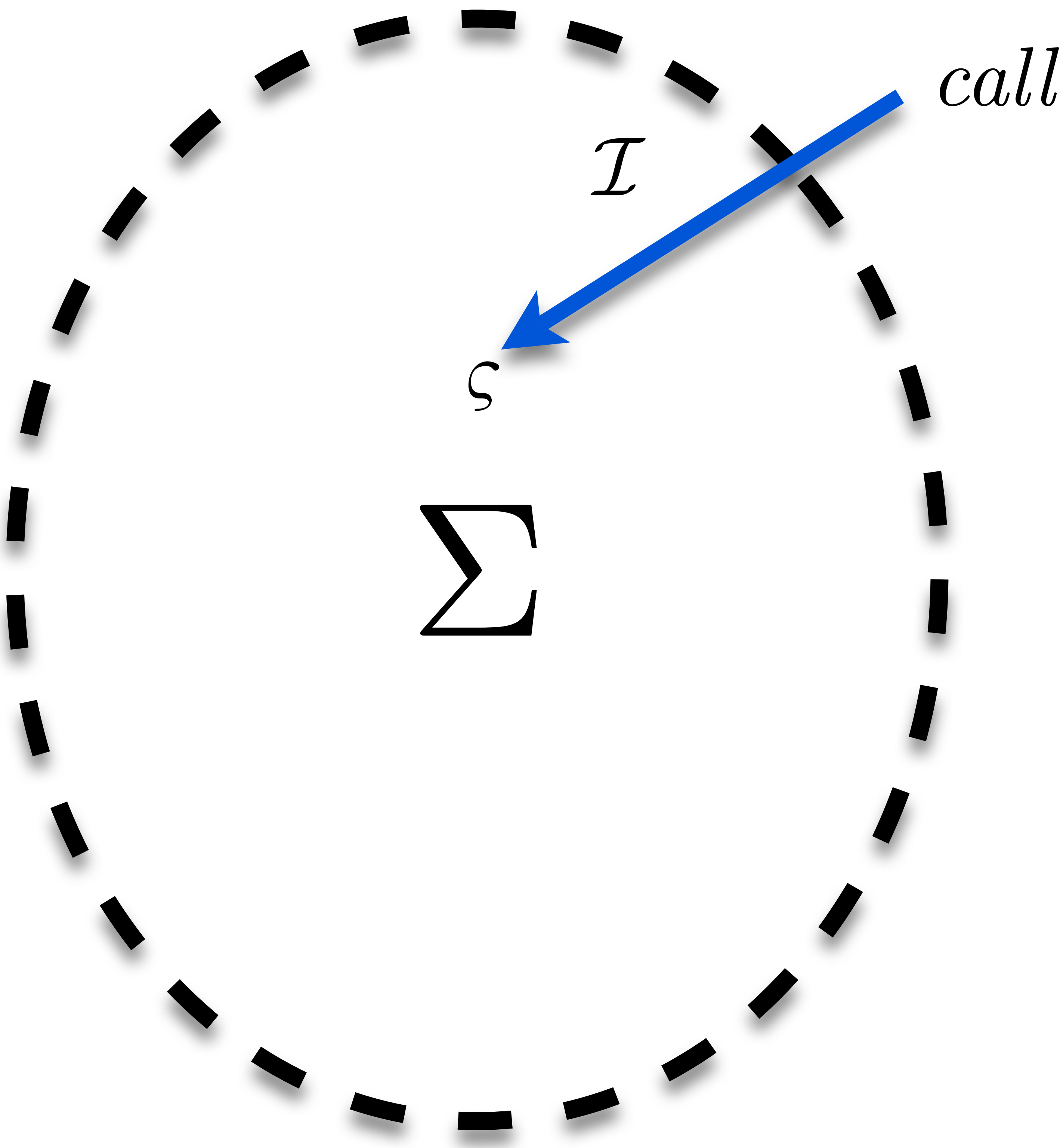
$$\mathcal{I} : \mathsf{Call} \rightarrow \Sigma$$

$$\mathcal{I}(call) = (call, [], [], t_0)$$

State-spaces



State-spaces



Factored evaluator

$$\mathcal{E}(v, \beta, \sigma) = \sigma(\beta(v))$$
$$\mathcal{E}(lam, \beta, \sigma) = (lam, \beta)$$

Concrete semantics

When $call = [\![(f\ e_1 \ldots e_n)]\!]$:

$$
\begin{aligned}
(call, \beta, \sigma, t) &\Rightarrow (call', \beta'', \sigma', t'), \text{ where} \\
(lam, \beta') &= \mathcal{E}(f, \beta, \sigma) \\
clo_i &= \mathcal{E}(e_i, \beta, \sigma) \\
lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call')]\!] \\
t' &= tick(call, t) \\
a_i &= alloc(v_i, t') \\
\beta'' &= \beta'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto clo_i]
\end{aligned}
$$

Concrete semantics

When $call = [\![(f\ e_1 \ldots e_n)]\!]$:

$$
\begin{aligned}
(call, \beta, \sigma, t) &\Rightarrow (call', \beta'', \sigma', t'), \text{ where} \\
(lam, \beta') &= \mathcal{E}(f, \beta, \sigma) \\
clo_i &= \mathcal{E}(e_i, \beta, \sigma) \\
lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call')]\!] \\
t' &= \boxed{tick(call, t)} \\
a_i &= \boxed{alloc(v_i, t')} \\
\beta'' &= \beta'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto clo_i]
\end{aligned}
$$

The easy solution

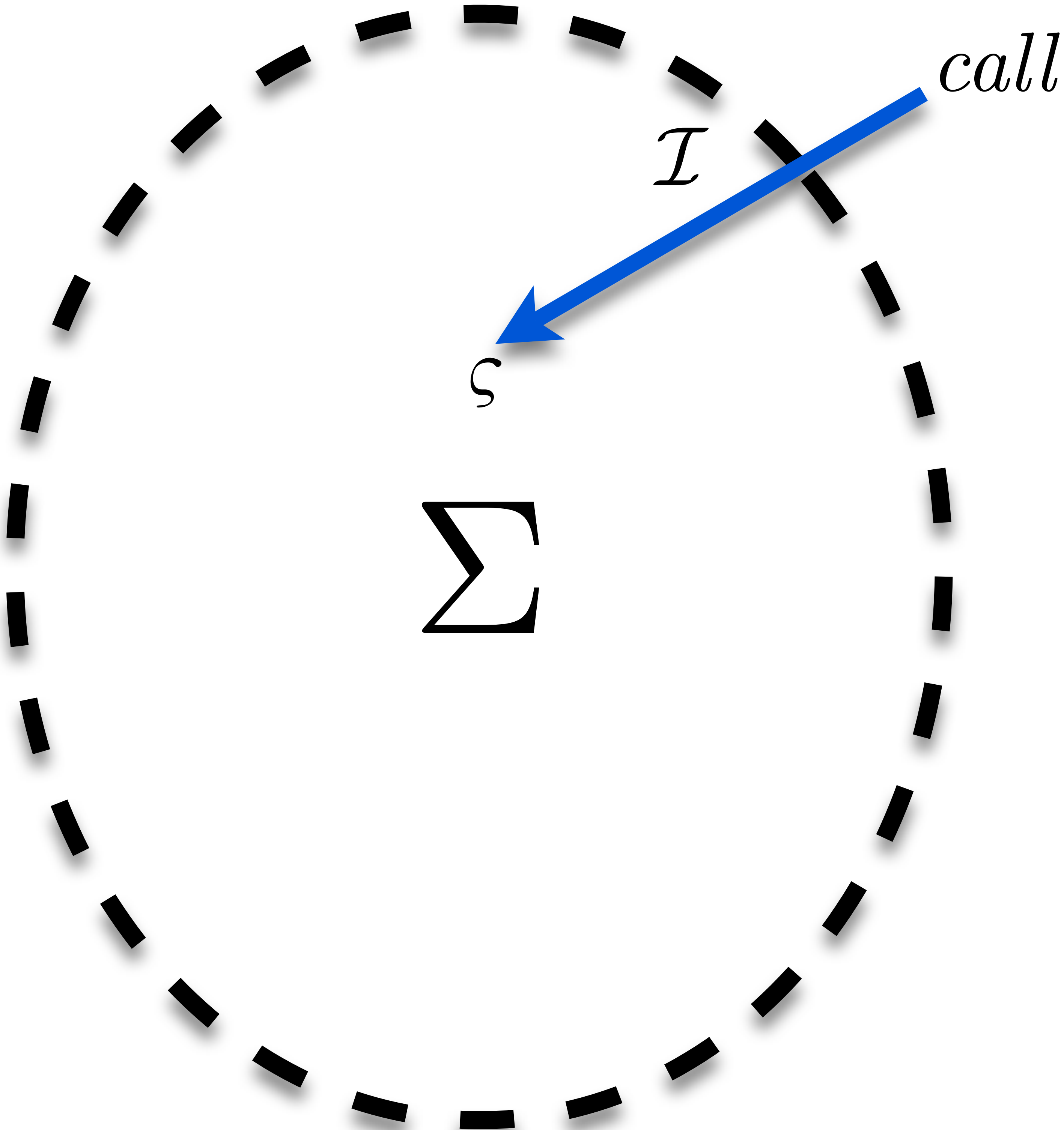
$$Time = \mathbb{N}$$

$$Addr = \mathsf{Var} \times Time$$

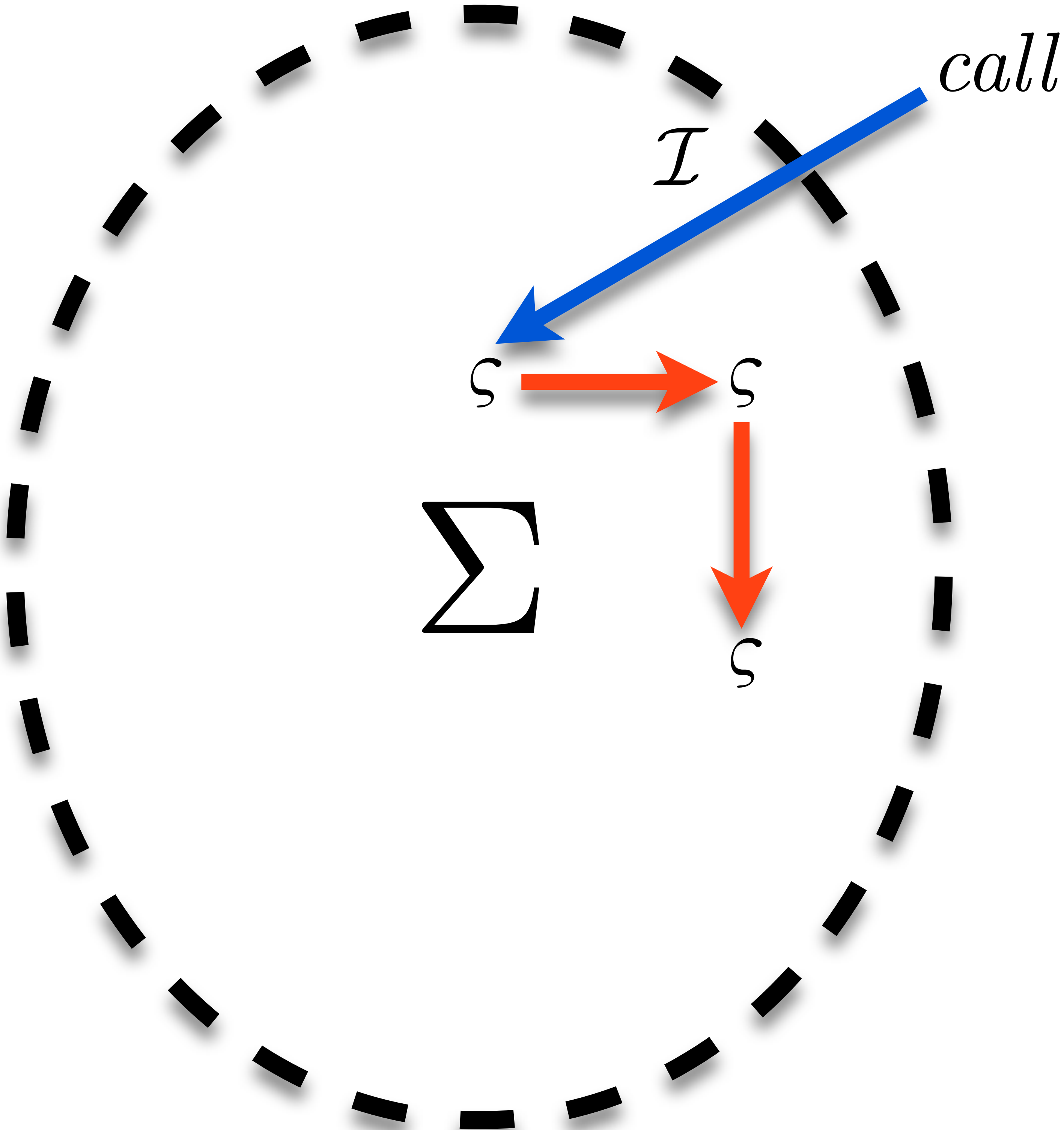
$$tick(\_, t) = t + 1$$

$$alloc(v, t) = (v, t)$$

State-spaces



State-spaces



Abstracting into a control-flow analysis

Abstract state-space

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{Time}$$

$$\hat{\beta} \in \widehat{BEnv} = \mathsf{Var} \rightarrow \widehat{Addr}$$

$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightarrow \mathcal{P}\left(\widehat{Clo}\right)$$

$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{BEnv}$$

$\hat{a} \in \widehat{Addr}$ is a **finite** set of addresses

$\hat{t} \in \widehat{Time}$ is a **finite** set of time-stamps

Abstract state-space

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{Time}$$
$$\hat{\beta} \in \widehat{BEnv} = \mathsf{Var} \to \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right)$$
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{BEnv}$$

$\hat{a} \in \widehat{Addr}$ is a **finite** set of addresses

$\hat{t} \in \widehat{Time}$ is a **finite** set of time-stamps

$$\varsigma \in \Sigma = \mathsf{Call} \times BEnv \times Store \times Time$$
$$\beta \in BEnv = \mathsf{Var} \rightharpoonup Addr$$
$$\sigma \in Store = Addr \rightharpoonup Clo$$
$$clo \in Clo = \mathsf{Lam} \times BEnv$$

$a \in Addr$ is a set of addresses

$t \in Time$ is a set of time-stamps

Abstract state-space

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{Time}$$
$$\hat{\beta} \in \widehat{BEnv} = \mathsf{Var} \to \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right)$$
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{BEnv}$$

$\hat{a} \in \widehat{Addr}$ is a **finite** set of addresses

$\hat{t} \in \widehat{Time}$ is a **finite** set of time-stamps

$$\varsigma \in \Sigma = \mathsf{Call} \times BEnv \times Store \times Time$$
$$\beta \in BEnv = \mathsf{Var} \rightharpoonup Addr$$
$$\sigma \in Store = Addr \rightharpoonup Clo$$
$$clo \in Clo = \mathsf{Lam} \times BEnv$$

$a \in Addr$ is a set of addresses

$t \in Time$ is a set of time-stamps

Is this state-space finite?

Non-recursive

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{Time}$$

$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right)$$

$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{BEnv}$$

$$\hat{\beta} \in \widehat{BEnv} = \mathsf{Var} \to \widehat{Addr}$$

$\hat{a} \in \widehat{Addr}$ is a **finite** set of addresses

$\hat{t} \in \widehat{Time}$ is a **finite** set of time-stamps

Non-recursive

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{Time}$$

$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right)$$

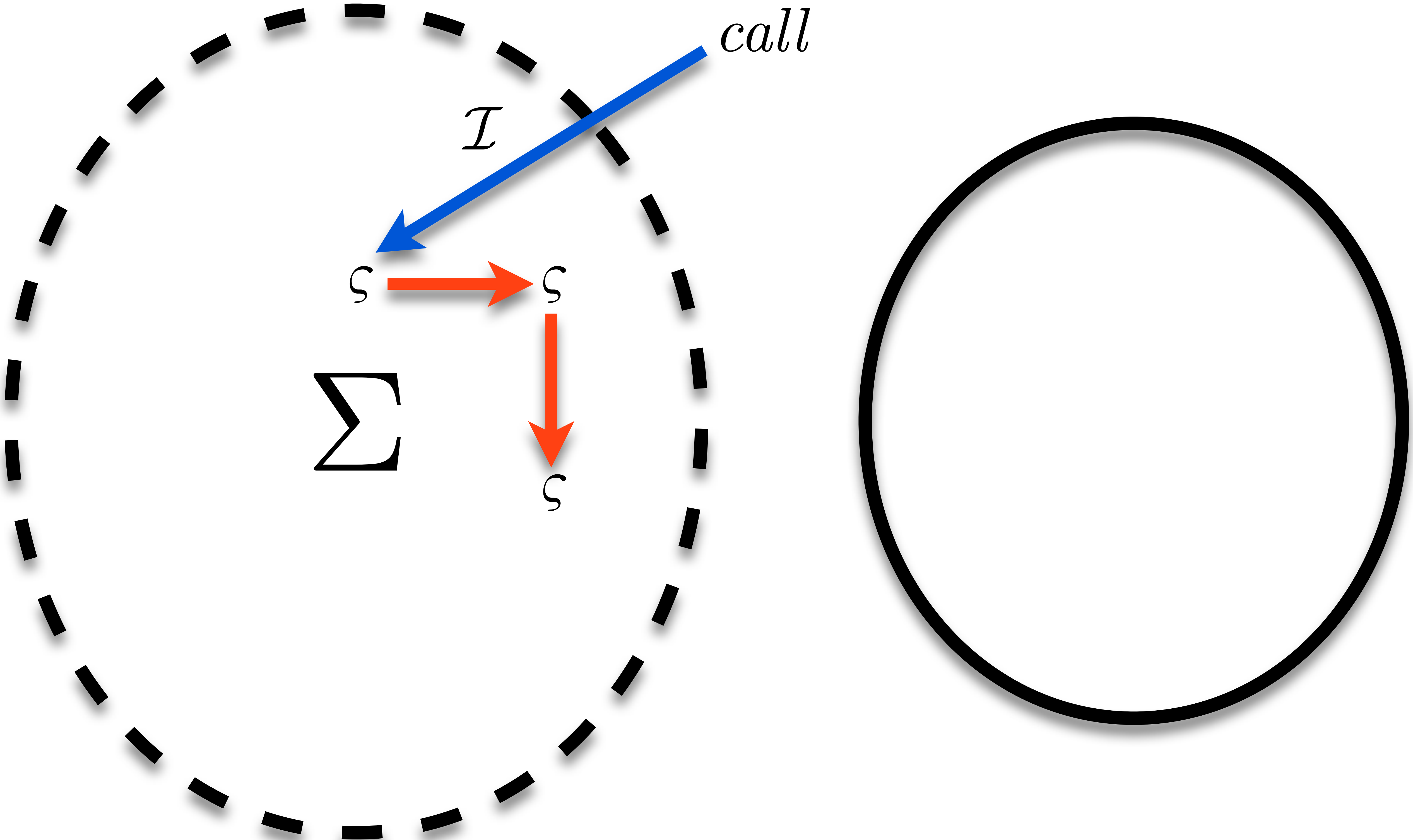
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{BEnv}$$

$$\hat{\beta} \in \widehat{BEnv} = \mathsf{Var} \to \widehat{Addr}$$

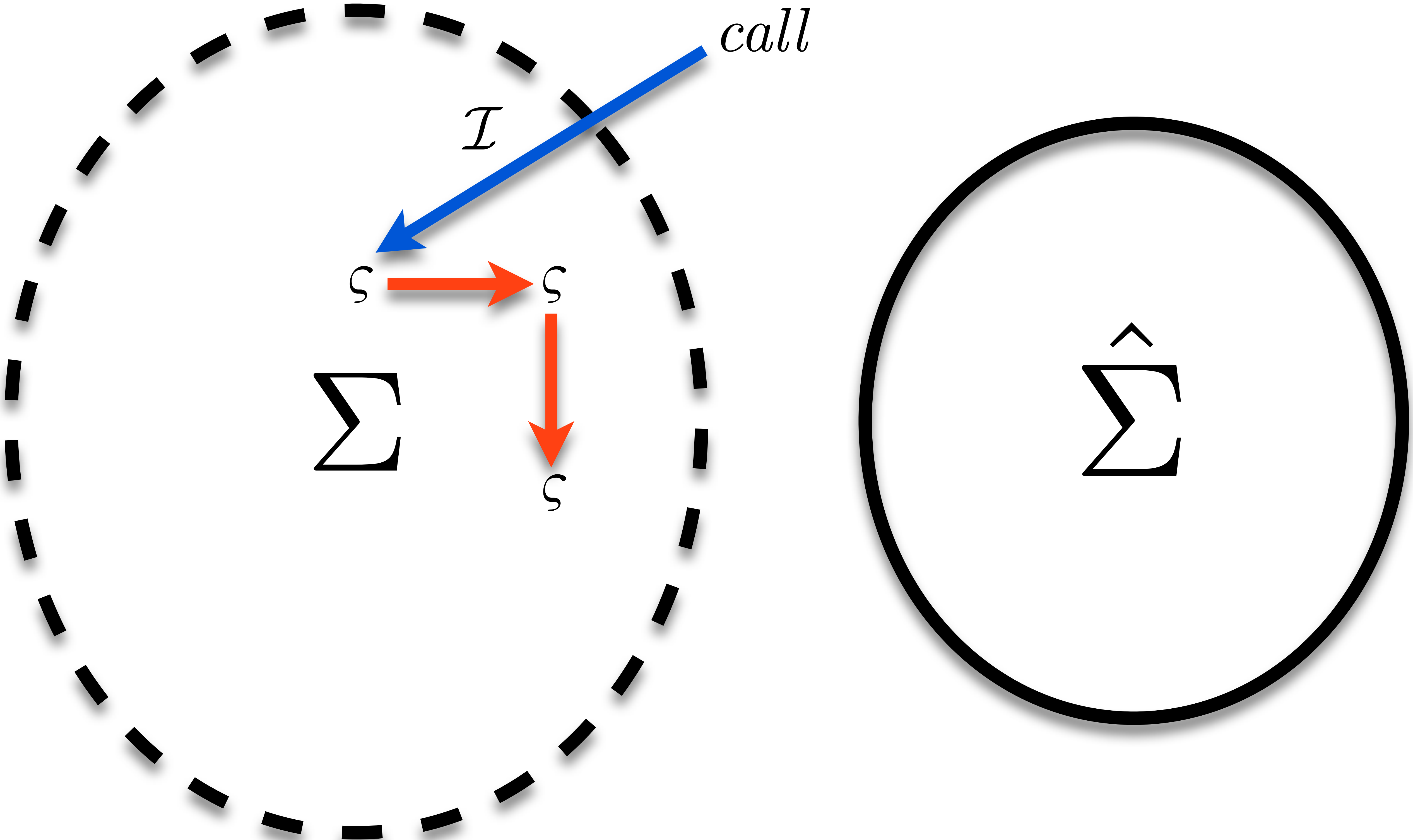
$\hat{a} \in \widehat{Addr}$ is a **finite** set of addresses

$\hat{t} \in \widehat{Time}$ is a **finite** set of time-stamps

State-spaces



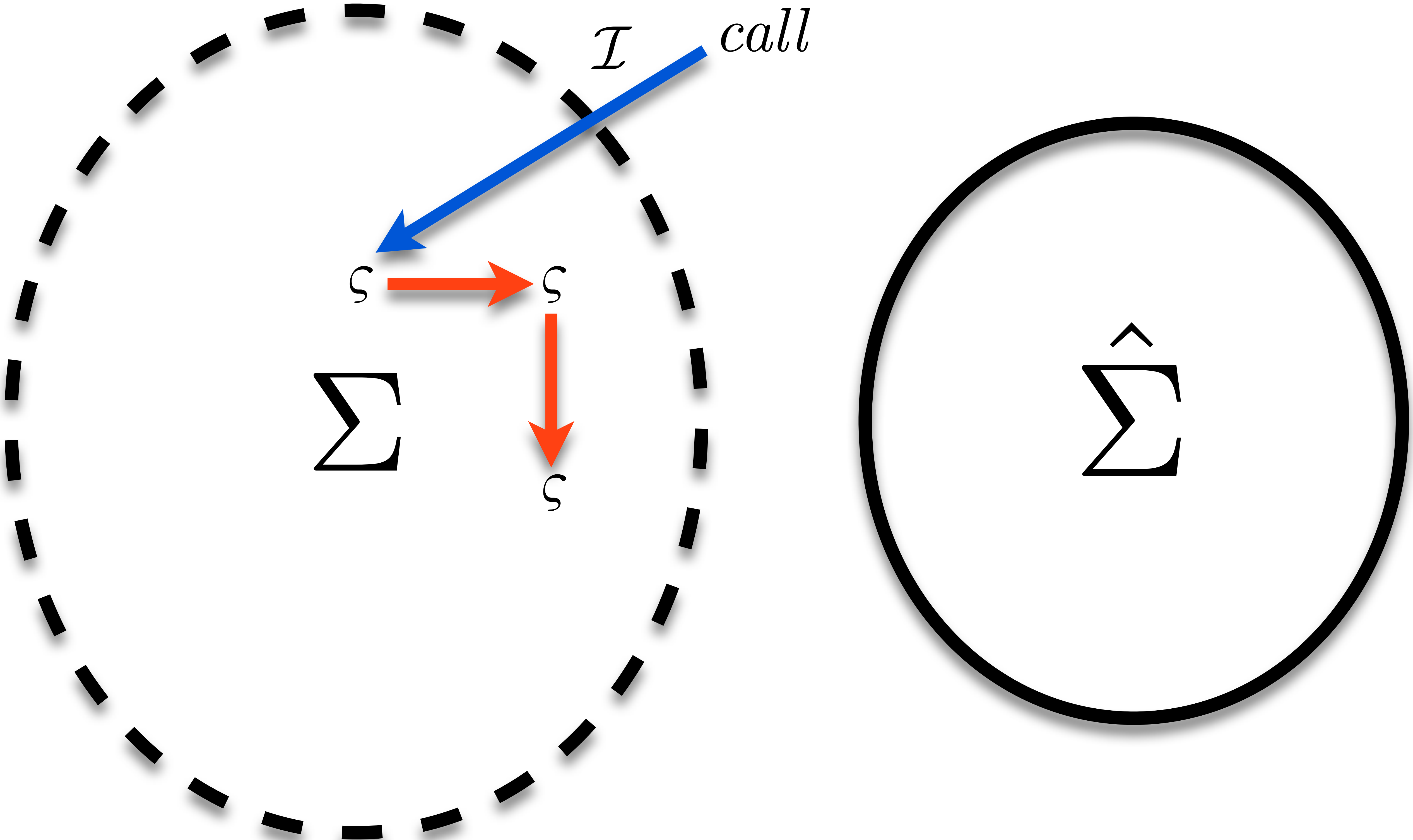
State-spaces



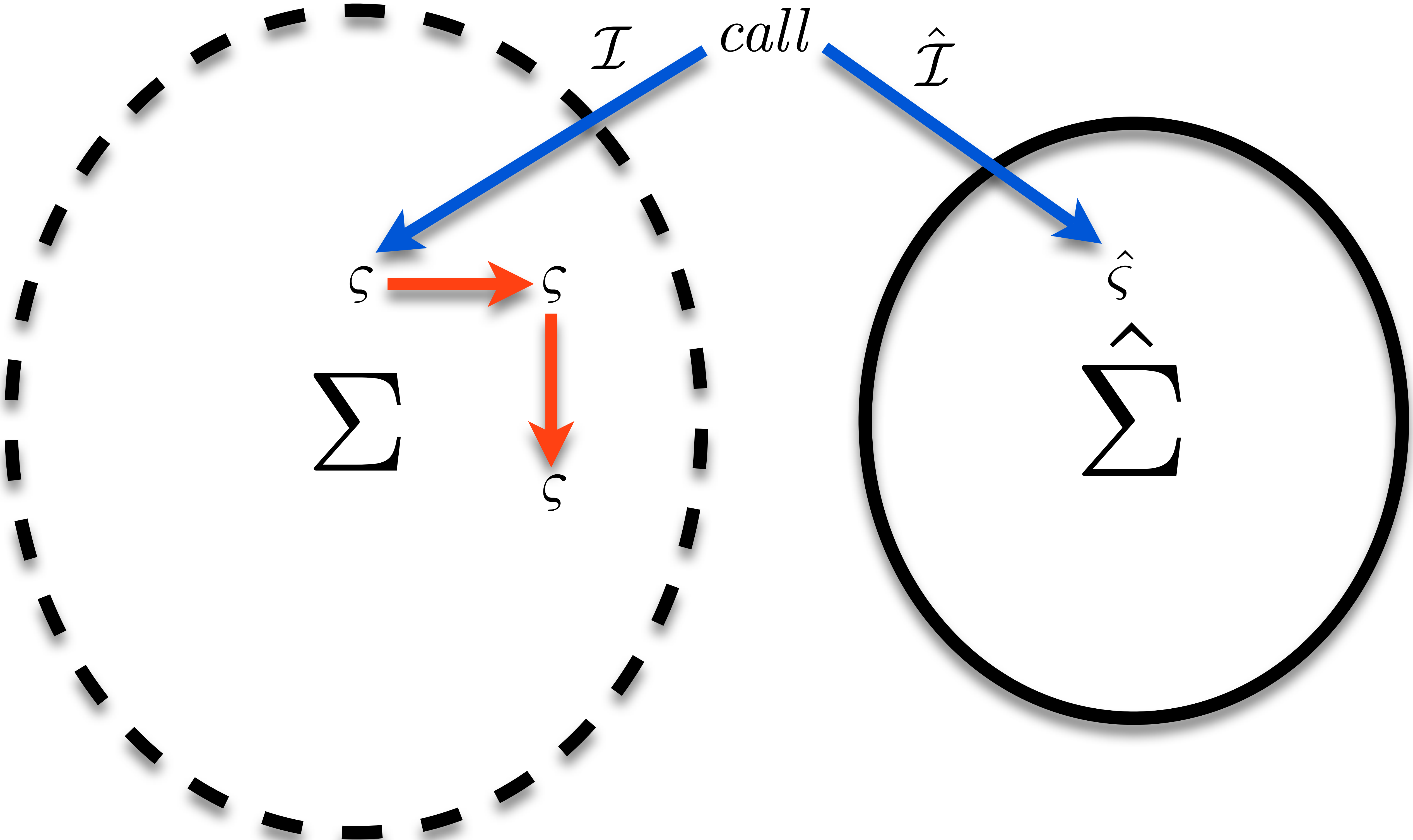
Injector

$$\hat{\mathcal{I}}(call) = (call, [], [], \hat{t}_0)$$

State-spaces



State-spaces



Abstract components

$$(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$$

$$\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{BEnv} \times \widehat{Store} \rightarrow \mathcal{P}\left(\widehat{Clo}\right)$$

$$\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{BEnv} \times \widehat{Store} \to \mathcal{P}\left(\widehat{Clo}\right)$$

$$\hat{\mathcal{E}}(v, \hat{\beta}, \hat{\sigma}) = \hat{\sigma}(\hat{\beta}(v))$$
$$\hat{\mathcal{E}}(lam, \hat{\beta}, \hat{\sigma}) = \left\{(lam, \hat{\beta})\right\}$$

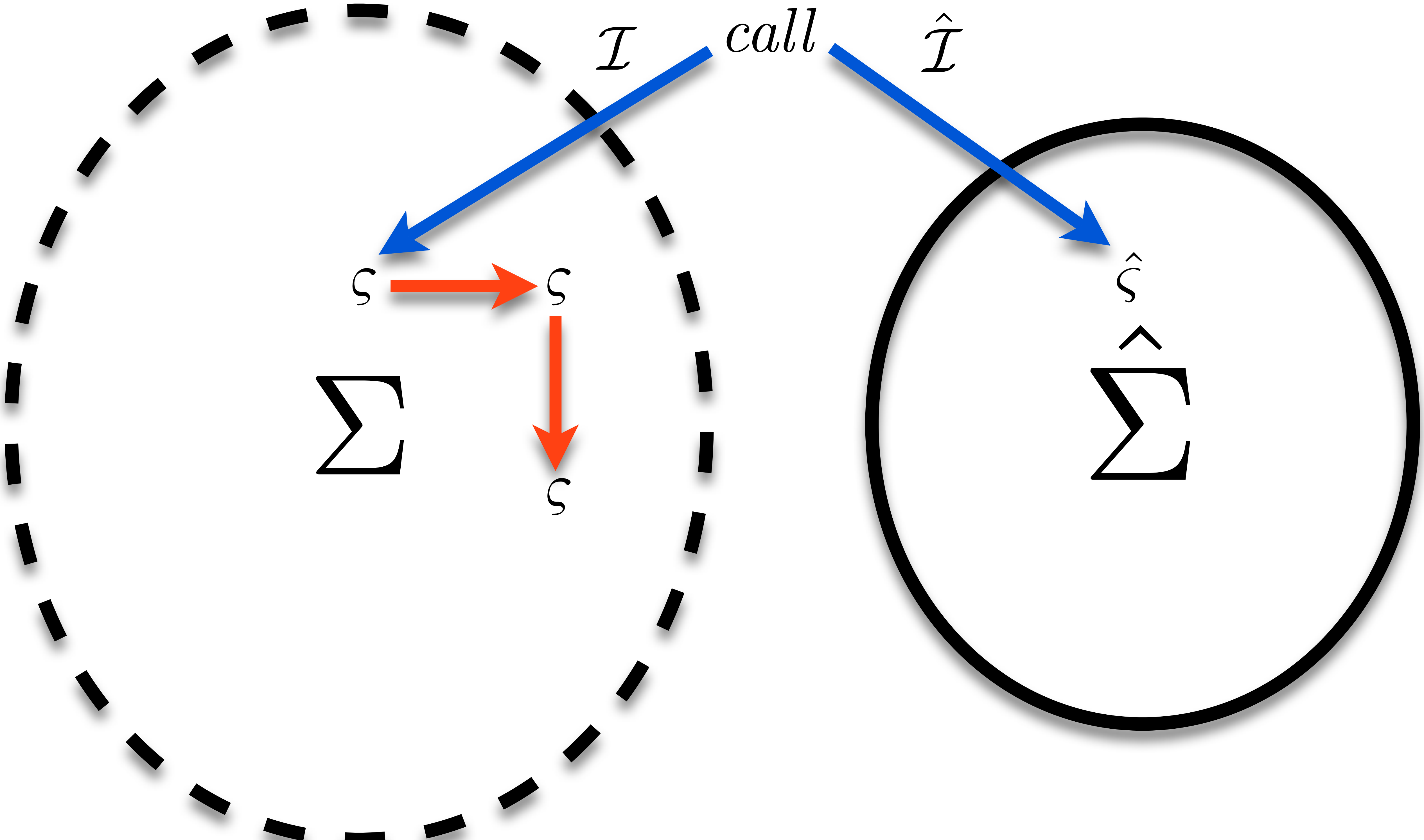
Abstract semantics

When $call = [\![\texttt{(}f\ e_1 \ldots e_n\texttt{)}]\!]$:

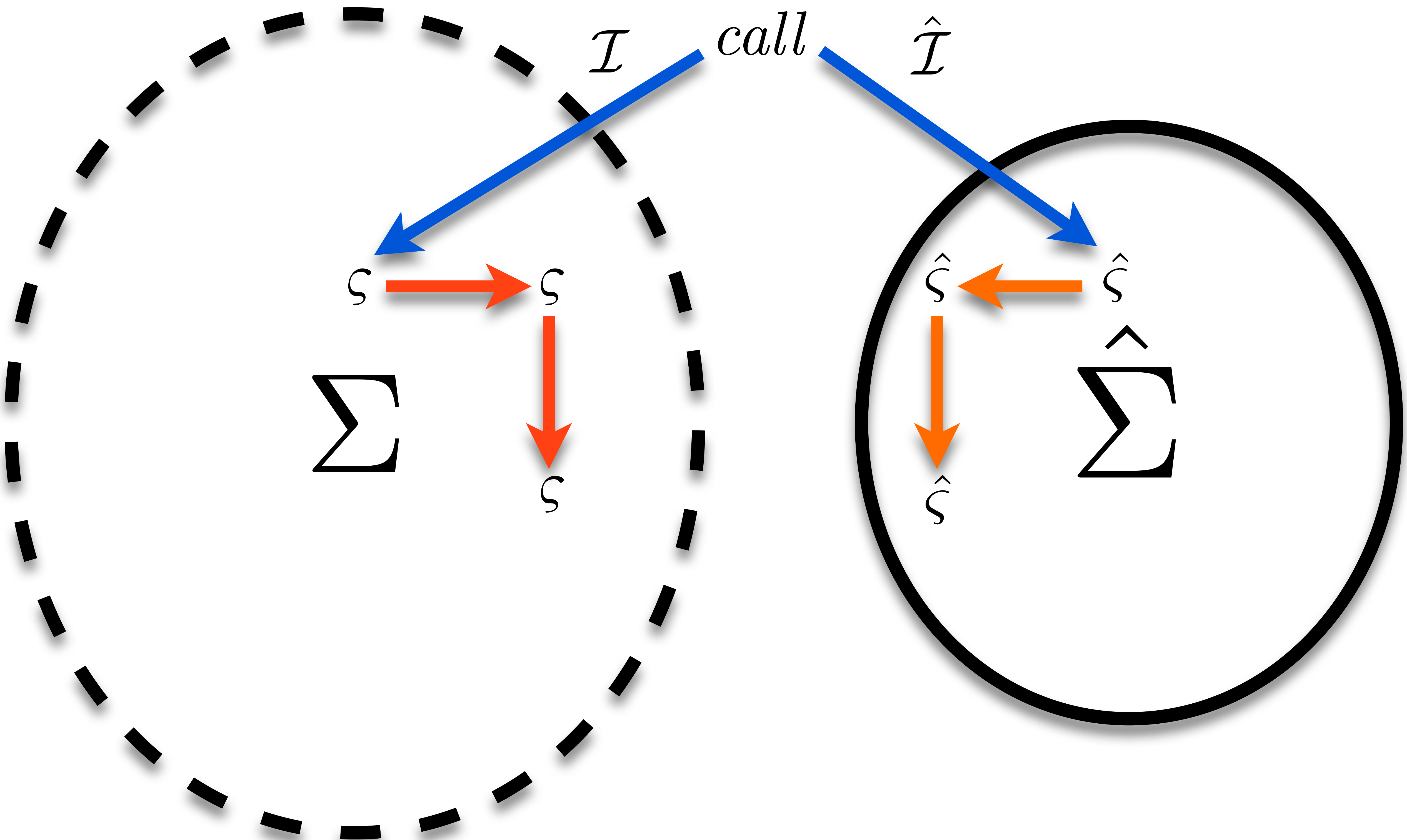
$$(call, \hat{\beta}, \hat{\sigma}, \hat{t}) \leadsto (call', \hat{\beta}'', \hat{\sigma}', \hat{t}'), \text{ where}$$

$$\begin{aligned}
(lam, \hat{\beta}') &\in \hat{\mathcal{E}}(f, \hat{\beta}, \hat{\sigma}) \\
\hat{C}_i &= \hat{\mathcal{E}}(e_i, \hat{\beta}, \hat{\sigma}) \\
lam &= [\![\texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ call'\texttt{)}]\!] \\
\hat{t}' &= \widehat{tick}(call, \hat{t}) \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{t}') \\
\hat{\beta}'' &= \hat{\beta}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{C}_i]
\end{aligned}$$

State-spaces



State-spaces



Abstraction maps

$$\alpha(call, \beta, \sigma, t) = (call, \alpha(\beta), \alpha(\sigma), \alpha(t))$$

$$\alpha(\beta) = \lambda v.\alpha(\beta(v))$$

$$\alpha(\sigma) = \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a))$$

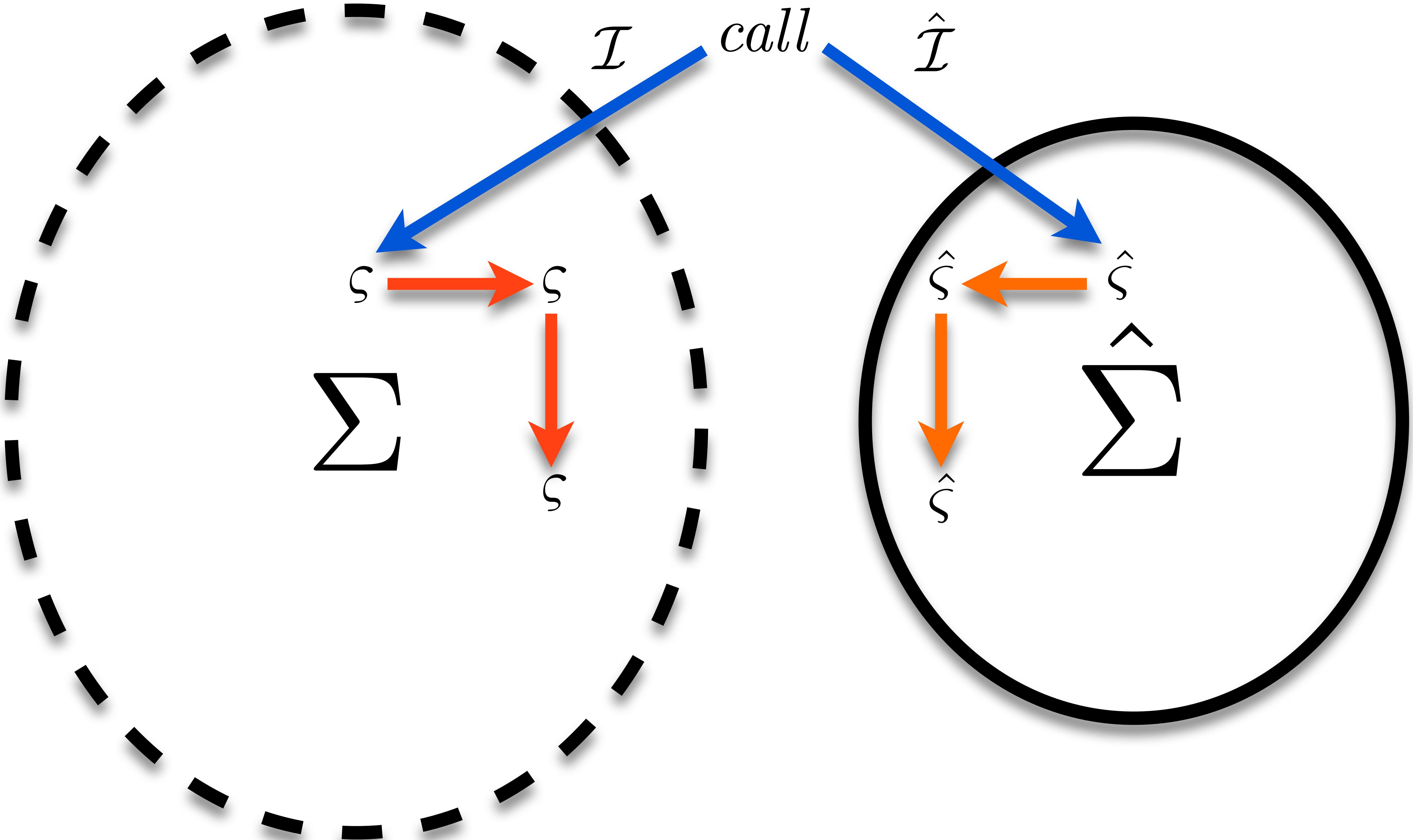
$$\alpha(lam, \beta) = \{(lam, \alpha(\beta))\}$$

$\alpha(a)$ is set by parameter

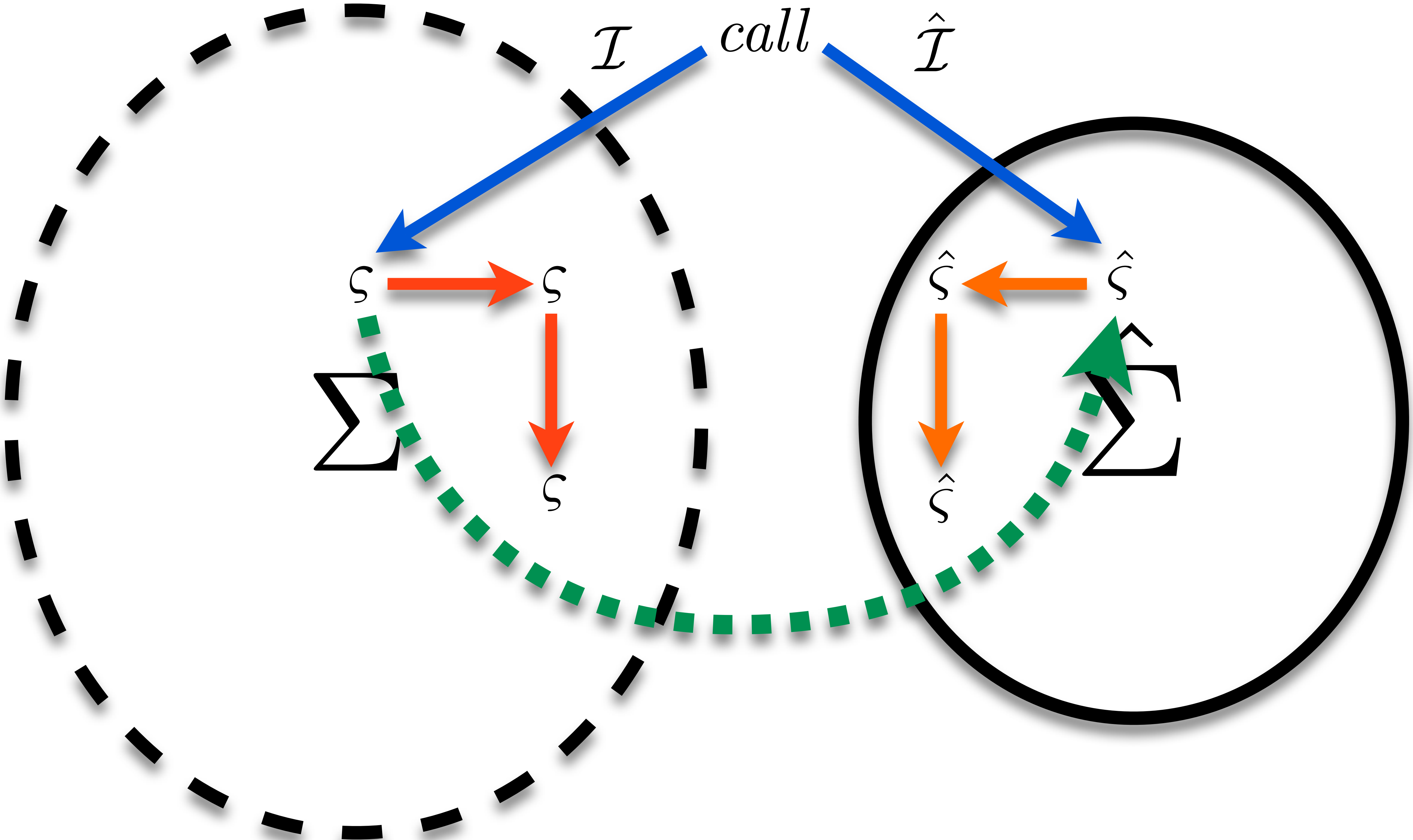
$\alpha(t)$ is set by parameter

$$\hat{\sigma} \sqcup \hat{\sigma}' = \lambda \hat{a}.(\hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a}))$$

State-spaces



State-spaces

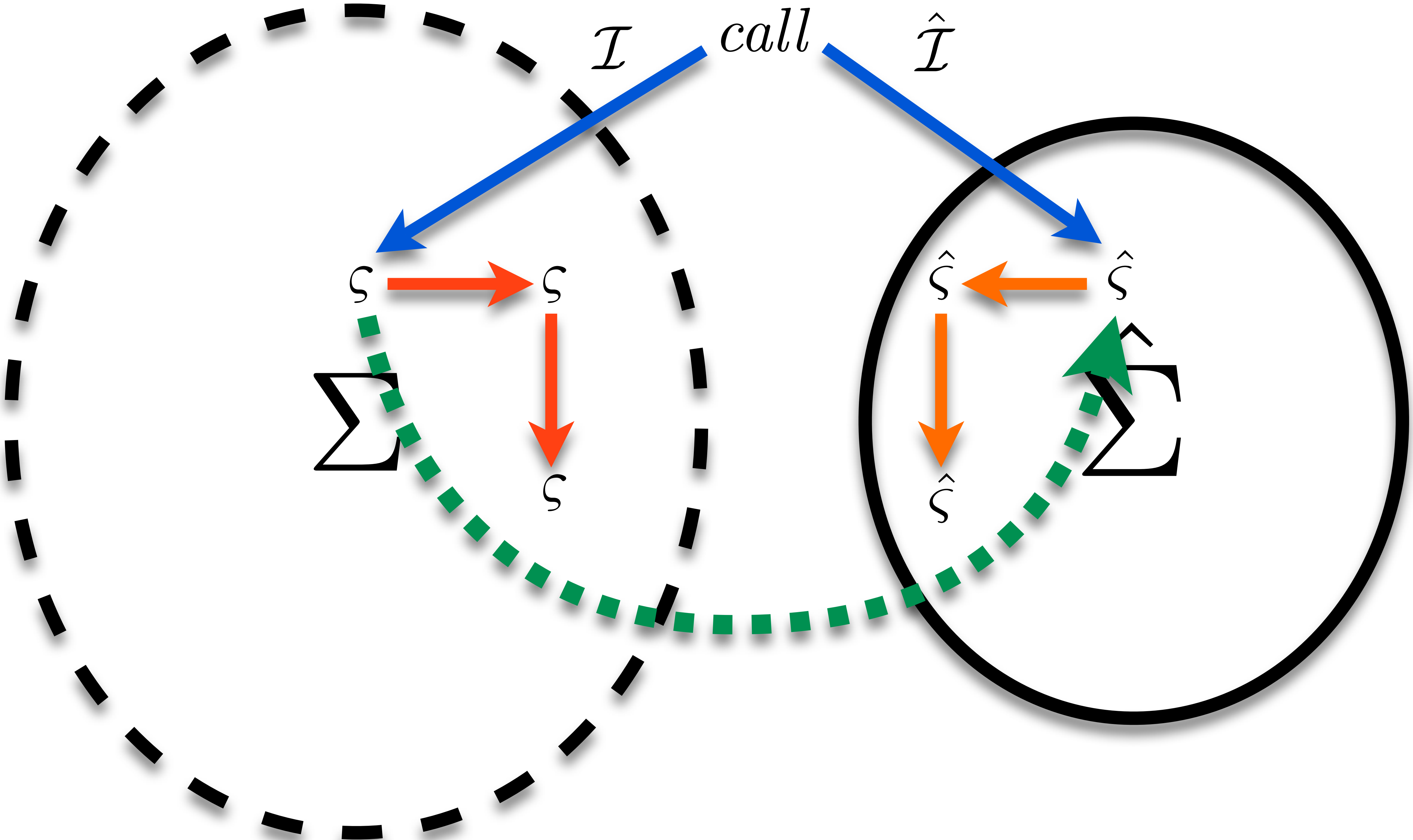


Soundness

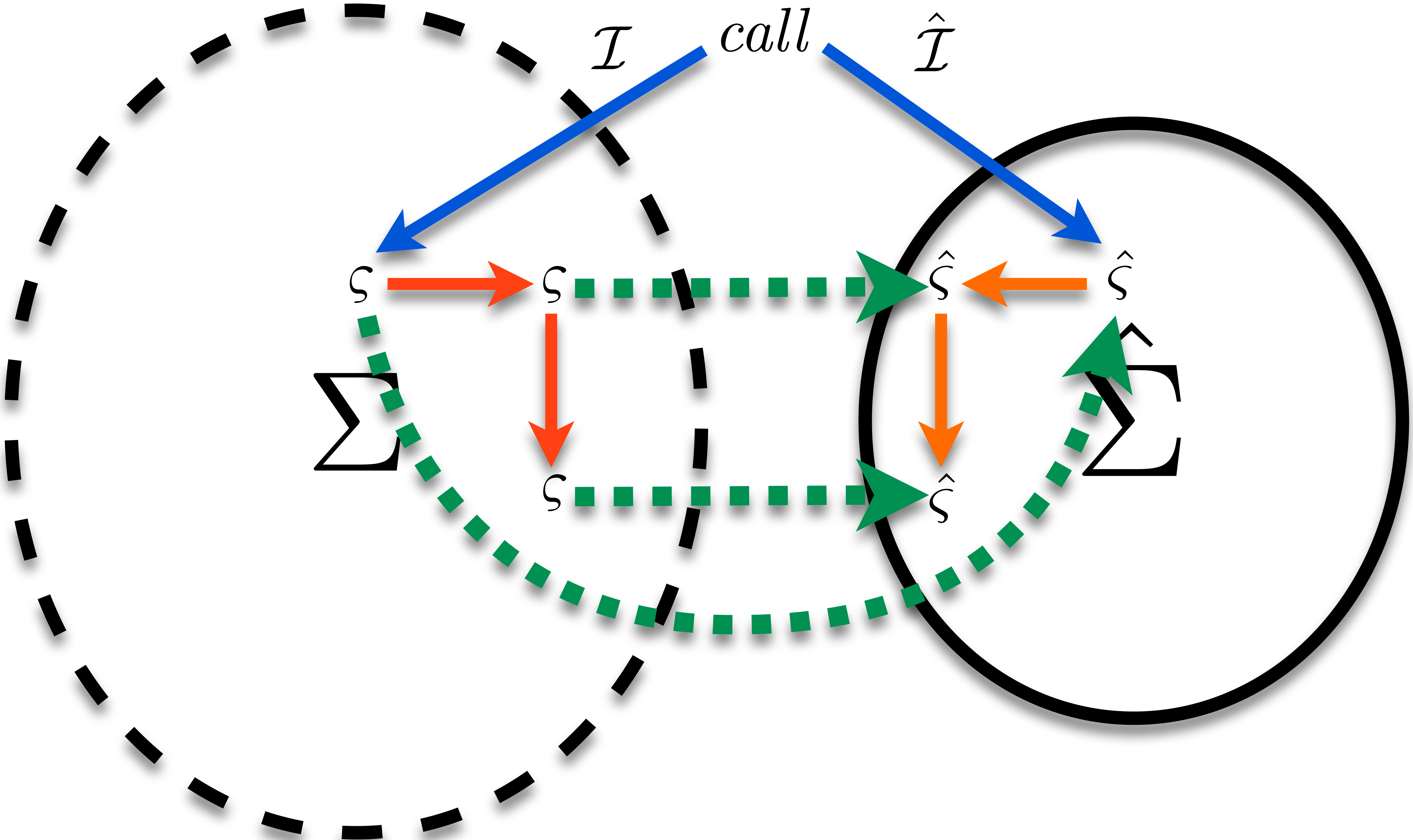
$$
\begin{tikzcd}
\varsigma \arrow[r, "\Rightarrow"] \arrow[d, "\alpha"'] & \varsigma' \arrow[d, "\alpha"] \\
{} \arrow[d, "\sqsubseteq"'] & {} \arrow[d, "\sqsubseteq"] \\
\hat{\varsigma} \arrow[r, dotted, "\leadsto"'] & \hat{\varsigma}'
\end{tikzcd}
$$

Theorem: If the concrete takes a step,
then the abstract can take a matching step.

State-spaces



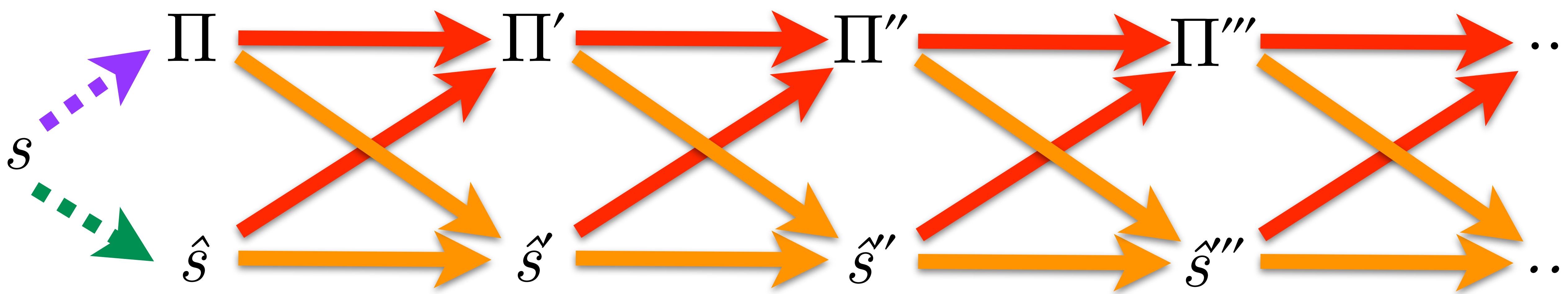
State-spaces



Application:
Buffer-overflow checks

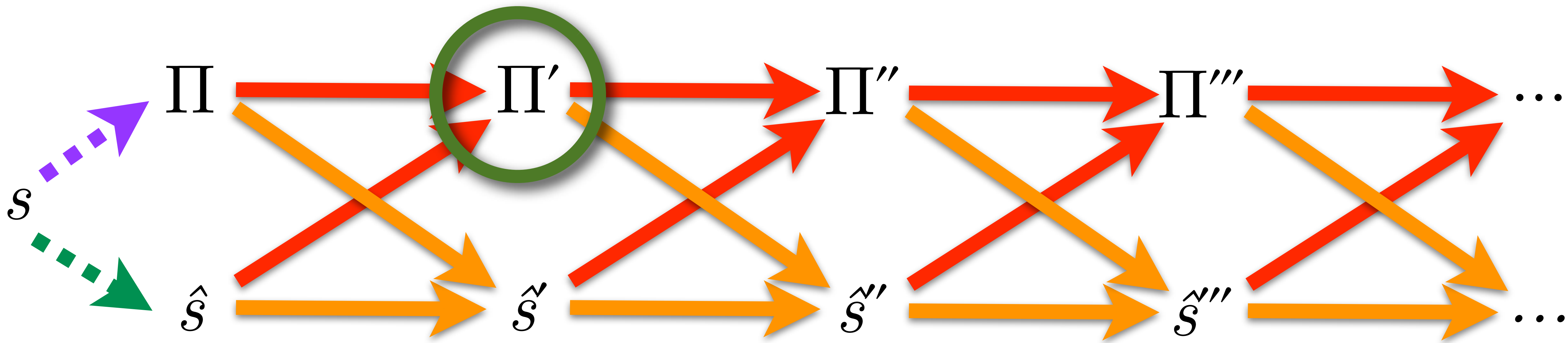
Logic-flow analysis

Logic-flow analysis



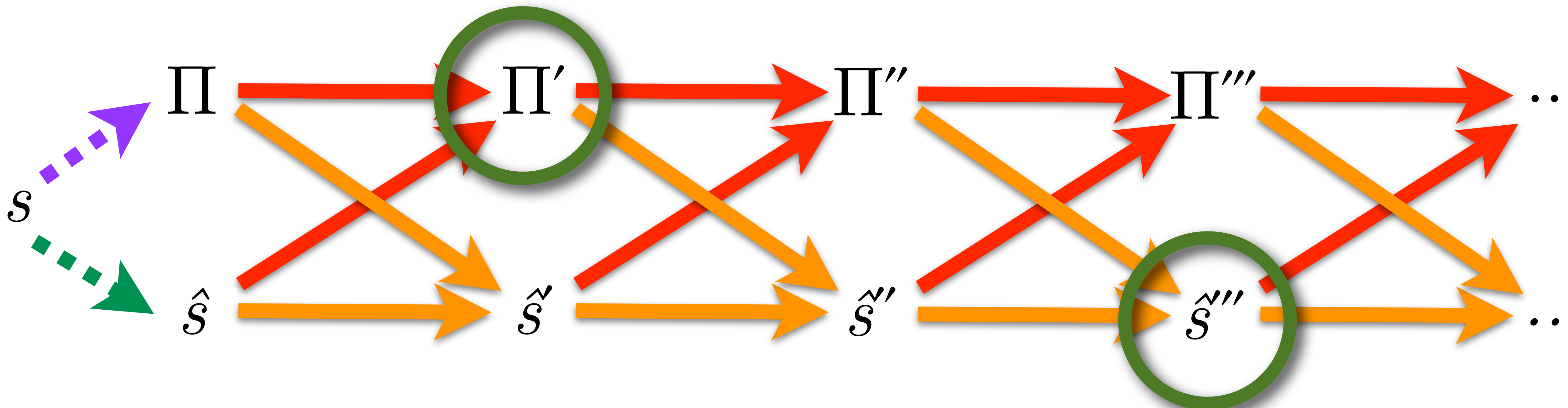
Logic-flow analysis

`i < length(a)`



Logic-flow analysis

`i < length(a)`



`a[i]`

Beyond CFA

- Abstract G.C.
- Nondeterministic A.I.
- Frame-string analysis
- Abstract counting
- Logic-flow analysis
- Dependence analysis
- See matt.might.net

Thanks!