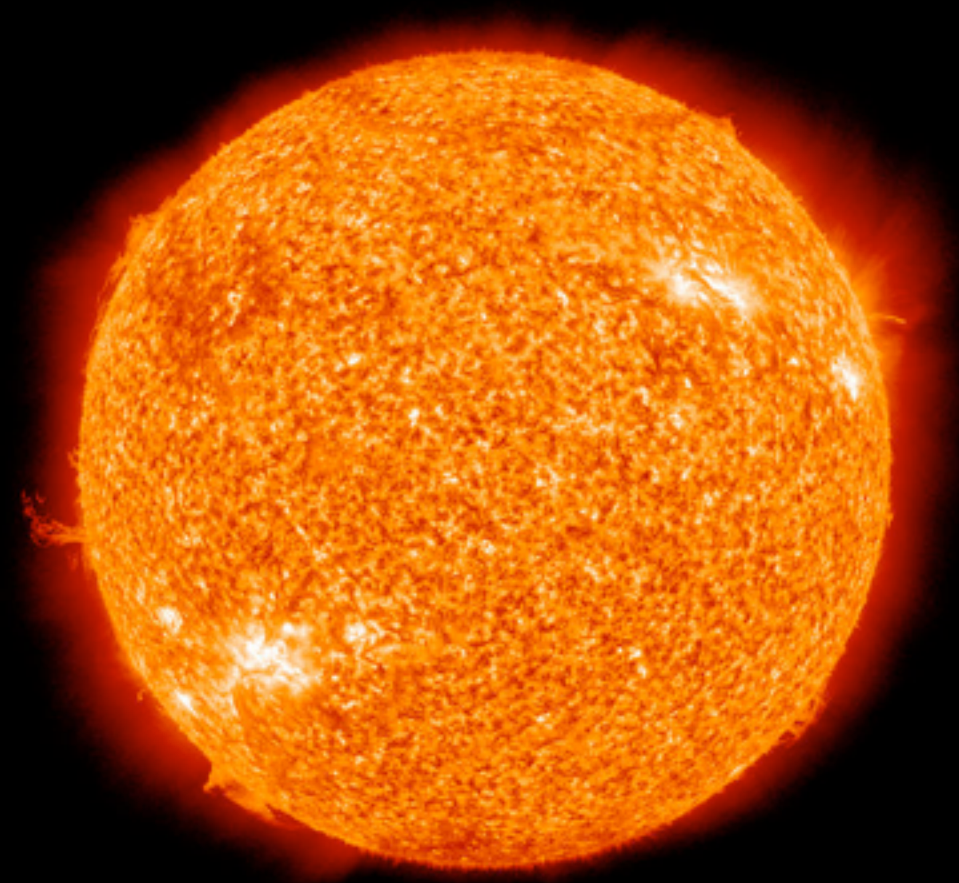
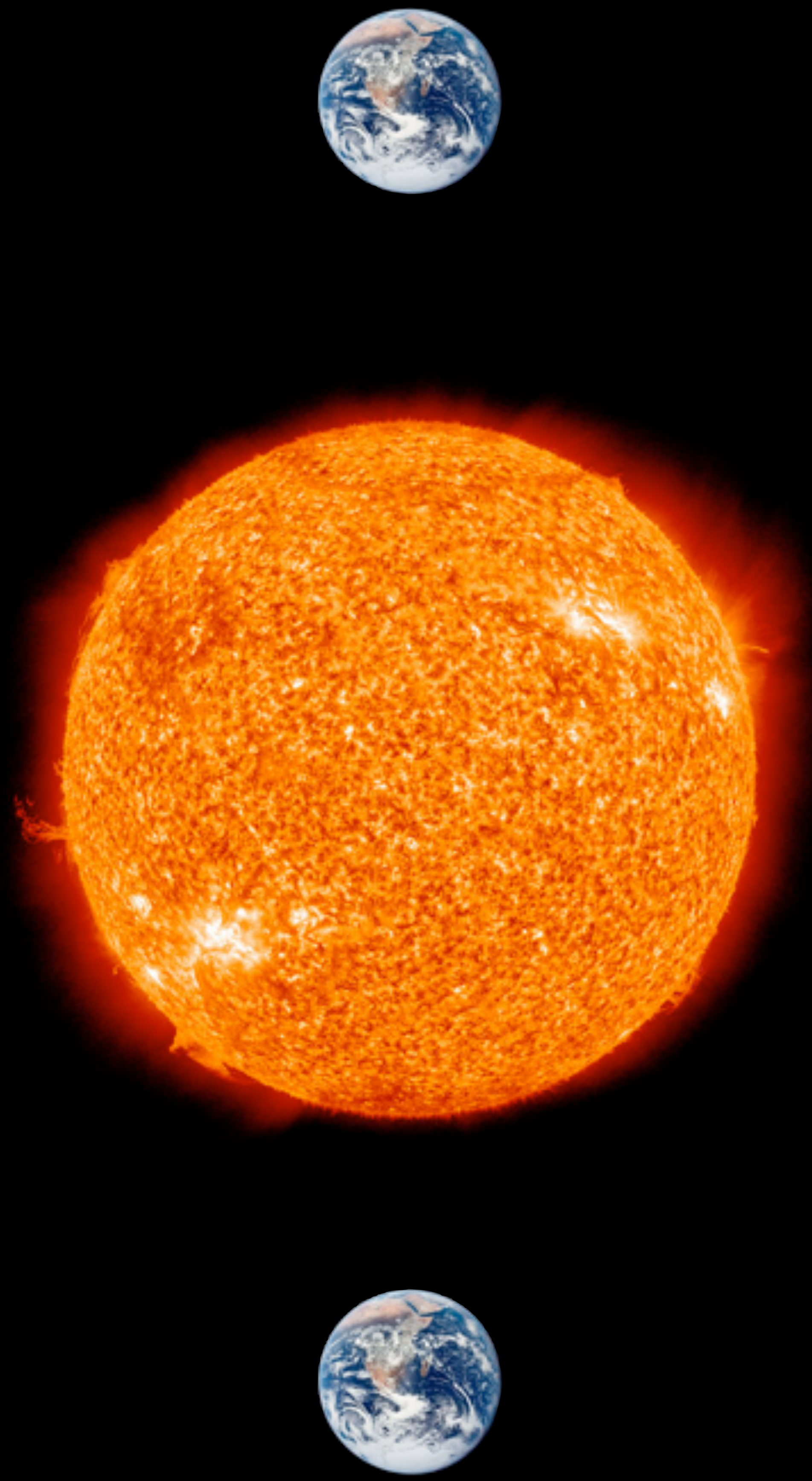
Planet CFA

Planet CFA

Planet PointsTo



Planet CFA



Control-flow Analysis of Scheme

Matt Might
University of Utah
matt.might.net

JavaScript is Lisp in C's clothing.

JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java.

Doug Crockford

The lure for Brendan Eich was that he would be able to base JavaScript on Scheme.

“JavaScript, How It All Began”

{ }

(   )

dynamic pointers

flexible (Turing-complete) syntax

lambda/higher-order functions

first-class continuations

lots of recursion

duck-typing

eval


"You merely adopted the darkness...
...I was born in it."

(Jones, 1981)

(Shivers, 1991)

What is control-flow analysis?

What is `f`?

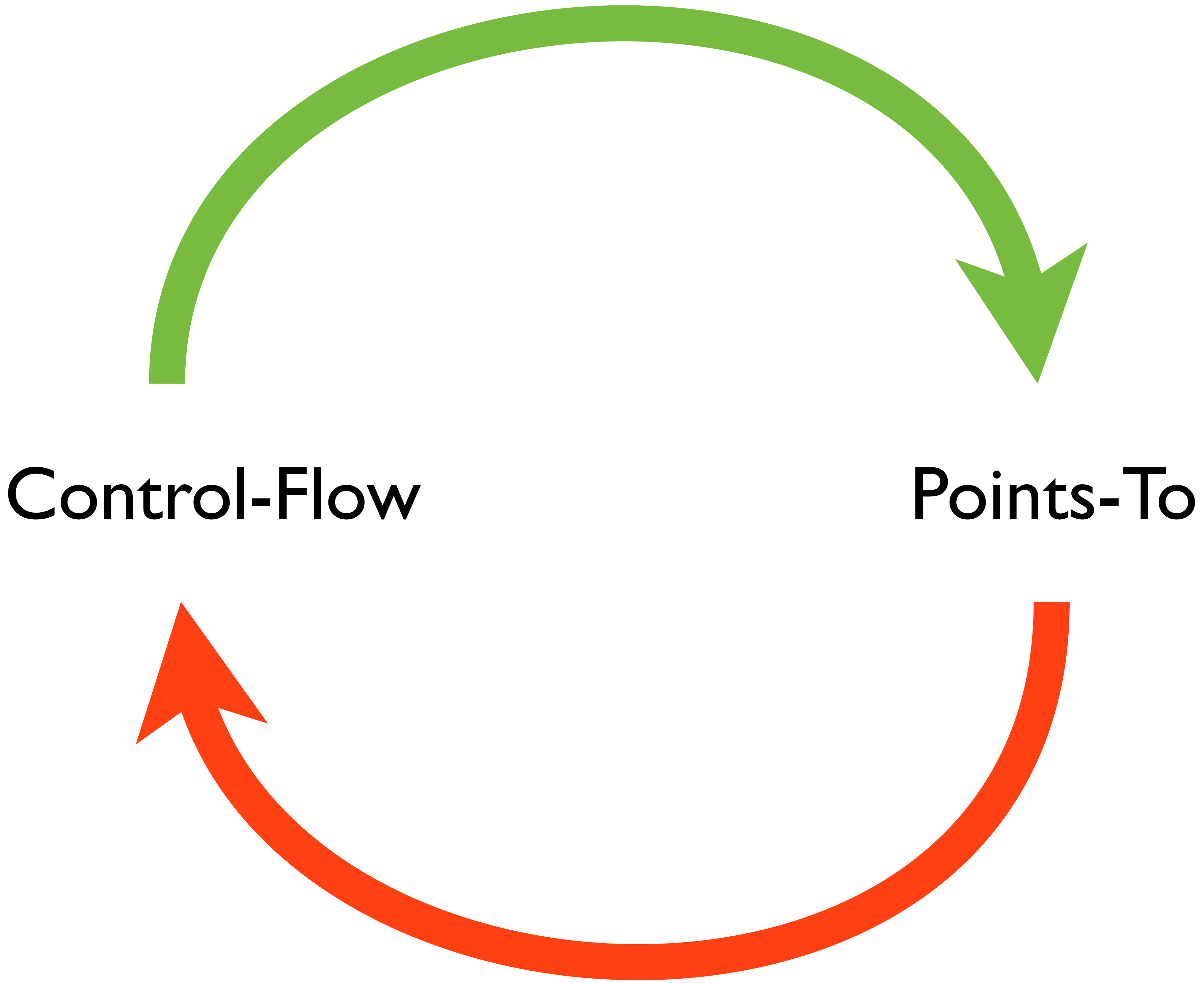
```
(f o)
```

```
o.f()
```

What is 0?

What is pointer analysis?

What control-flow analysis is not.


Control-Flow
Points-To

`(f list)`

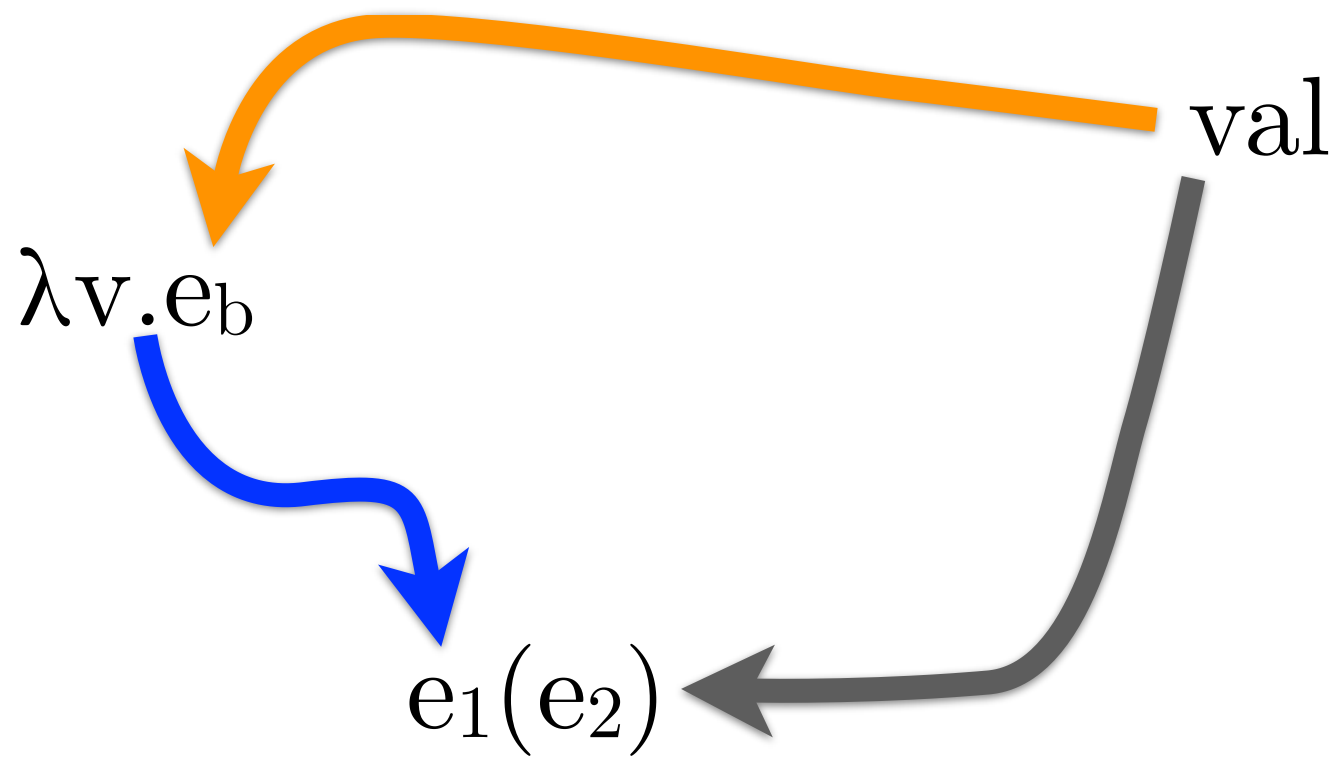
```
(apply f list)
```

```
f.apply(o,list)
```

`(λ (v) e)`

$(\lambda\ \mathtt{v}\ e)$

$$e_1(e_2)$$



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

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

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

We don’t add flow-sensitivity

We take away flow-sensitivity

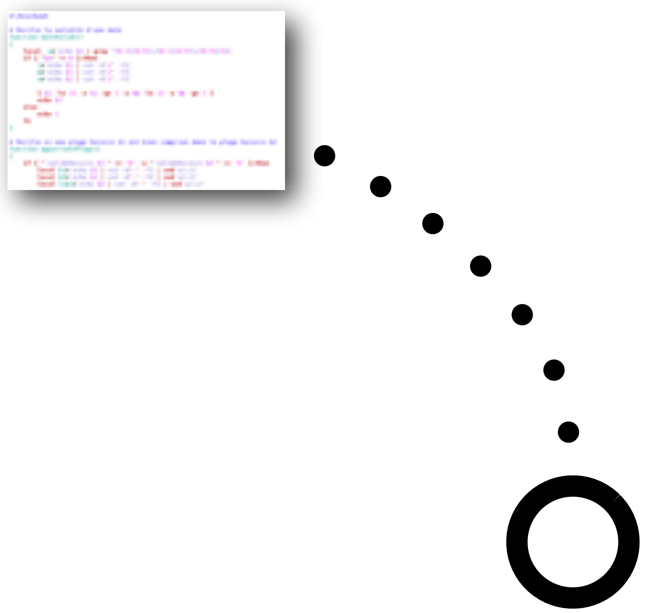
We don’t add path-sensitivity

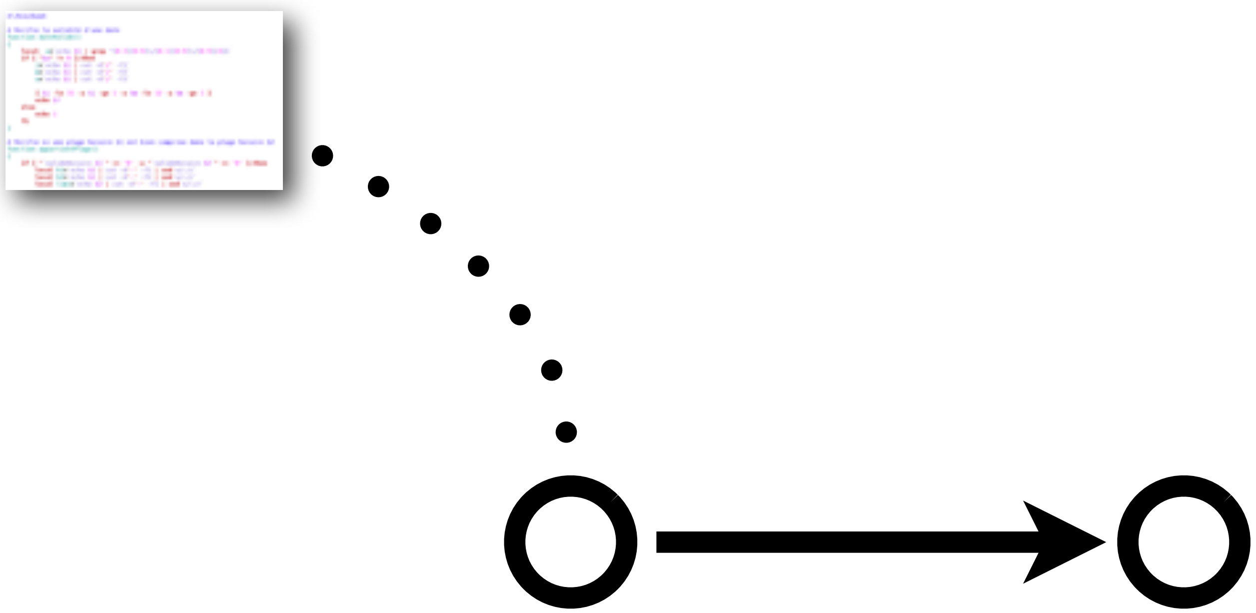
We take away path-sensitivity

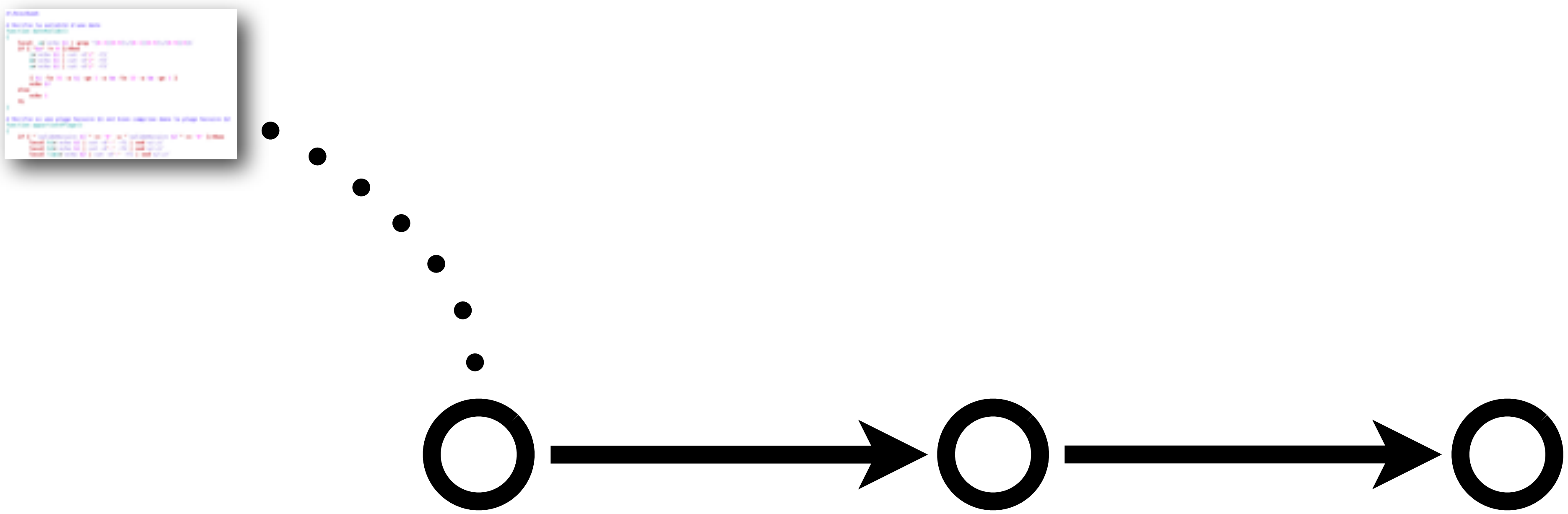
How do we analyze control-flow?

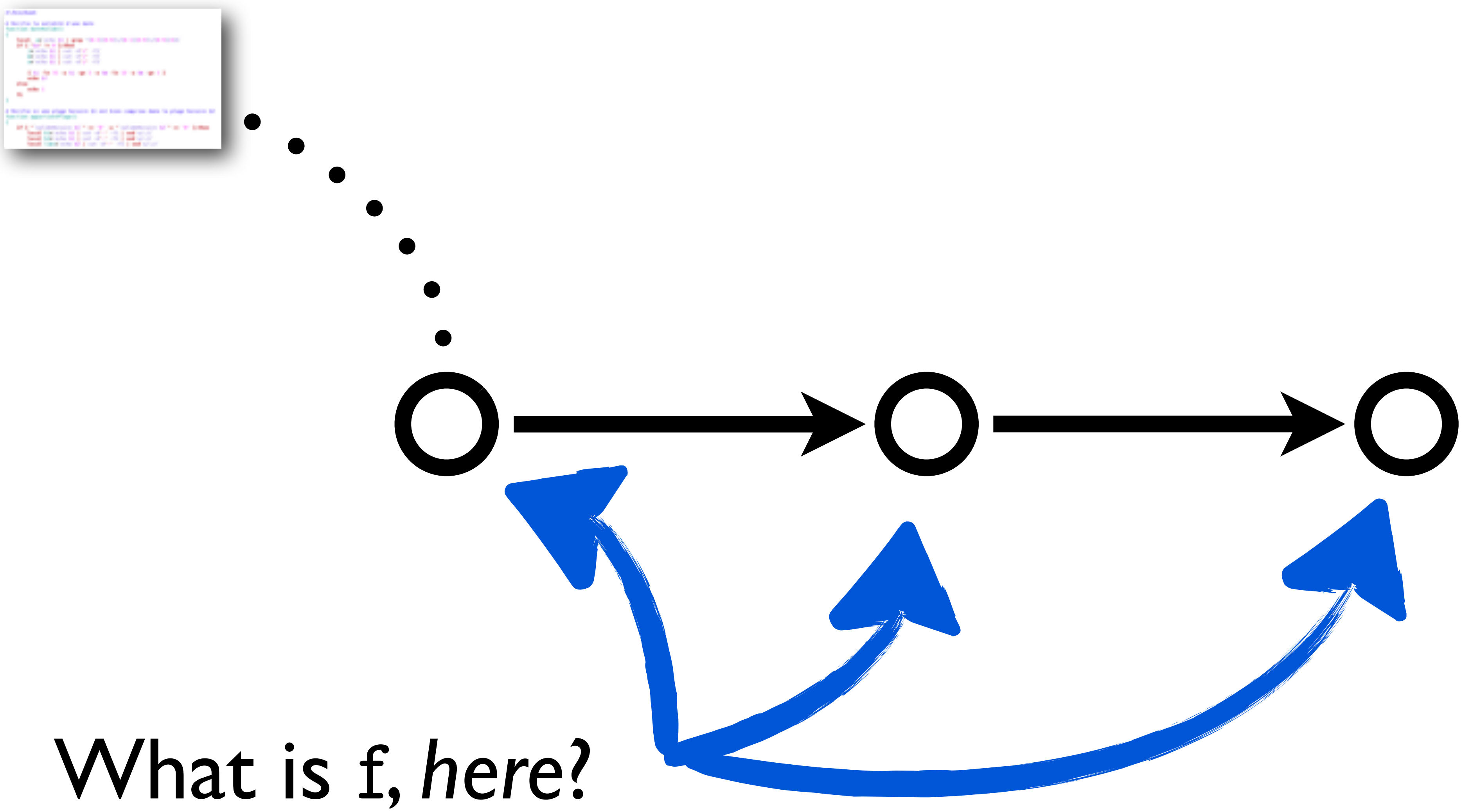
Build an interpreter.

```
#!/bin/bash

# Verifie la validité d'une date
function dateValide()
{
    local  a=`echo $1 | grep ^[0-3][0-9]\/[0-1][0-9]\/[0-9]{4}$`
    if [ "$a" != 0 ];then
        j=`echo $1 | cut -d'/' -f1`
        m=`echo $1 | cut -d'/' -f2`
        a=`echo $1 | cut -d'/' -f3`

        [ $j -le 31 -a $j -ge 1 -a $m -le 12 -a $m -ge 1 ]
        echo $?
    else
        echo 1
    fi
}

# Verifie si une plage horaire $1 est bien comprise dans la plage horaire $2
function appartientPlage()
{
    if [ "`valideHoraire $1`" == "0" -a "`valideHoraire $2`" == "0" ];then
        local h1=`echo $1 | cut -d'-' -f1 | sed s/://`
        local h2=`echo $1 | cut -d'-' -f2 | sed s/://`
        local lim1=`echo $2 | cut -d'-' -f1 | sed s/://`
```

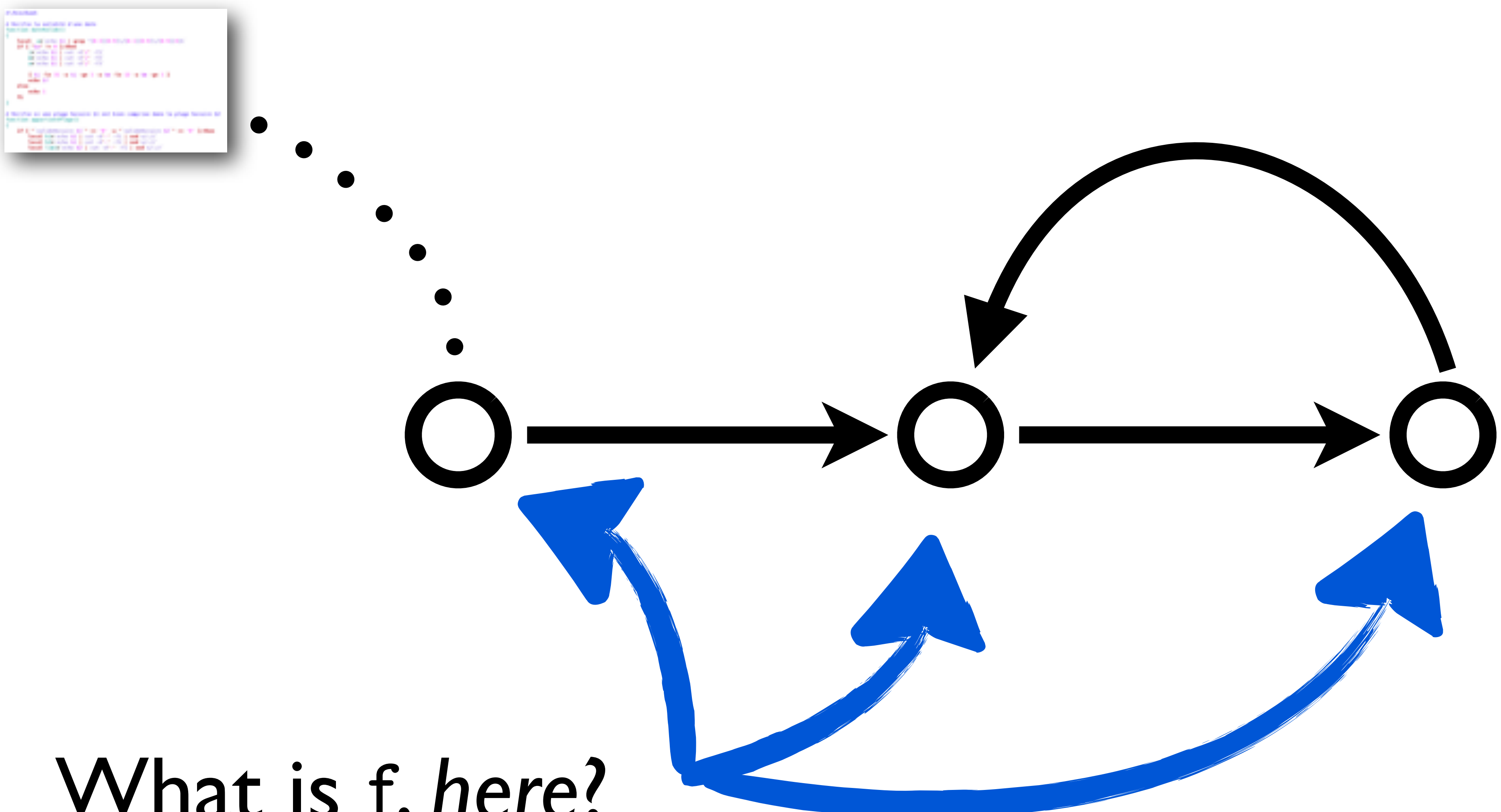




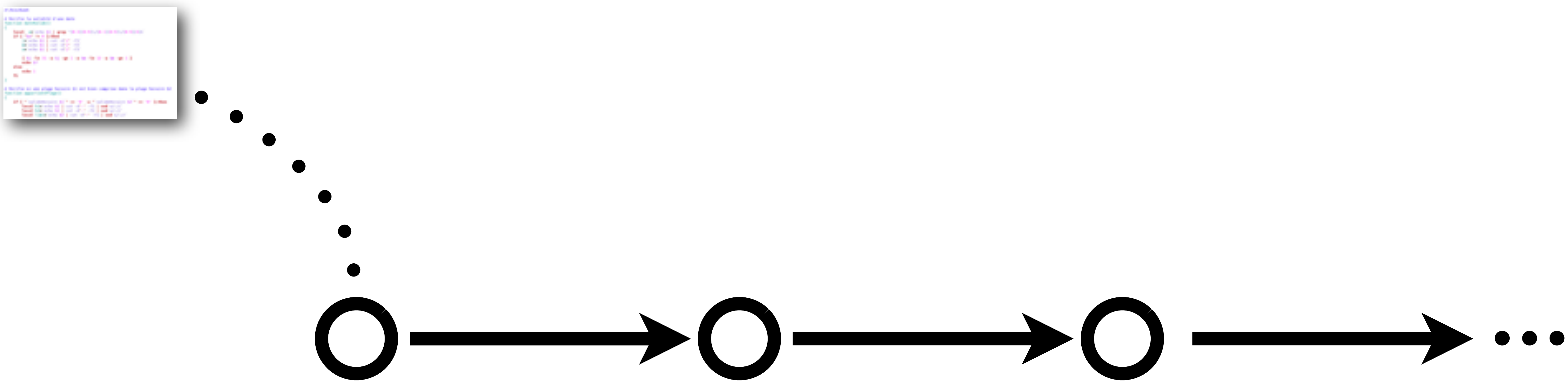




What is f, *here*?



What is f, *here*?



Build an interpreter.

Build an abstract interpreter.

Make it finite.

Make it finite.

(Might, SAS 2010) (Van Horn and Might, ICFP 2010)

CESK

(Felleisen & Friedman, 1986)

Control

Environment

Store

Kontinuation

Statement

Registers

Heap

Stack

Control

Environment

Store

Kontinuation

CESK

Control

$$
\begin{aligned}
e \ ::= \ & (\lambda \ (v_1 \ \ldots \ v_n) \ e) \\
\mid \ & (e_0 \ e_1 \ \ldots \ e_{\mathrm{n}}) \\
\mid \ & v
\end{aligned}
$$

Control

$$
\begin{array}{rcl}
e & ::= & ae \\
  & | & ce \\
  & | & (\texttt{let}\ ([v\ ce])\ e) \\
ae & ::= & (\lambda\ (v_1\ \ldots\ v_n)\ e) \\
  & | & v \\
ce & ::= & (e_0\ e_1\ \ldots\ e_n)
\end{array}
$$

(Flanagan *et al.*, 1993)

CESK

Environment

$$\text{Variable} \rightarrow \mathit{Address}$$

Store

$$Address \rightarrow Value$$

Store

$$Value = Closure$$

Store

$$Closure = \textsf{Lambda} \times Env$$

Store

$$Closure = \mathsf{Lambda} \times Env$$

$$Object = \mathsf{Class} \times Struct$$

Store

$$Value = \mathsf{Code} \times Data$$

$$Closure = \ \mathsf{Lambda} \times Env$$

$$Object = \mathsf{Class} \times Struct$$

CESK

Kontinuation

$$Frame^*$$

Kontinuation

$$Frame = \mathsf{Var} \times \mathsf{Exp} \times Env$$

CESK

C E S K

$$\mathsf{Exp} \times Env \times Store \times Kont$$

$$\Sigma = \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont}$$

$$\Sigma = \mathsf{Exp} \times \ Env \times \ Store \times Kont$$

$$Env = \mathsf{Var} \rightarrow \ Addr$$

$$Store = Addr \rightarrow Value$$

$$Value = \ Clo$$

$$Clo = \ \mathsf{Lambda} \ \times Env$$

$$Kont = \ Frame^{*}$$

$$Frame = \mathsf{Var} \times \mathsf{Exp} \times Env$$

$$\left(\Rrightarrow\right) \subseteq \Sigma \times \Sigma$$

$$
\begin{aligned}
(\llbracket \texttt{(}f\ æ_1 \ldots æ_n\texttt{)} \rrbracket, \rho, \sigma, \kappa) &\Rightarrow (e, \rho'', \sigma', \kappa), \text{ where} \\
(\llbracket \texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ e\texttt{)} \rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\
a_i &= alloc(v_i, \ldots)
\end{aligned}
$$

$$
\begin{aligned}
(\text{æ}, \rho, \sigma, \kappa) &\Rightarrow (e, \rho'', \sigma', \kappa'), \text{ where} \\
(v, e, \rho') : \kappa' &= \kappa \\
\rho'' &= \rho'[v \mapsto a] \\
\sigma' &= \sigma[a \mapsto \mathcal{A}(\text{æ}, \rho, \sigma)] \\
a &= alloc(v, \ldots)
\end{aligned}
$$

$$(\llbracket \texttt{(let ((}v\ ce\texttt{))}\ e\texttt{)} \rrbracket, \rho, \sigma, \kappa) \Rightarrow (ce, \rho, \sigma', \kappa'), \text{ where}$$

$$\kappa' = (v, e, \rho) : \kappa$$

$$\sum$$

$$\infty$$

$$\Sigma = \mathsf{Exp} \times \ Env \times \ Store \times Kont$$

$$Env = \mathsf{Var} \rightarrow \ Addr$$

$$Store = Addr \rightarrow Value$$

$$Value = \ Clo$$

$$Clo = \ \mathsf{Lambda} \ \times Env$$

$$Kont = \ Frame^*$$

$$Frame = \mathsf{Var} \times \mathsf{Exp} \times Env$$

$$Addr$$

$$Frame^*$$

$$\Sigma = \mathsf{Exp} \times \ Env \times \ Store \times Kont$$

$$Env = \mathsf{Var} \rightarrow \ Addr$$

$$Store = Addr \rightarrow Value$$

$$Value = \ Clo$$

$$Clo = \ \mathsf{Lambda} \ \times Env$$

$$Kont \ = Frame^*$$

$$Frame = \mathsf{Var} \times \mathsf{Exp} \times Env$$

$$\Sigma = \mathsf{Exp} \times \ Env \times \ Store \times Kont$$

$$Env = \mathsf{Var} \rightarrow \ Addr$$

$$Store = Addr \rightarrow Value$$

$$Value = \ Clo + Kont$$

$$Clo = \ \mathsf{Lambda} \ \times Env$$

$$Kont = Frame^*$$

$$Frame = \mathsf{Var} \times \mathsf{Exp} \times Env$$

$$\Sigma = \mathsf{Exp} \times \ Env \times \ Store \times Addr$$

$$Env = \mathsf{Var} \rightarrow \ Addr$$

$$Store = Addr \rightarrow Value$$

$$Value = \ Clo + Kont$$

$$Clo = \ \mathsf{Lambda} \ \times Env$$

$$Kont = \mathsf{Var} \times \mathsf{Exp} \times Env \times Addr$$

$$\Sigma = \mathsf{Exp} \times \; Env \times \; Store \times Addr$$

$$Env = \mathsf{Var} \to \; \widehat{Addr}$$

$$Store = \widehat{Addr} \to Value$$

$$Value = \; Clo + Kont$$

$$Clo = \; \mathsf{Lambda} \; \times Env$$

$$Kont = \mathsf{Var} \times \mathsf{Exp} \times \; Env \times \widehat{Addr}$$

$$\Sigma = \mathsf{Exp} \times Env \times Store \times Addr$$

$$Env = \mathsf{Var} \rightarrow \widehat{Addr}$$

$$Store = \widehat{Addr} \rightarrow \mathcal{P}(Value)$$

$$Value = Clo + Kont$$

$$Clo = \mathsf{Lambda} \times Env$$

$$Kont = \mathsf{Var} \times \mathsf{Exp} \times Env \times \widehat{Addr}$$

$$\hat{\Sigma} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Addr}$$

$$\widehat{Env} = \mathsf{Var} \rightarrow \widehat{Addr}$$

$$\widehat{Store} = \widehat{Addr} \rightarrow \mathcal{P}(\widehat{Value})$$

$$\widehat{Value} = \widehat{Clo} + \widehat{Kont}$$

$$\widehat{Clo} = \mathsf{Lambda} \times \widehat{Env}$$

$$\widehat{Kont} = \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} \times \widehat{Addr}$$

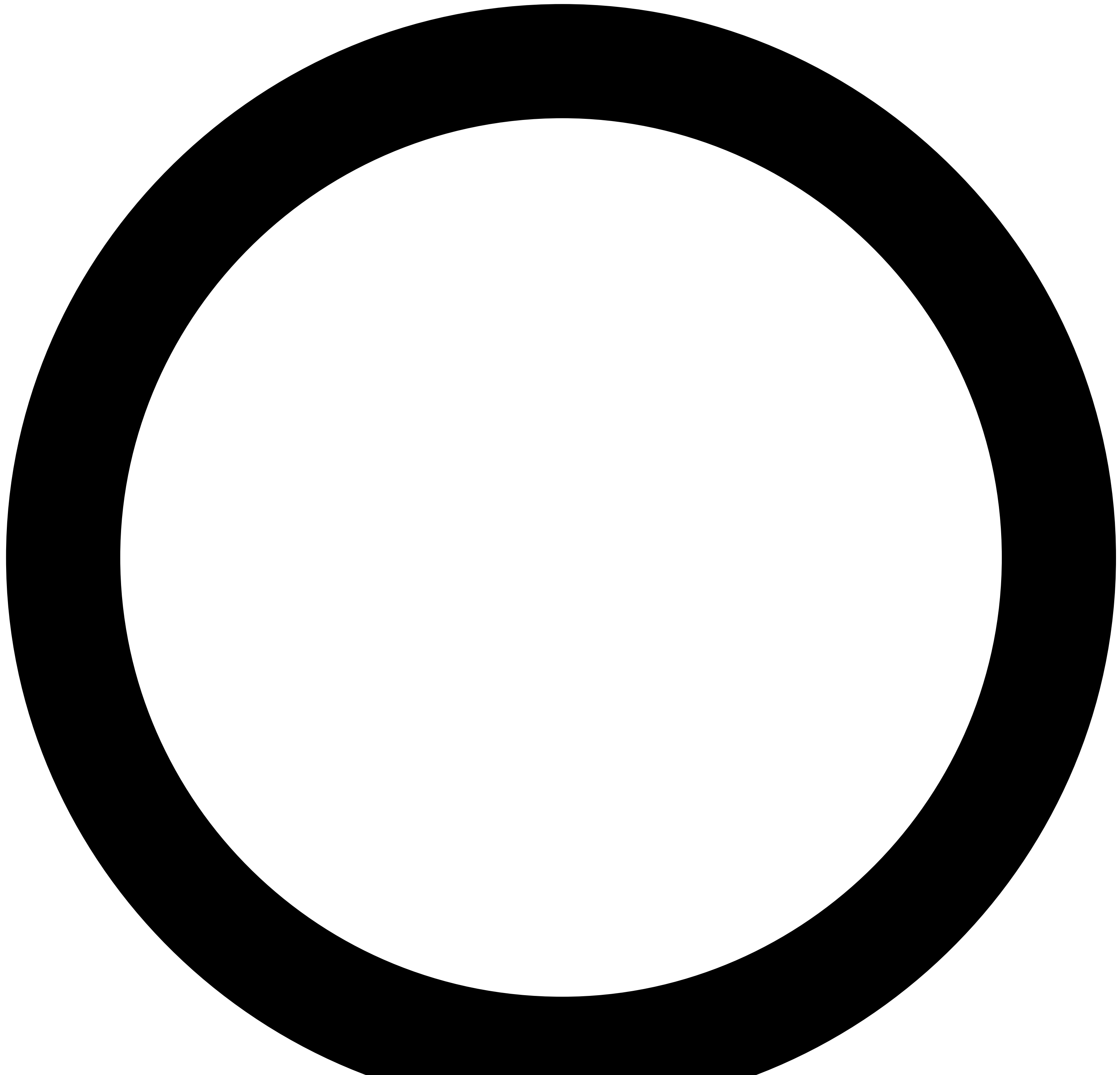
$$\left(\rightsquigarrow\right) \subseteq \widehat{\sum} \times \widehat{\sum}$$

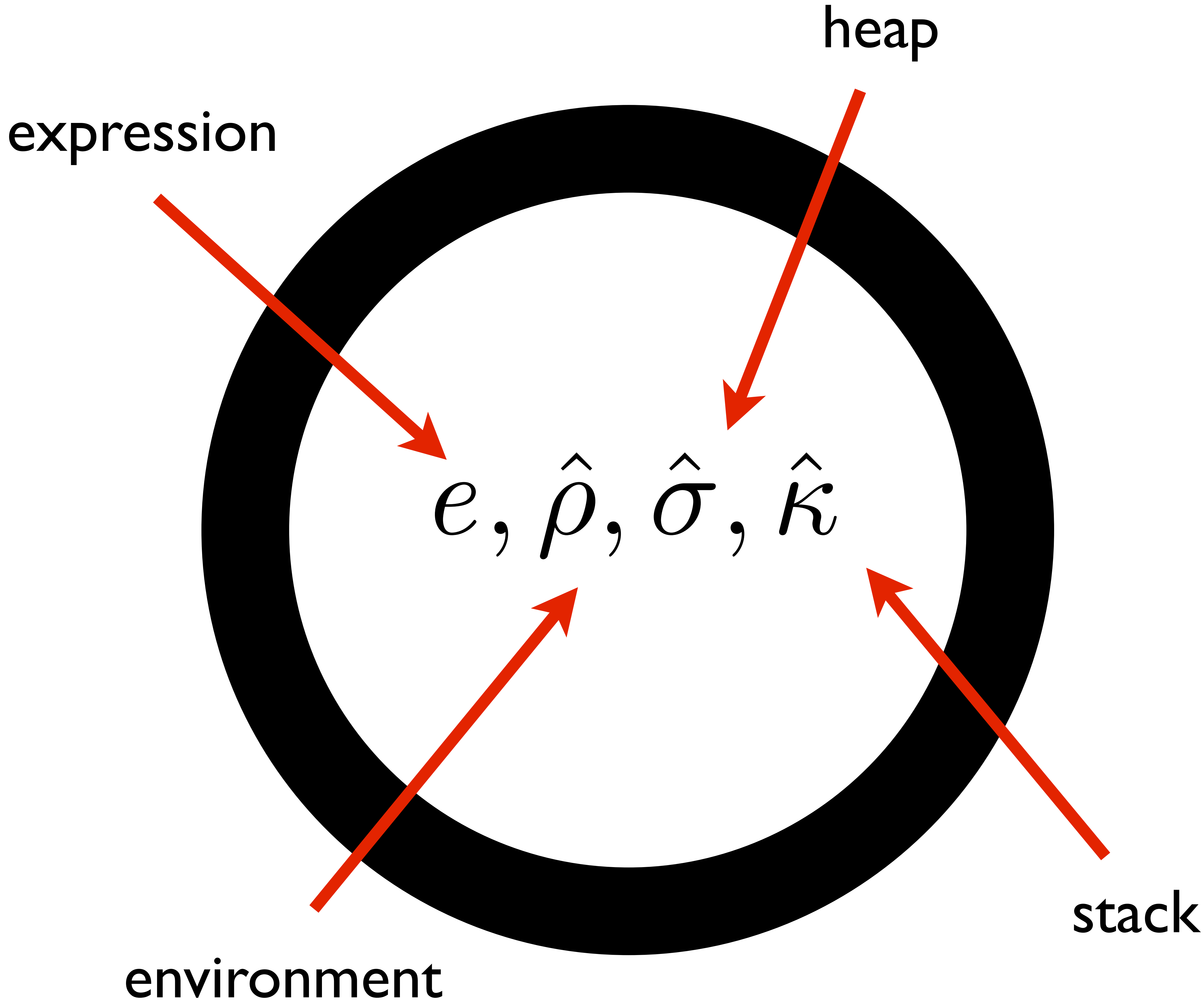
$$
\begin{aligned}
(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{a}_\kappa) &\rightsquigarrow (e, \hat{\rho}'', \hat{\sigma}', \hat{a}_\kappa), \text{ where} \\
(\llbracket (\lambda\ (v_1 \ldots v_n)\ e) \rrbracket, \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})] \\
\hat{a}_i &= \widehat{alloc}(v_i, \ldots)
\end{aligned}
$$

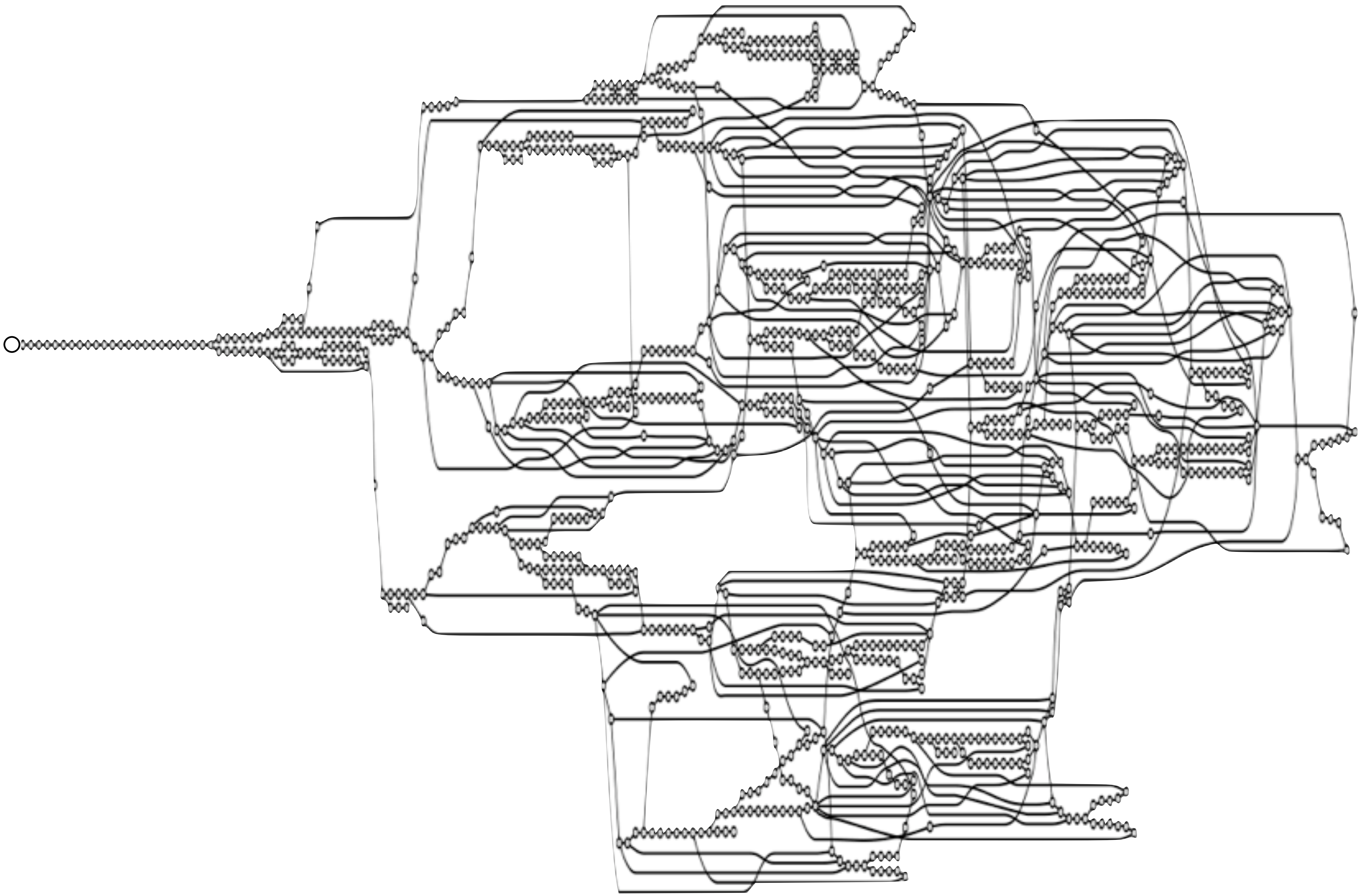
$$
\begin{aligned}
(\text{æ}, \hat{\rho}, \hat{\sigma}, \hat{a}_\kappa) &\leadsto (e, \hat{\rho}'', \hat{\sigma}', \hat{a}'_\kappa), \text{ where} \\
\mathbf{letk}(v, e, \hat{\rho}', \hat{a}'_\kappa) &\in \hat{\sigma}(\hat{a}_\kappa) \\
\hat{\rho}'' &= \hat{\rho}'[v \mapsto \hat{a}] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})] \\
\hat{a} &= \widehat{alloc}(v, \ldots)
\end{aligned}
$$

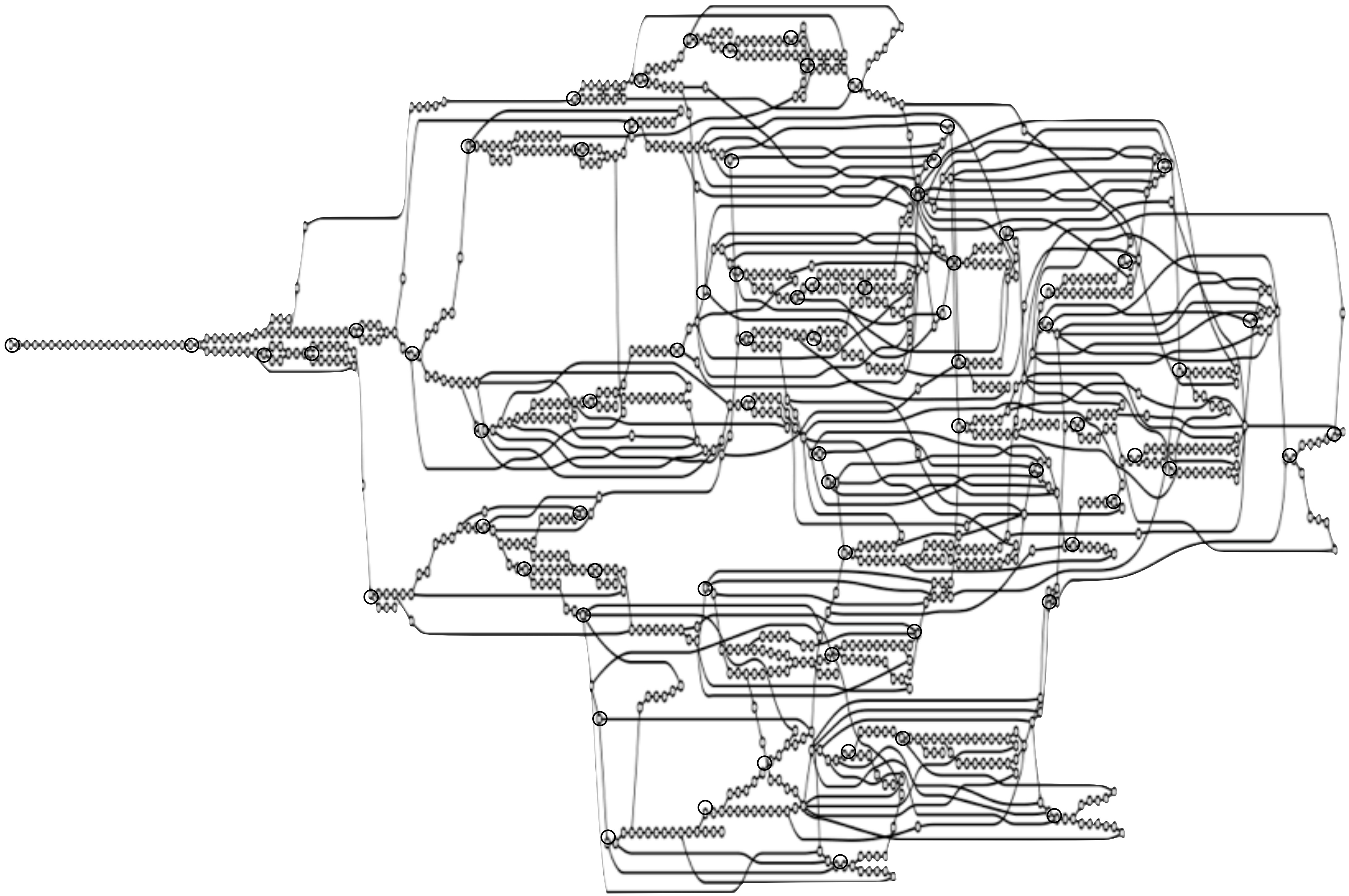
$$
\begin{aligned}
(\llbracket \texttt{(let ((}v\ ce\texttt{))}\ e\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{a}_\kappa) \leadsto (ce, \hat{\rho}, \hat{\sigma}', \hat{a}'_\kappa), \text{ where} \\
\hat{\sigma}' &= \sigma \sqcup [\hat{a}'_\kappa \mapsto \{\mathbf{letk}(v, e, \rho, \hat{a}_{\hat{\kappa}})\}] \\
\hat{a}'_\kappa &= \widehat{alloc}(\ldots)
\end{aligned}
$$

```
(define `λ (λ 'love 'I 'you))
((λ λ λ)
  `(λ (⊙ λ U) λ)
  `(λ (λ ⊙ U) λ)
  `(λ (U ⊙ λ) λ))
```




heap
expression
$e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}$
environment
stack





$$\bigsqcup_{i} \hat{\sigma}_i$$

$$\bigsqcup_i \hat{\sigma}_i$$

Abstract garbage collection

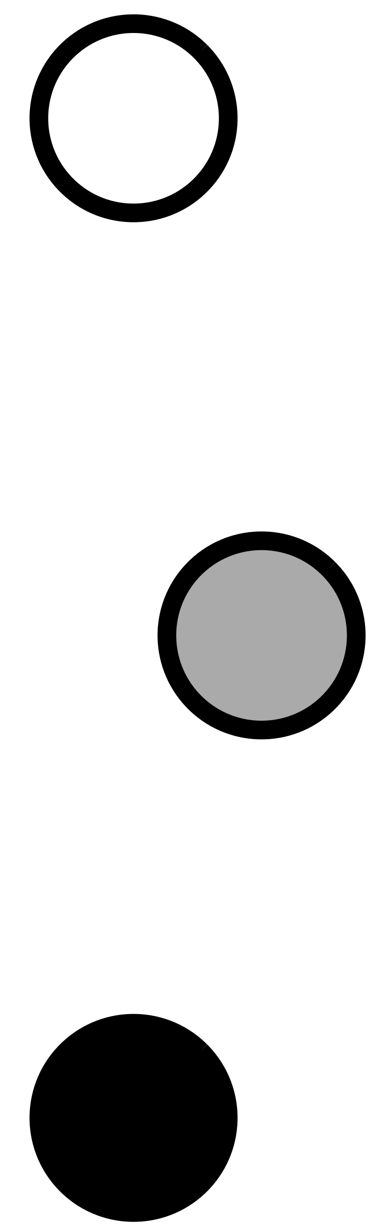
(Might and Shivers, 2006)

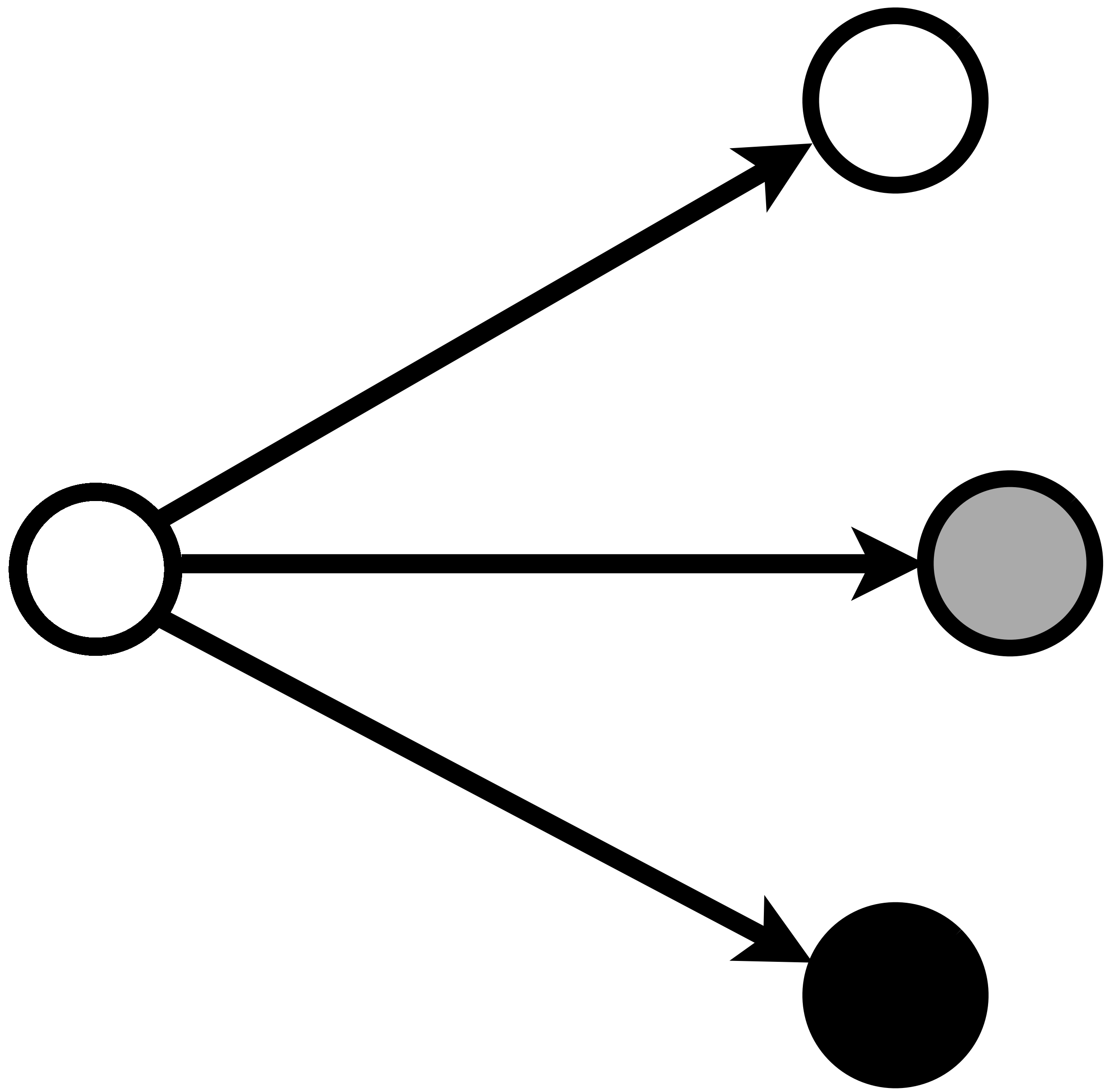
(f o)

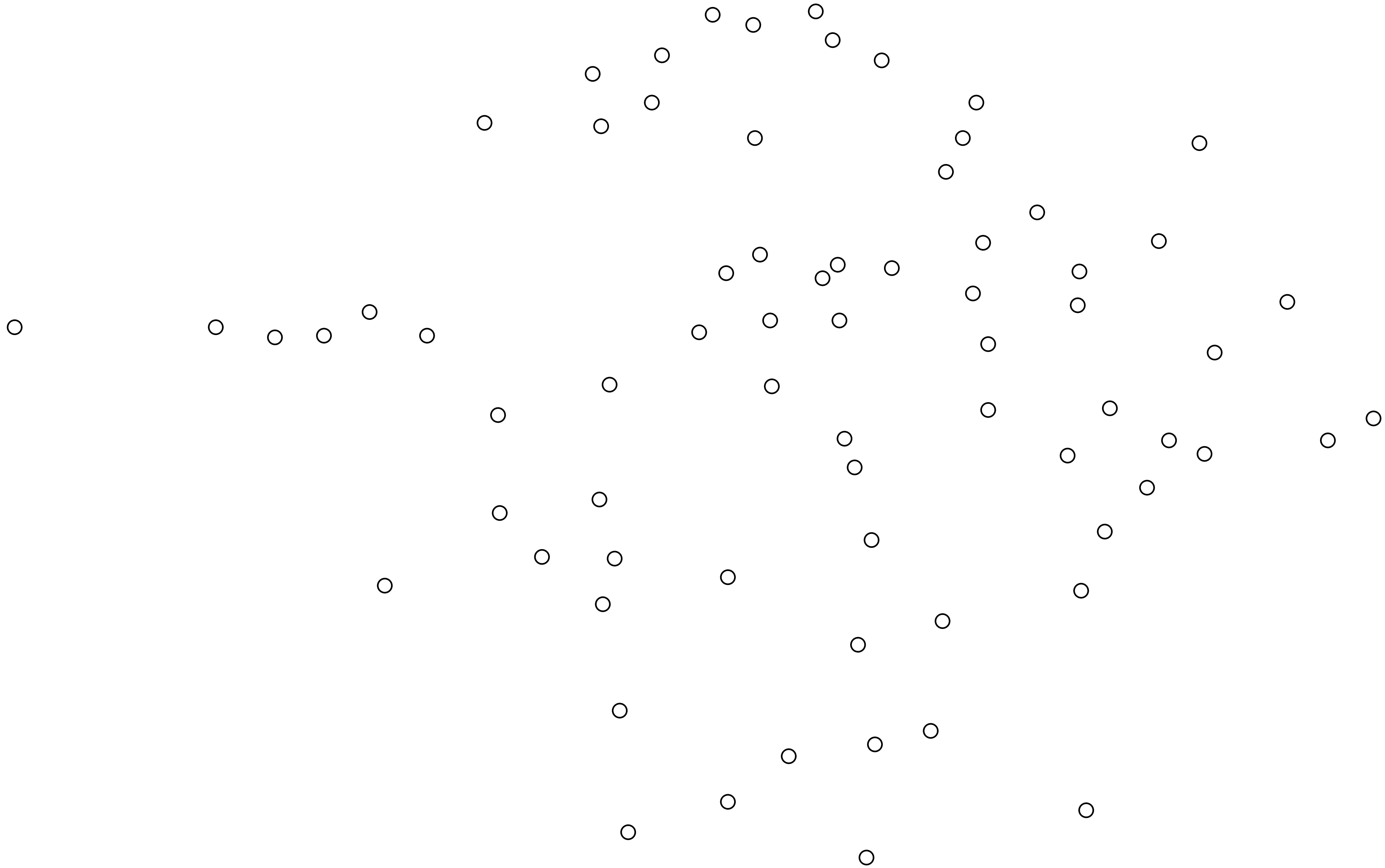
What is `f`?

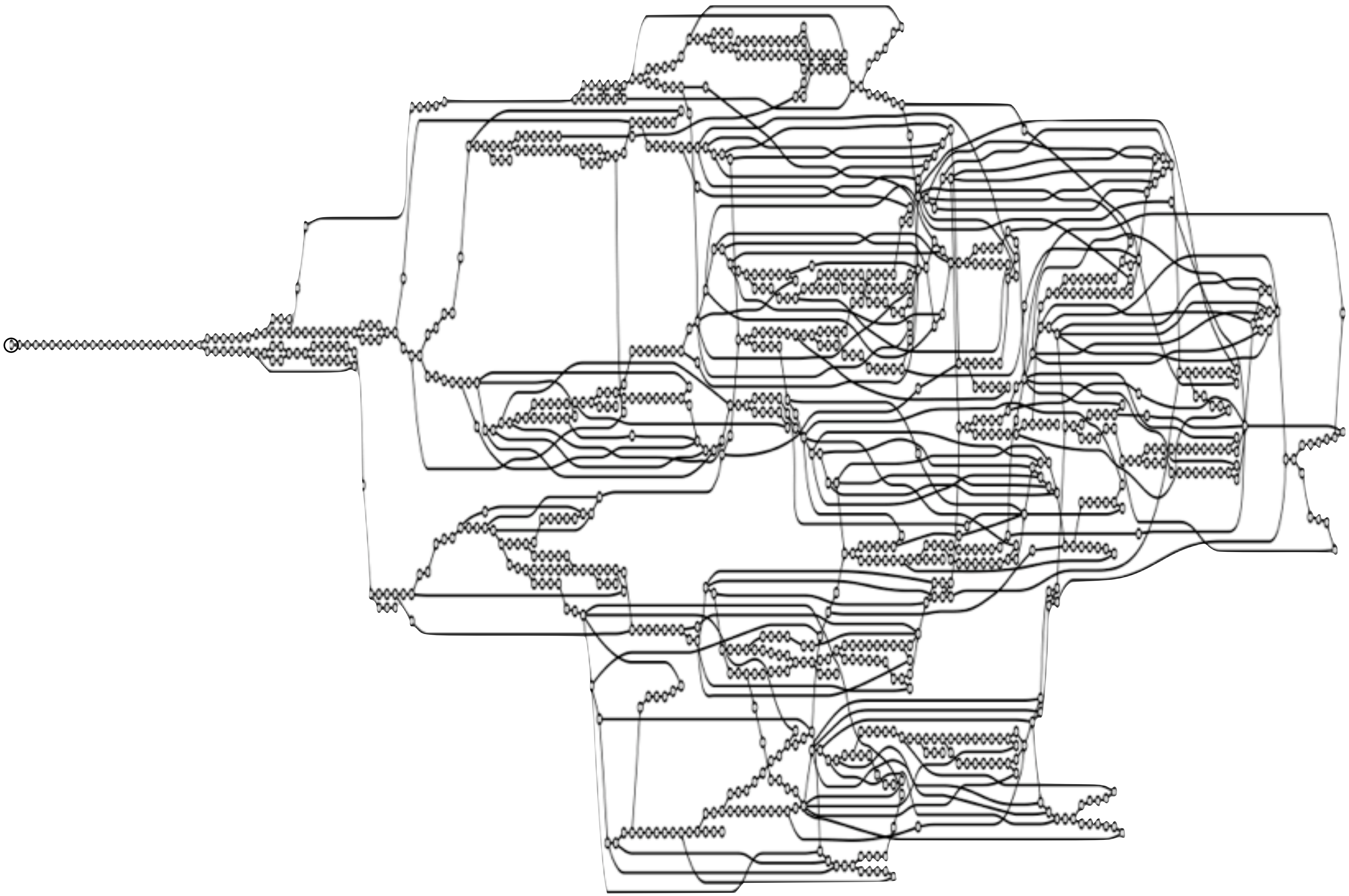
`f` is or or

`f` is ○ or ○ or ●







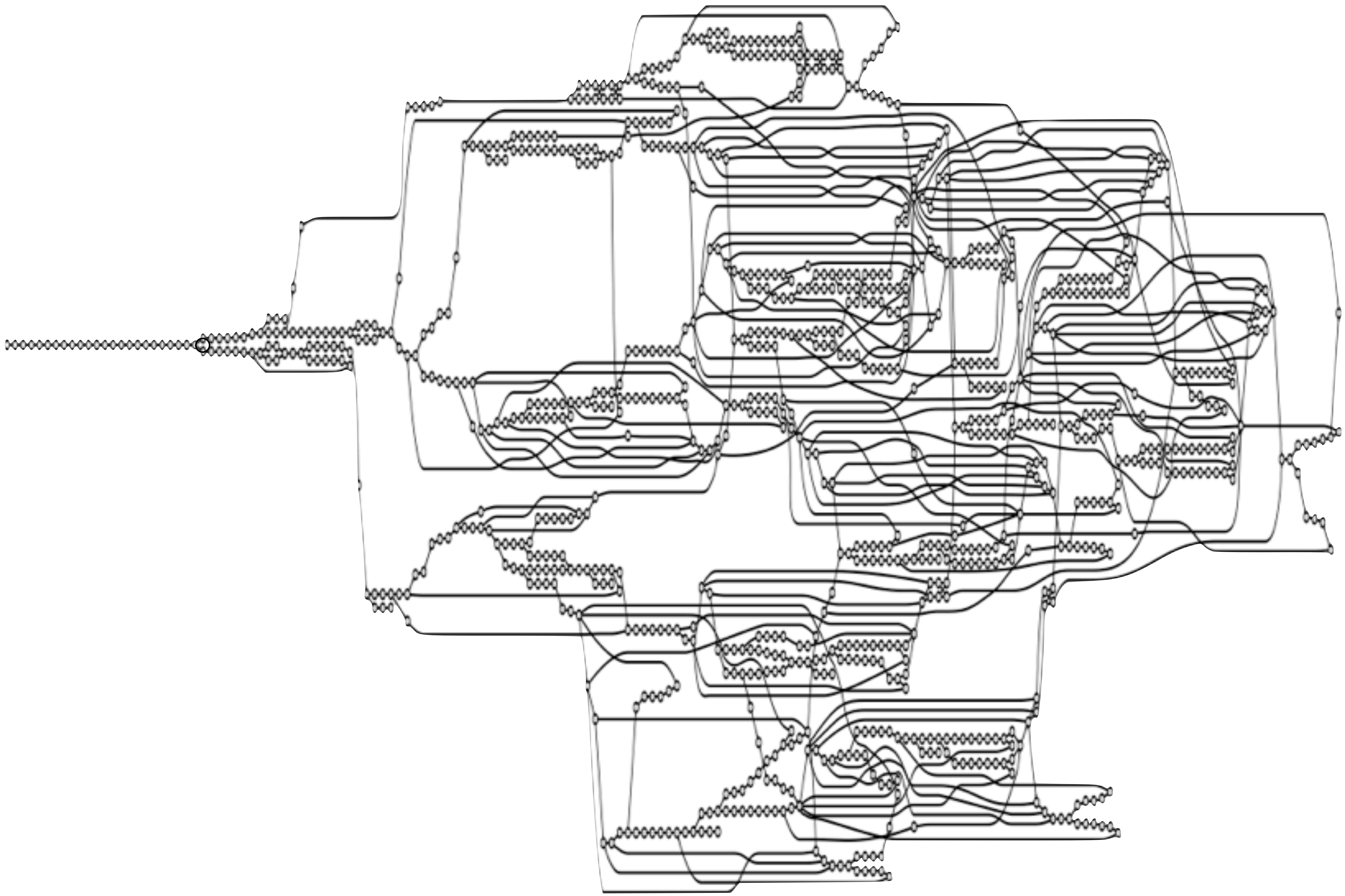


Problem?

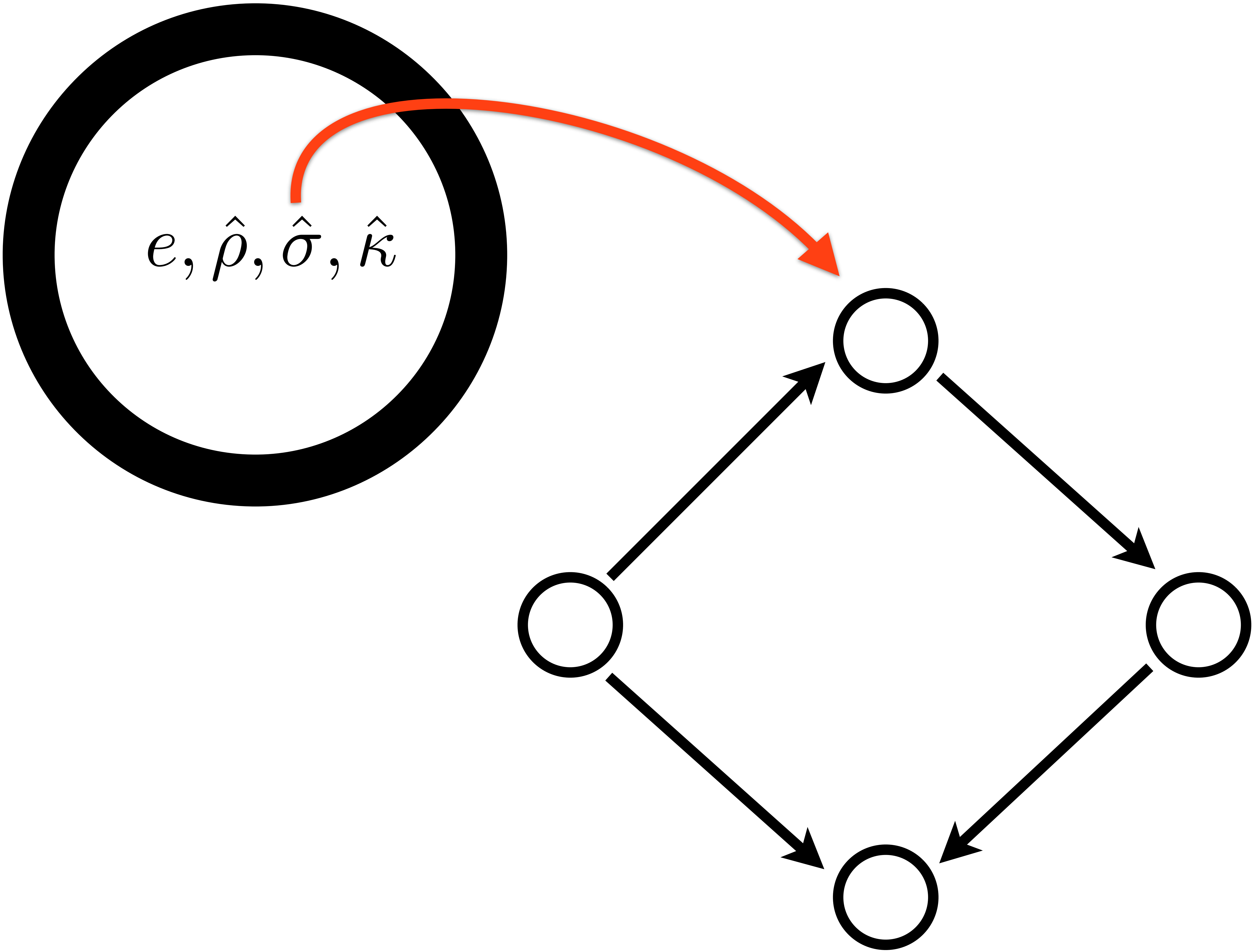
Finite heap.

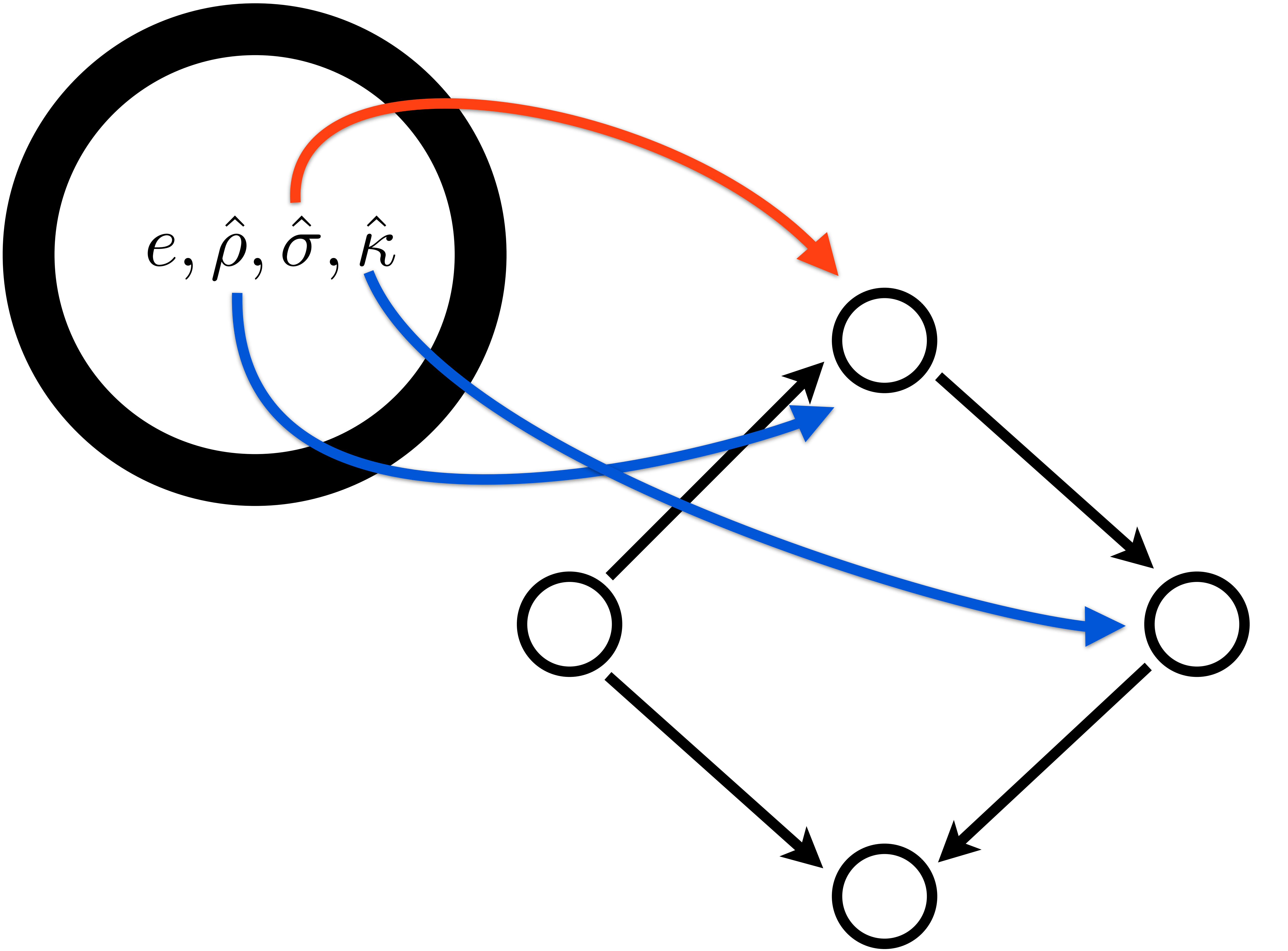
Solution?

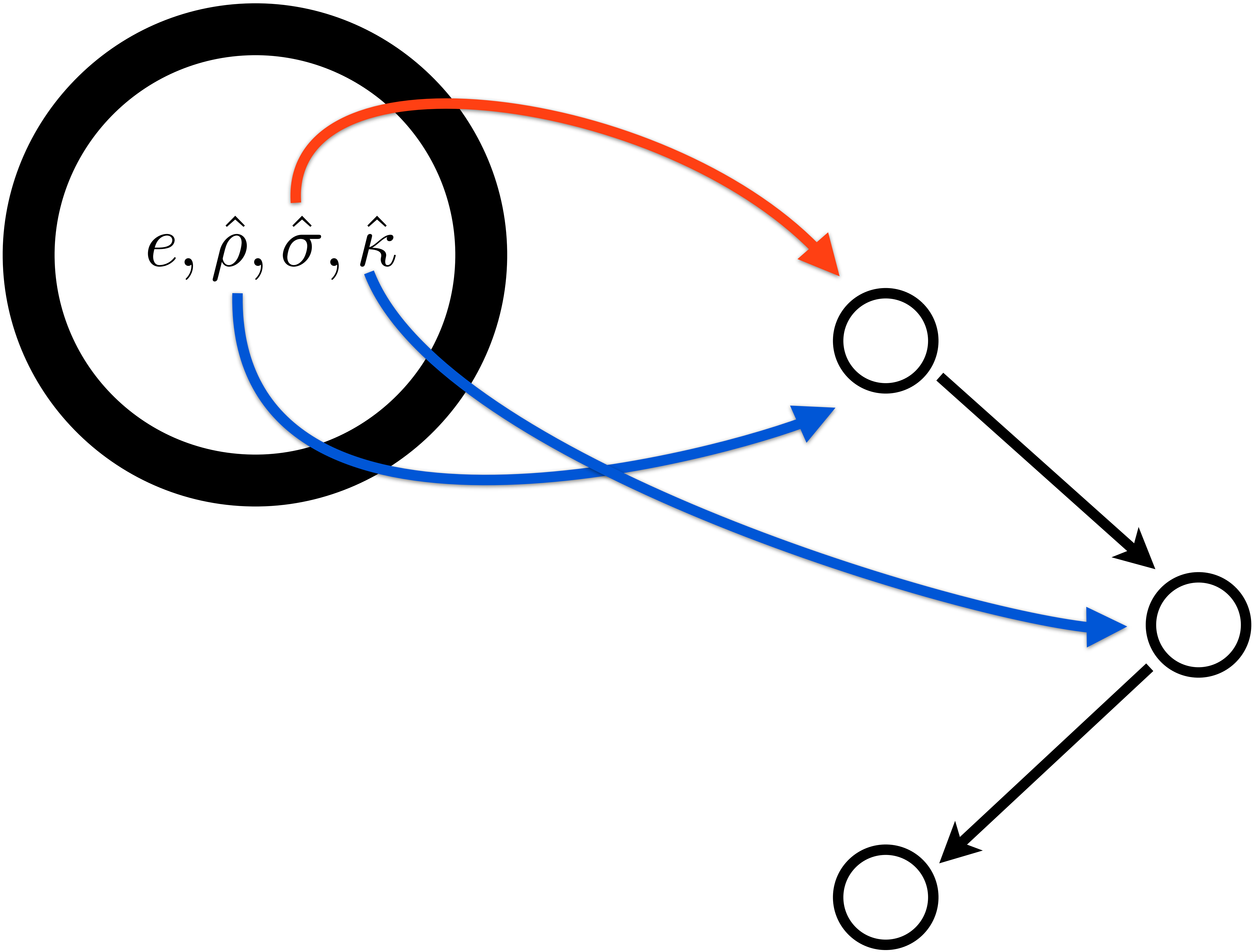
Toss garbage.



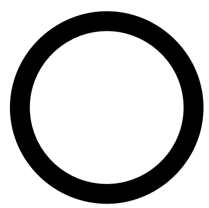
$$e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}$$

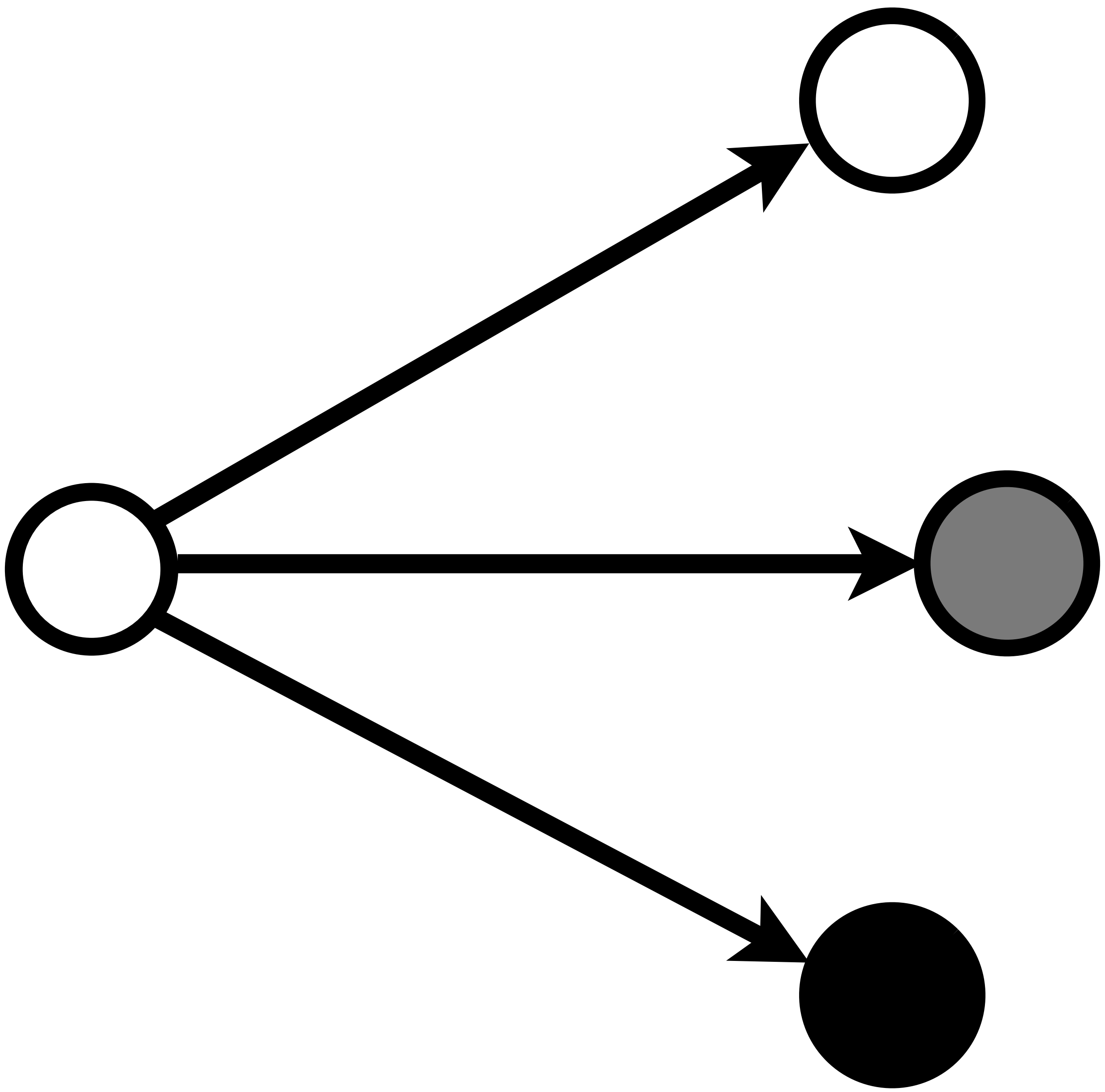

$e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}$

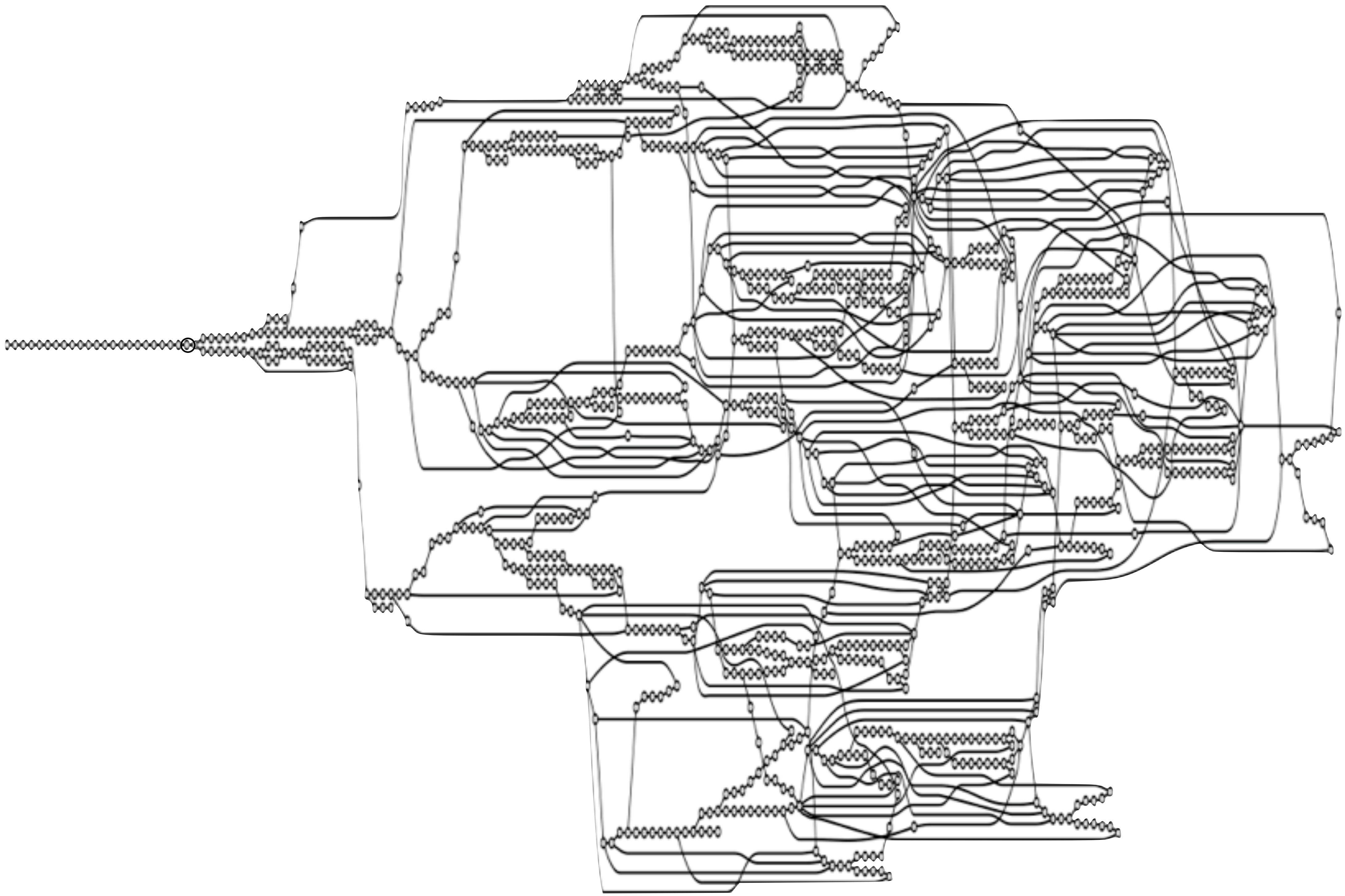

$e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}$

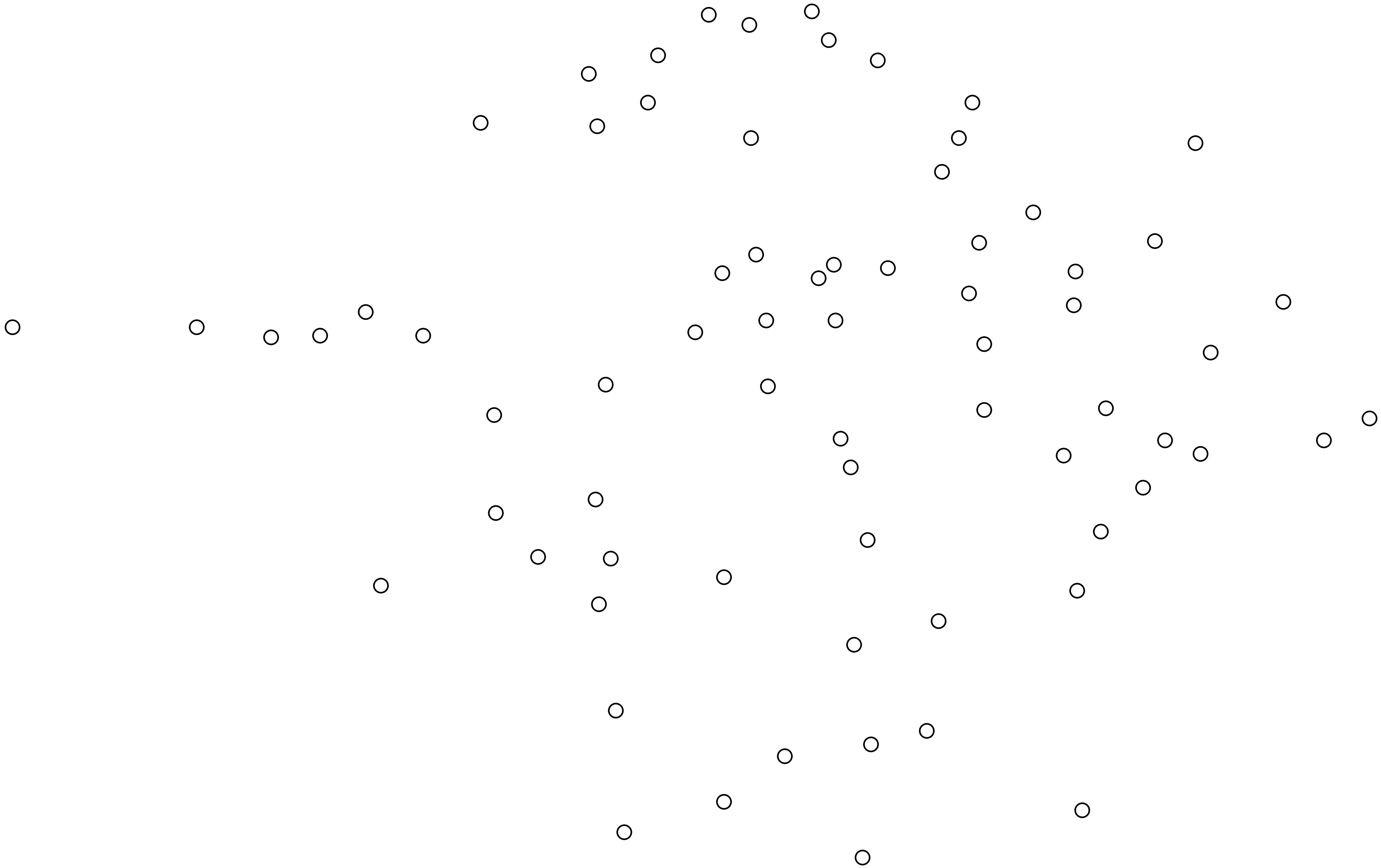

$e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}$

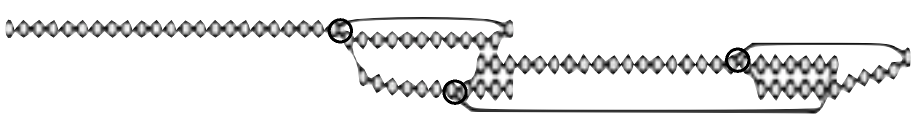
$e, \hat{\rho}, \hat{\sigma}', \hat{\kappa}$











Connections

Featherweight Java

Resolving and Exploiting the k-CFA Paradox

Illuminating Functional vs. Object-Oriented Program Analysis

Matthew Might
University of Utah
might@cs.utah.edu

Yannis Smaragdakis
University of Massachusetts
yannis@cs.umass.edu

David Van Horn
Northeastern University
dvanhorn@ccs.neu.edu

Abstract

Low-level program analysis is a fundamental problem, taking the shape of "flow analysis" in functional languages and "points-to" analysis in imperative and object-oriented languages. Despite the similarities, the vocabulary and results in the two communities remain largely distinct, with limited cross-understanding. One of the few links is Shivers's k-CFA work, which has advanced the concept of "context-sensitive analysis" and is widely known in both communities.

Recent results indicate that the relationship between the functional and object-oriented incarnations of k-CFA is not as well understood as thought. Van Horn and Mairson proved k-CFA for $k \geq 1$ to be EXPTIME-complete; hence, no polynomial-time algorithm can exist. Yet, there are several polynomial-time formulations of context-sensitive points-to analyses in object-oriented languages. Thus, it seems that functional k-CFA may actually be a profoundly different analysis from object-oriented k-CFA. We resolve this paradox by showing that the exact same specification of k-CFA is polynomial-time for object-oriented languages yet exponential-time for functional ones: objects and closures are subtly different, in a way that interacts crucially with context-sensitivity and complexity. This illumination leads to an immediate payoff: by projecting the object-oriented treatment of objects onto closures, we derive a polynomial-time hierarchy of context-sensitive CFAs for functional programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Languages, Theory

Keywords static analysis, control-flow analysis, pointer analysis, functional, object-oriented, k-CFA, m-CFA

1. Introduction

One of the most fundamental problems in program analysis is determining the entities to which an expression may refer at run-time. In imperative and object-oriented (OO) languages, this is commonly phrased as a *points-to* (or *pointer*) analysis: to which objects can a variable point? In functional languages, the problem is called *flow analysis* [11]: to which expressions can a value flow?

Both points-to and flow analysis acquire a degree of complexity for higher-order languages: functional languages have first-class functions and object-oriented languages have dynamic dispatch; these features conspire to make call-target resolution depend on the flow of values, even as the flow of values depends on what targets are possible for a call. That is, data-flow depends on control-flow, yet control-flow depends on data-flow. Appropriately, this problem is commonly called *control-flow analysis* (CFA).

Shivers's k-CFA [17] is a well-known family of control-flow analysis algorithms, widely recognized in both the functional and the object-oriented world. k-CFA popularized the idea of context-sensitive flow analysis.[1] Nevertheless, there have always been annoying discrepancies between the experiences in the application of k-CFA in the functional and the OO world. Shivers himself notes in his "Best of PLDI" retrospective that "the basic analysis, for any $k > 0$ [is] intractably slow for large programs" [16]. This contradicts common experience in the OO setting, where a 1- and 2-CFA analysis is considered heavy but certainly possible [2, 10].

To make matters formally worse, Van Horn and Mairson [19] recently proved k-CFA for $k \geq 1$ to be EXPTIME-complete, i.e., non-polynomial. Yet the OO formulations of k-CFA have provably polynomial complexity (e.g., Bravenboer and Smaragdakis [2] express the algorithm in Datalog, which is a language that can only express polynomial-time algorithms). This paradox seems hard to resolve. Is k-CFA misunderstood? Has inaccuracy crept into the transition from functional to OO?

In this paper we resolve the paradox and illuminate the deep differences between functional and OO context-sensitive program analyses. We show that the exact same formulation of k-CFA is exponential-time for functional programs yet polynomial-time for OO programs. To ensure fidelity, our proof appeals directly to Shivers's original definition of k-CFA and applies it to the most common formal model of Java, Featherweight Java.

As might be expected, our finding hinges on the fundamental difference between typical functional and OO languages: the former create implicit closures when lambda expressions are created, while the latter require the programmer to explicitly "close" (i.e., pass to a constructor) the data that a newly created object can reference. At an intuitive level, this difference also explains why the

[1] Although the k-CFA work is often used as a synonym for "k-context-sensitive" in the OO world, k-CFA is more correctly an algorithm that packages context-sensitivity together with several other design decisions. In the terminology of OO points-to analysis, k-CFA is a k-call-site-sensitive, field-sensitive points-to analysis algorithm with a context-sensitive heap and with on-the-fly call-graph construction. (Lhoták [9] and Lhoták and Hendren [10] are good references for the classification of points-to analysis algorithms.) In this paper we use the term "k-CFA" with this more precise meaning, as is common in the functional programming world, and not just as a synonym for "k-context-sensitive". Although this classification is more precise, it still allows for a range of algorithms, as we discuss later.

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Stmt} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{KontPtr} \times \widehat{Time} \\
\hat{\beta} \in \widehat{BEnv} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \widehat{D} \\
\hat{d} \in \hat{D} &= \mathcal{P}\left(\widehat{Val}\right) \\
\widehat{val} \in \widehat{Val} &= \widehat{Obj} + \widehat{Kont} \\
\hat{o} \in \widehat{Obj} &= \mathsf{ClassName} \times \widehat{BEnv} \\
\hat{\kappa} \in \widehat{Kont} &= \mathsf{Var} \times \mathsf{Stmt} \times \widehat{BEnv} \times \widehat{KontPtr} \\
\hat{a} \in \widehat{Addr} &\text{ is a } \textbf{finite} \text{ set of addresses} \\
\overset{\hat{\kappa}}{p} \in \widehat{KontPtr} &\subseteq \widehat{Addr} \\
\hat{t} \in \widehat{Time} &\text{ is a } \textbf{finite} \text{ set of time-stamps.}
\end{aligned}
$$

Figure 7. Abstract state-space for A-Normal Featherweight Java.

Dalvik

$$
\begin{aligned}
\langle \texttt{nop} :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma, \kappa\rangle\\
\langle \texttt{move-object}(r_d, r_s) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(r_d, fp) \mapsto \sigma(r_s, fp)], \kappa\rangle\\
\langle \texttt{return-void} :: \vec{stmt}', fp', \sigma, \mathbf{fnk}(\vec{stmt}, fp, \kappa)\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma, \kappa\rangle\\
\langle \texttt{return-object}(r) :: \vec{stmt}', fp', \sigma, \mathbf{fnk}(\vec{stmt}, fp, \kappa)\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(\texttt{ret}, fp) \mapsto \sigma(n, fp')], \kappa\rangle\\
\langle \texttt{const}(r, c) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(r, fp) \mapsto c], \kappa\rangle\\
\langle \texttt{throw}^{\ell}(r) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \mathcal{S}(\ell'), fp', \sigma[(\texttt{exn}, fp') \mapsto \sigma(r, fp)], \kappa'\rangle\\
&\quad\text{where } (\ell', fp', \kappa') = \mathcal{H}(\ell, fp, \kappa)\\
\langle \texttt{goto}(\ell) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \mathcal{S}(\ell), fp, \sigma, \kappa\rangle\\
\langle \texttt{new-instance}(r, \tau) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(r, fp) \mapsto o], \kappa\rangle\\
&\quad\text{where } o = new(\varsigma)\\
\langle \texttt{if-eq}(r, r', \ell) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \mathcal{S}(\ell), fp, \sigma, \kappa\rangle \text{ if } \sigma(r, fp) = \sigma(r', fp)\\
\langle \texttt{if-eq}(r, r', \ell) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma, \kappa\rangle \text{ if } \sigma(r, fp) \neq \sigma(r', fp)\\
\langle \texttt{iget}(r_d, r_s, \mathit{field}) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(r_d, fp) \mapsto \sigma(a)], \kappa\rangle\\
&\quad\text{where } \sigma(r_s, fp) = o \text{ and } o.\mathit{field} = a\\
\langle \texttt{iput}(r_v, r_s, \mathit{field}) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[a \mapsto \sigma(r_v, fp)], \kappa\rangle\\
&\quad\text{where } \sigma(r_s, fp) = o \text{ and } o.\mathit{field} = a\\
\langle \texttt{invoke-direct}(r_0, \ldots, r_n, id) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \mathcal{M}(id), fp', \sigma', \mathbf{fnk}(\vec{stmt}, fp, \kappa)\rangle\\
&\quad\text{where } \sigma' = \sigma[(\texttt{0}, fp') \mapsto \sigma(r_0, fp), \ldots, (n, fp') \mapsto \sigma(r_n, fp)]\\
&\qquad\qquad fp' = alloc(\varsigma)\\
\langle \texttt{invoke-virtual}(r_0, \ldots, r_n, id) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \mathcal{V}(id, \sigma(r_0, fp)), fp', \sigma', \mathbf{fnk}(\vec{stmt}, fp, \kappa)\rangle\\
&\quad\text{where } \sigma' = \sigma[(\texttt{0}, fp') \mapsto \sigma(r_0, fp), \ldots, (n, fp') \mapsto \sigma(r_n, fp)]\\
&\qquad\qquad fp' = alloc(\varsigma)\\
\langle \mathtt{\mathit{unop}}(r_d, r_s) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(r_d, fp) \mapsto v], \kappa\rangle\\
&\quad\text{where } v = \delta(\mathtt{\mathit{unop}}, \sigma(r_s, fp))\\
\langle \mathtt{\mathit{binop}}(r_d, r_1, r_2) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, fp, \sigma[(r_d, fp) \mapsto v], \kappa\rangle\\
&\quad\text{where } v = \delta(\mathtt{\mathit{binop}}, \sigma(r_1, fp), \sigma(r_2, fp))
\end{aligned}
$$

$$
\begin{aligned}
\langle \texttt{nop} :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle \\
\langle \texttt{move-object}(r_d, r_s) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(r_d, \hat{fp}) \mapsto \hat{\sigma}(r_s, \hat{fp})], \hat{\kappa}, \hat{t}\rangle \\
\langle \texttt{return-void} :: \vec{stmt}', \hat{fp}', \hat{\sigma}, \mathbf{fnk}(\vec{stmt}, \hat{fp}, \hat{a}_\kappa)\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}\rangle \text{ if } \hat{\kappa} \in \hat{\sigma}(\hat{a}_\kappa) \\
\langle \texttt{return-object}(r) :: \vec{stmt}', \hat{fp}', \hat{\sigma}, \mathbf{fnk}(\vec{stmt}, \hat{fp}, \hat{a}_\kappa)\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(\texttt{ret}, \hat{fp}) \mapsto \hat{\sigma}(n, \hat{fp}')], \hat{\kappa}\rangle \text{ if } \hat{\kappa} \in \hat{\sigma}(\hat{a}_\kappa) \\
\langle \texttt{const}(r, c) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(r, \hat{fp}) \mapsto c], \hat{\kappa}, \hat{t}\rangle \\
\langle \texttt{throw}^{\ell}(r) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}\rangle &\longmapsto \langle \mathcal{S}(\ell'), fp', \hat{\sigma} \sqcup [(\texttt{exn}, \hat{fp}') \mapsto \hat{\sigma}(r, \hat{fp})], \hat{\kappa}'\rangle \\
&\quad \text{where } (\ell', \hat{fp}', \hat{\kappa}') \in \widehat{\mathcal{H}}_{\hat{\sigma}}(\ell, \hat{fp}, \hat{\kappa}) \\
\langle \texttt{goto}(\ell) :: \vec{stmt}', \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \mathcal{S}(\ell), \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle \\
\langle \texttt{new-instance}(r, \tau) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(r, \hat{fp}) \mapsto o], \hat{\kappa}, \hat{t}\rangle \\
&\quad \text{where } o = \widehat{new}(\varsigma) \\
\langle \texttt{if-eq}(r, r', \ell) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \mathcal{S}(\ell), \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle \\
&\quad \text{if } \exists v_1 \in \hat{\sigma}(r, \hat{fp}), \exists v_2 \in \hat{\sigma}(r', \hat{fp}). v_1 = v_2 \\
\langle \texttt{if-eq}(r, r', \ell) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle \\
&\quad \text{if } \exists v_1 \in \hat{\sigma}(r, \hat{fp}), \exists v_2 \in \hat{\sigma}(r', \hat{fp}). v_1 \neq v_2 \\
\langle \texttt{iget}(r_d, r_s, \mathit{field}) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(r_d, \hat{fp}) \mapsto \hat{\sigma}(a)], \hat{\kappa}, \hat{t}\rangle \\
&\quad \text{where } \hat{\sigma}(r_s, fp) \ni o \text{ and } o.\mathit{field} = a \\
\langle \texttt{iput}(r_v, r_s, \mathit{field}) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [a \mapsto \hat{\sigma}(r_v, \hat{fp})], \hat{\kappa}, \hat{t}\rangle \\
&\quad \text{where } \hat{\sigma}(r_s, \hat{fp}) \ni o \text{ and } o.\mathit{field} = a \\
\langle \texttt{invoke-direct}(r_0, \ldots, r_n, id) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \mathcal{M}(id), \hat{fp}', \hat{\sigma}'', \mathbf{fnk}(\vec{stmt}, \hat{fp}, \hat{a}_\kappa), \hat{t}'\rangle \\
&\quad \text{where } \hat{\sigma}'' = \hat{\sigma}' \sqcup [(0, \hat{fp}') \mapsto \sigma(r_0, \hat{fp}), \ldots, (n, \hat{fp}') \mapsto \sigma(r_n, \hat{fp})] \\
&\qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_\kappa \mapsto \hat{\kappa}] \\
&\qquad \hat{fp}' = \widehat{alloc}(\hat{\varsigma}) \\
&\qquad \hat{a}_\kappa = \widehat{allock}(\hat{\varsigma}) \\
&\qquad \hat{t}' = \widehat{tick}(\hat{t}) \\
\langle \texttt{invoke-virtual}(r_0, \ldots, r_n, id) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}\rangle &\longmapsto \langle \mathcal{V}(id, v), \hat{fp}', \hat{\sigma}'', \mathbf{fnk}(\vec{stmt}, \hat{fp}, \hat{\kappa}), \hat{t}'\rangle \text{ if } v \in \hat{\sigma}(r_0, \hat{fp}) \\
&\quad \text{where } \hat{\sigma}'' = \hat{\sigma}' \sqcup [(0, fp') \mapsto \hat{\sigma}(r_0, \hat{fp}), \ldots, (n, \hat{fp}') \mapsto \sigma(r_n, \hat{fp})] \\
&\qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_\kappa \mapsto \hat{\kappa}] \\
&\qquad \hat{fp}' = \widehat{alloc}(\hat{\varsigma}) \\
&\qquad \hat{a}_\kappa = \widehat{allock}(\hat{\varsigma}) \\
&\qquad \hat{t}' = \widehat{tick}(\hat{t}) \\
\langle \mathit{unop}(r_d, r_s) :: \vec{stmt}, \hat{fp}, \hat{\sigma}, \hat{\kappa}\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(r_d, \hat{fp}) \mapsto v], \hat{\kappa}\rangle \\
&\quad \text{where } v \in \hat{\delta}(\mathit{unop}, \sigma(r_s, fp)) \\
\langle \mathit{binop}(r_d, r_1, r_2) :: \vec{stmt}, fp, \sigma, \kappa\rangle &\longmapsto \langle \vec{stmt}, \hat{fp}, \hat{\sigma} \sqcup [(r_d, \hat{fp}) \mapsto v], \hat{\kappa}\rangle \\
&\quad \text{where } v \in \hat{\delta}(\mathit{binop}, \hat{\sigma}(r_1, \hat{fp}), \hat{\sigma}(r_2, \hat{fp}))
\end{aligned}
$$

Soundness

Soundness

$$
\begin{tikzcd}
\varsigma \arrow[r] \arrow[d, leftrightarrow] & \varsigma' \arrow[d, leftrightarrow] \\
\hat{\varsigma} \arrow[r, dashed] & \hat{\varsigma}'
\end{tikzcd}
$$

JavaScript

Pushdown Abstractions of JavaScript

David Van Horn[1] and Matthew Might[2]

[1] Northeastern University, Boston, Massachusetts, USA
[2] University of Utah, Salt Lake City, Utah, USA

Abstract. We design a family of program analyses for JavaScript that make *no approximation* in matching calls with returns, exceptions with handlers, and breaks with labels. We do so by starting from an established reduction semantics for JavaScript and systematically deriving its intensional abstract interpretation. Our first step is to transform the semantics into an equivalent low-level abstract machine: the JavaScript Abstract Machine (JAM). We then give an infinite-state yet decidable pushdown machine whose stack precisely models the structure of the concrete program stack. The precise model of stack structure in turn confers *precise* control-flow analysis even in the presence of control effects, such as exceptions and finally blocks. We give pushdown generalizations of traditional forms of analysis such as k-CFA, and prove the pushdown framework for abstract interpretation is sound and computable.

1 Introduction

JavaScript is the dominant language of the web, making it the most ubiquitous programming language in use today. Beyond the browser, it is increasingly important as a general-purpose language, as a server-side scripting language, and as an embedded scripting language—notably, Java 6 includes support for scripting applications via the `javax.script` package, and the JDK ships with the Mozilla Rhino JavaScript engine. Due to its ubiquity, JavaScript has become the target language for an array of compilers for languages such as C#, Java, Ruby, and others, making JavaScript a widely used "assembly language." As JavaScript cements its foundational role, the importance of robust static reasoning tools for that foundation grows.

Motivated by the desire to handle non-local control effects such as exceptions and `finally` precisely, we will depart from standard practice in higher-order program analysis to derive an *infinite*-state yet decidable pushdown abstraction from our original abstract machine. The stack of the pushdown abstract interpreter *exactly* models the stack of the original abstract machine with no loss of structure—approximation is inflicted on only the control states. This pushdown framework offers a degree of precision in reasoning about control inaccessible to previous analyzers.

Pushdown analysis is an alternative paradigm for the analysis of higher-order programs in which the run-time program stack is precisely modeled with the stack of a pushdown system [40, 14]. Consequently, a pushdown analysis can

$$
\begin{array}{rcl}
s & \in & \mathit{String} \\
n & \in & \mathit{Number} \\
a & \in & \mathit{Address} \\
x & \in & \mathit{Variable}
\end{array}
$$

$$
\begin{array}{rcl}
e, f, g & ::= & x \mid s \mid n \mid a \mid \textbf{true} \mid \textbf{false} \mid \textbf{undef} \mid \textbf{null} \\
& \mid & \textbf{fun}(\overline{x})\ \{\ e\ \} \mid \{\overline{s\,\texttt{:}\,e}\} \mid \textbf{let}\ (x \texttt{=} e)\ e \mid e(\overline{e}) \mid e[e] \\
& \mid & e[e] \texttt{=} e \mid \textbf{del}\ e[e] \mid e \texttt{=} e \mid \textbf{ref}\ e \mid \textbf{deref}\ e \\
& \mid & \textbf{if}(e)\{e\}\{e\} \mid e\texttt{;}e \mid \textbf{while}(e)\{e\} \\
& \mid & \ell\texttt{:}\{\ e\ \} \mid \textbf{break}\ \ell\ e \mid \textbf{try}\ \{e\}\ \textbf{catch}\ (x)\{e\} \\
& \mid & \textbf{try}\ \{e\}\ \textbf{finally}\ \{e\} \mid \textbf{throw}\ e \mid op(\overline{e}) \\
t, u, v & ::= & s \mid n \mid a \mid \textbf{true} \mid \textbf{false} \mid \textbf{undef} \mid \textbf{null} \mid (\textbf{fun}(\overline{x})\ \{\ e\ \}, \rho) \mid \{\overline{s\,\texttt{:}\,v}\} \\
c, d & ::= & (e, \rho) \mid \{\overline{s\,\texttt{:}\,c}\} \mid \textbf{let}\ (x \texttt{=} c)\ c \mid c(\overline{c}) \mid c[c] \\
& \mid & c[c] \texttt{=} c \mid \textbf{del}\ c[c] \mid c \texttt{=} c \mid \textbf{ref}\ c \mid \textbf{deref}\ c \\
& \mid & \textbf{if}(c)\{c\}\{c\} \mid c\texttt{;}c \mid \textbf{while}(c)\{c\} \\
& \mid & \ell\texttt{:}\{\ c\ \} \mid \textbf{break}\ \ell\ c \mid \textbf{try}\ \{c\}\ \textbf{catch}\ (x)\{c\} \\
& \mid & \textbf{try}\ \{c\}\ \textbf{finally}\ \{c\} \mid \textbf{throw}\ c \mid op(\overline{c})
\end{array}
$$

Fig. 1: Syntax of λ_{JS}

$$
\begin{array}{lll}
\langle \sigma, (x,\rho), E\rangle_{ap} & \longmapsto & \langle \sigma, v, E\rangle_{co} \text{ if } v \in \mathit{get}(\sigma, a) \\
\langle \sigma, \textsf{let } (x = v)\ c, E\rangle_{ap} & \longmapsto & \langle \mathit{put}(\sigma, a, v), (e, \rho[x \mapsto a]), E\rangle_{ev} \\
 & & \text{where } a = \mathit{alloc}(\varsigma) \\
\langle \sigma, (\textsf{fun}(\overline{x})\ \{\ e\ \}, \rho)(\overline{v}), E\rangle_{ap} & \longmapsto & \langle \mathit{put}(\sigma, \overline{a}, \overline{v}), (e, \rho[\overline{x} \mapsto \overline{a}]), E\rangle_{ev} \\
 & & \text{if } |\overline{x}| = |\overline{v}|,\ \text{where } \overline{a} = \mathit{alloc}(\varsigma) \\
\langle \sigma, \{\overline{s:v}, s_i : v, \overline{s:v}'\}[s_i], E\rangle_{ap} & \longmapsto & \langle \sigma, v, E\rangle_{co} \\
\langle \sigma, \{\overline{s:v}\}[s_x], E\rangle_{ap} & \longmapsto & \langle \sigma, \textsf{undef}, E\rangle_{co} \text{ if } s_x \notin \overline{s} \\
\langle \sigma, \{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i] = v, E\rangle_{ap} & \longmapsto & \langle \sigma, \{\overline{s:v}, s_i : v, \overline{s:v}'\}, E\rangle_{co} \\
\langle \sigma, \textsf{del } \{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i], E\rangle_{ap} & \longmapsto & \langle \sigma, \{\overline{s:v}, \overline{s:v}'\}, E\rangle_{co} \\
\langle \sigma, \textsf{del } \{\overline{s:v}\}[s_x], E\rangle_{ap} & \longmapsto & \langle \sigma, \{\overline{s:v}\}, E\rangle_{co} \text{ if } s_x \notin \overline{s} \\
\langle \sigma, \textsf{if}(\textsf{true})\{c\}\{d\}, E\rangle_{ap} & \longmapsto & \langle \sigma, c, E\rangle_{ev} \\
\langle \sigma, \textsf{if}(\textsf{false})\{c\}\{d\}, E\rangle_{ap} & \longmapsto & \langle \sigma, d, E\rangle_{ev} \\
\langle \sigma, \mathit{op}_n(v_1, \ldots, v_n), E\rangle_{co} & \longmapsto & \langle \sigma, v, E\rangle_{co} \text{ if } \delta(\mathit{op}_n, v_1, \ldots, v_n) = v \\
\langle \sigma, \textsf{ref } v, E\rangle_{ap} & \longmapsto & \langle \mathit{put}(\sigma, a, v), a, E\rangle_{co} \\
 & & \text{where } a = \mathit{alloc}(\varsigma) \\
\langle \sigma, \textsf{deref } a, E\rangle_{ap} & \longmapsto & \langle \sigma, v, E\rangle_{co} \text{ if } v \in \mathit{get}(\sigma, a) \\
\langle \sigma, a = v, E\rangle_{ap} & \longmapsto & \langle \mathit{put}(\sigma, a, v), v, E\rangle_{co} \\
\langle \sigma, \textsf{throw } v, \mathbf{nil}\rangle & \longmapsto & \langle \textsf{err } v, \sigma\rangle \\
\langle \sigma, \textsf{throw } v, \textsf{try } \{\bullet\} \textsf{ catch } (x)\{(e,\rho)\} :: E\rangle_{ap} & \longmapsto & \langle \mathit{put}(\sigma, a, v), (e, \rho[x \mapsto a]), E\rangle_{ev} \\
 & & \text{where } a = \mathit{alloc}(\varsigma) \\
\langle \sigma, \textsf{throw } v, \textsf{try } \{\bullet\} \textsf{ finally } \{c\} :: E\rangle_{ap} & \longmapsto & \langle \sigma, c; \textsf{throw } v, E\rangle_{ev} \\
\langle \sigma, \textsf{throw } v, \ell : \{\ \bullet\ \} :: E\rangle_{ap} & \longmapsto & \langle \sigma, \textsf{throw } v, E\rangle_{ap} \\
\langle \sigma, \textsf{throw } v, \mathcal{C} :: E\rangle_{ap} & \longmapsto & \langle \sigma, \textsf{throw } v, E\rangle_{ap} \\
\langle \sigma, \textsf{break } \ell\ v, \textsf{try } \{x\} \textsf{ catch } (\bullet)\{c\} :: E\rangle_{ap} & \longmapsto & \langle \sigma, \textsf{break } \ell\ v, E\rangle_{ev} \\
\langle \sigma, \textsf{break } \ell\ v, \textsf{try } \{\bullet\} \textsf{ finally } \{c\} :: E\rangle_{ap} & \longmapsto & \langle \sigma, c; \textsf{break } \ell\ v, E\rangle_{ev} \\
\langle \sigma, \textsf{break } \ell\ v, \ell : \{\ \bullet\ \} :: E\rangle_{ap} & \longmapsto & \langle \sigma, v, E\rangle_{co} \\
\langle \sigma, \textsf{break } \ell\ v, \ell' : \{\ \bullet\ \} :: E\rangle_{ap} & \longmapsto & \langle \sigma, v, E\rangle_{co} \text{ if } \ell \neq \ell' \\
\langle \sigma, \textsf{break } \ell\ v, \mathcal{C} :: E\rangle_{ap} & \longmapsto & \langle \sigma, \textsf{break } \ell\ v, E\rangle_{ap}
\end{array}
$$

Fig. 6: Application transitions

Shape analysis

Environment analysis

Cultural Differences

Theory, soundness first.

Efficiency, eventually.

Polynomial 0CFA: 7 years

Subcubic 0CFA: 14 years

Polynomial *k*CFA: 20 years

Never restrict.

Microbenchmarks.

Decidable

↓

Polynomial

↓

Feasible

↓

Evaluated

The *end* of pointer analysis?

The beginning of control-flow!

Danke!

matt.might.net

matt.might.net

Danke!

matt.might.net

Danke!