Shape analysis of higher-order programs: A colorless green idea?

Matthew Might
University of Utah
matt.might.net
www.ucombinator.org

Is shape analysis of higher-order programs meaningful?

What is **shape analysis** of higher-order programs?

It’s still **shape analysis**, but with different words.

address :: binding

structure :: binding environment

heap :: value environment

shape analysis :: environment analysis

Why bother?

Top-down reason: Need to move beyond CFAs.

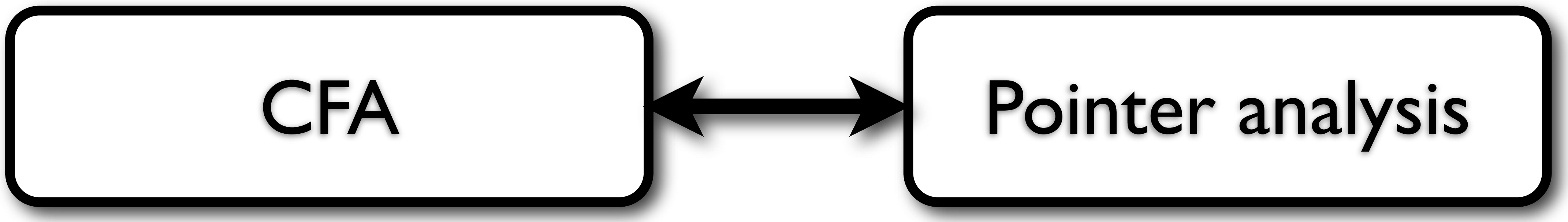
Bottom-up reason

Bottom-up reason

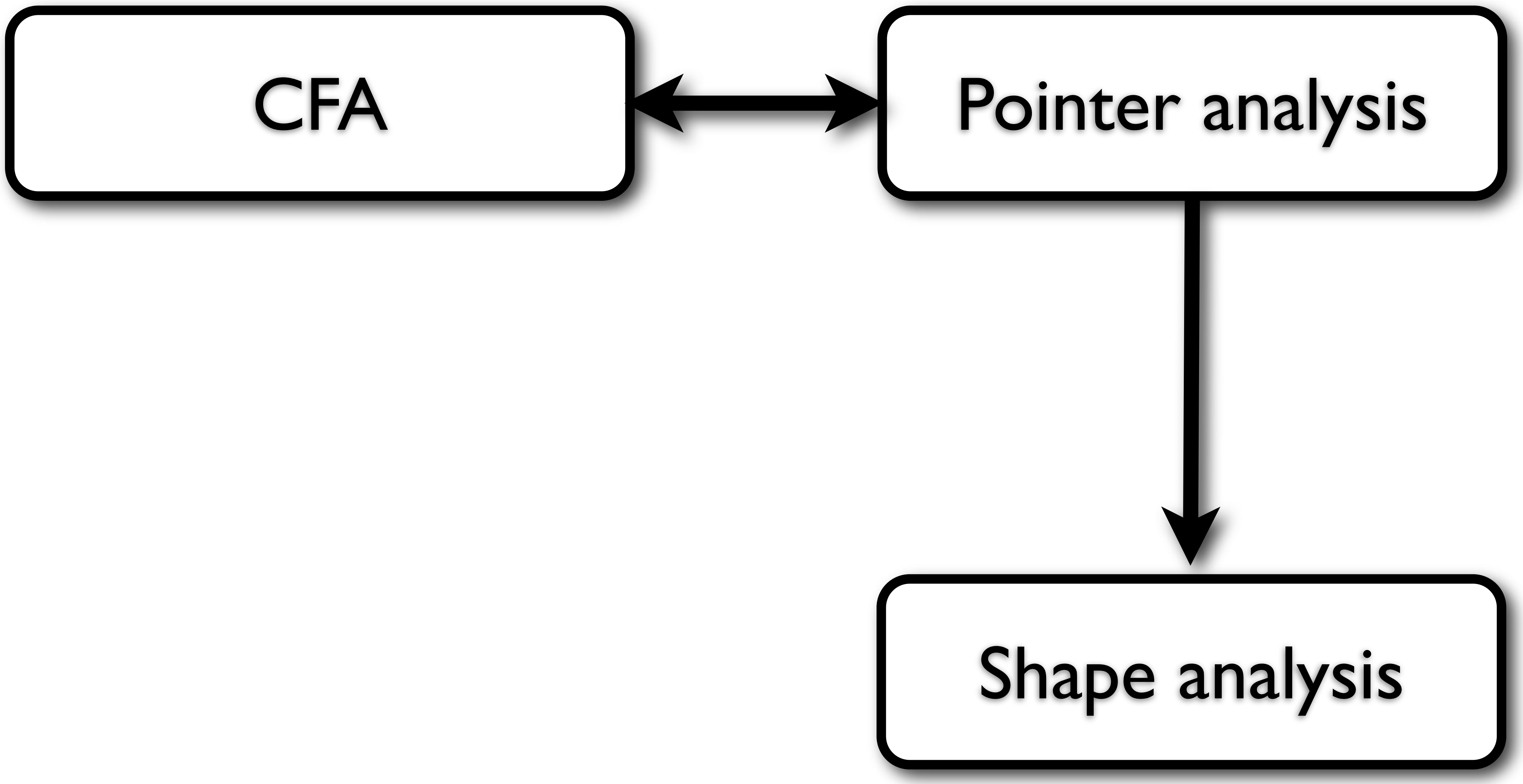
CFA

Pointer analysis

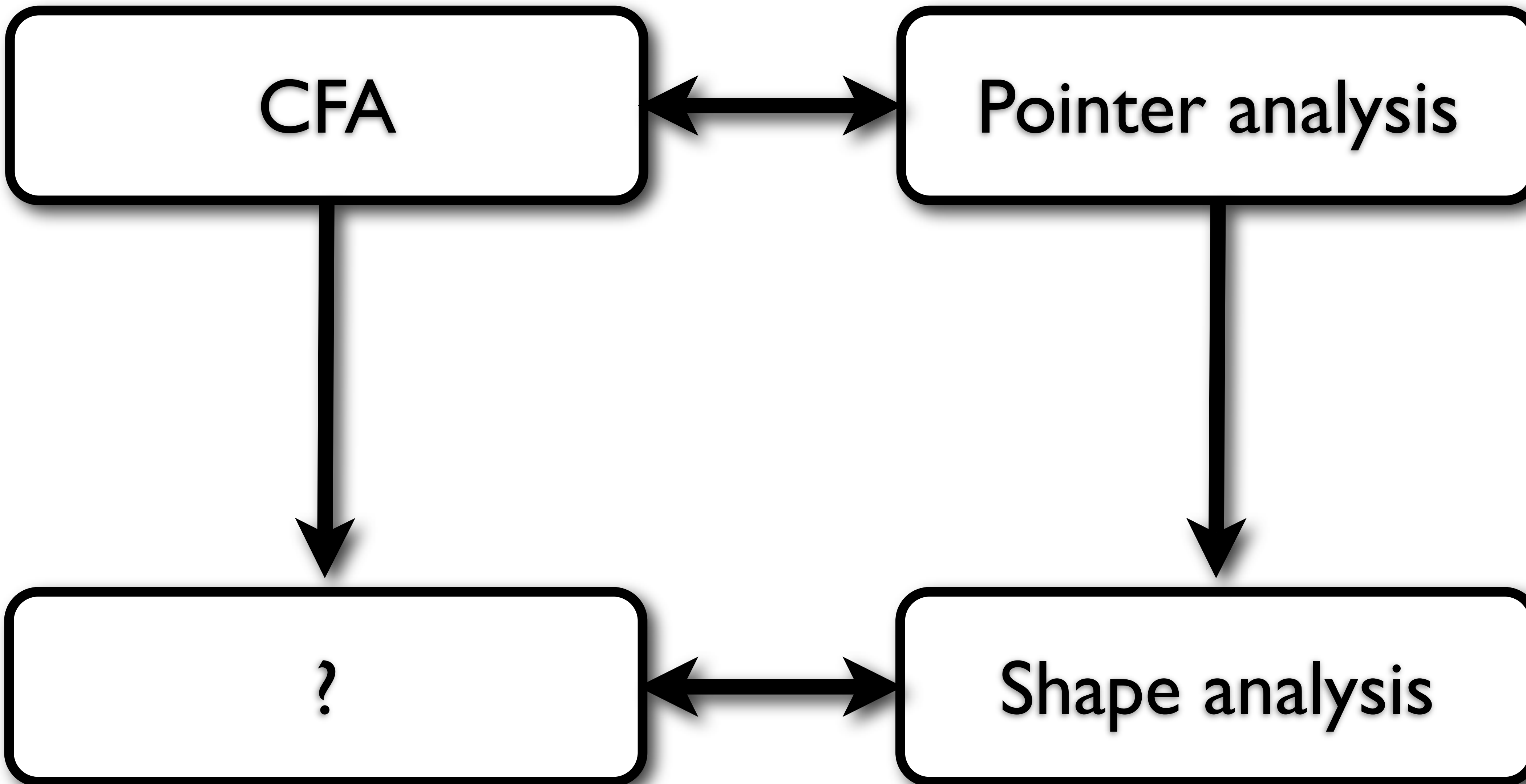
Bottom-up reason



Bottom-up reason



Bottom-up reason



What is “higher order?”

The essence of higher-order: Lambda calculus.

Syntax

Variables; function abstractions; applications.

Syntax

Variables; function abstractions; applications.

`v` `(λ (v) e)` $(e_1\ e_2)$

Semantics

$$Value = Value \to Value$$

No integers.

No floats.

No arrays.

No structs.

No pointers.

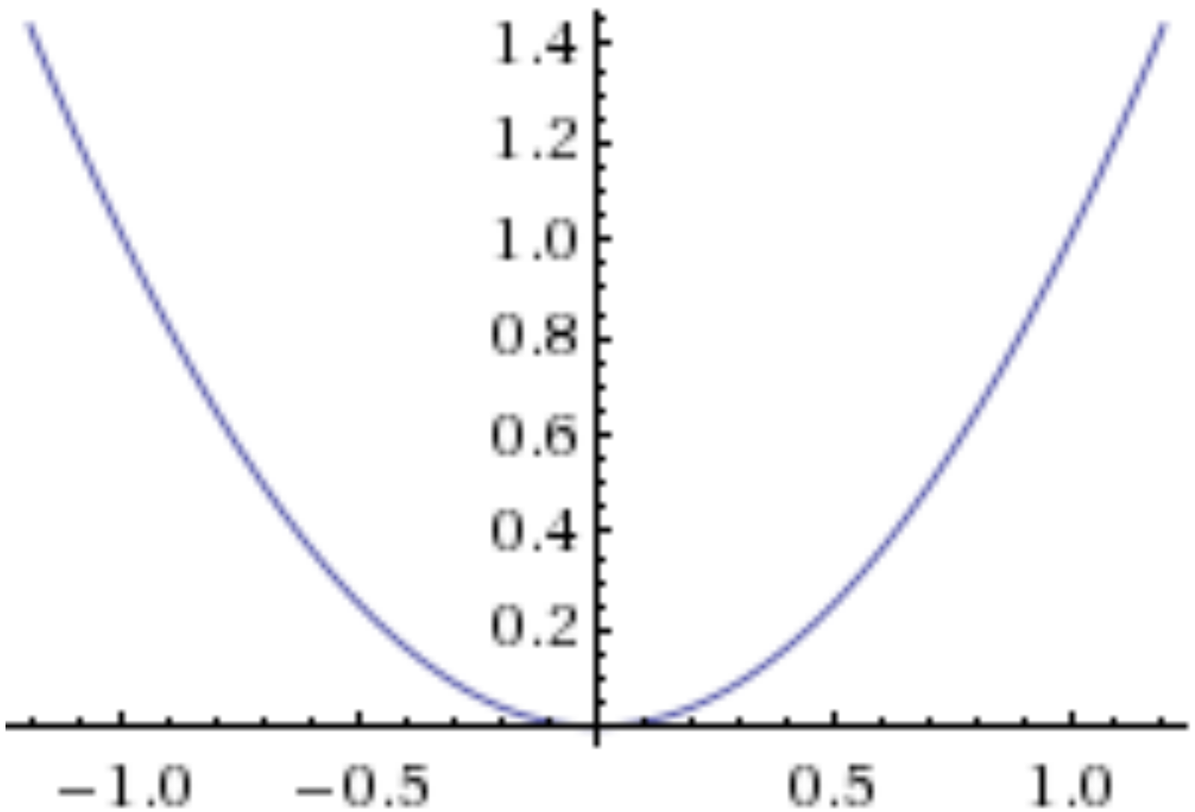
No mutation.

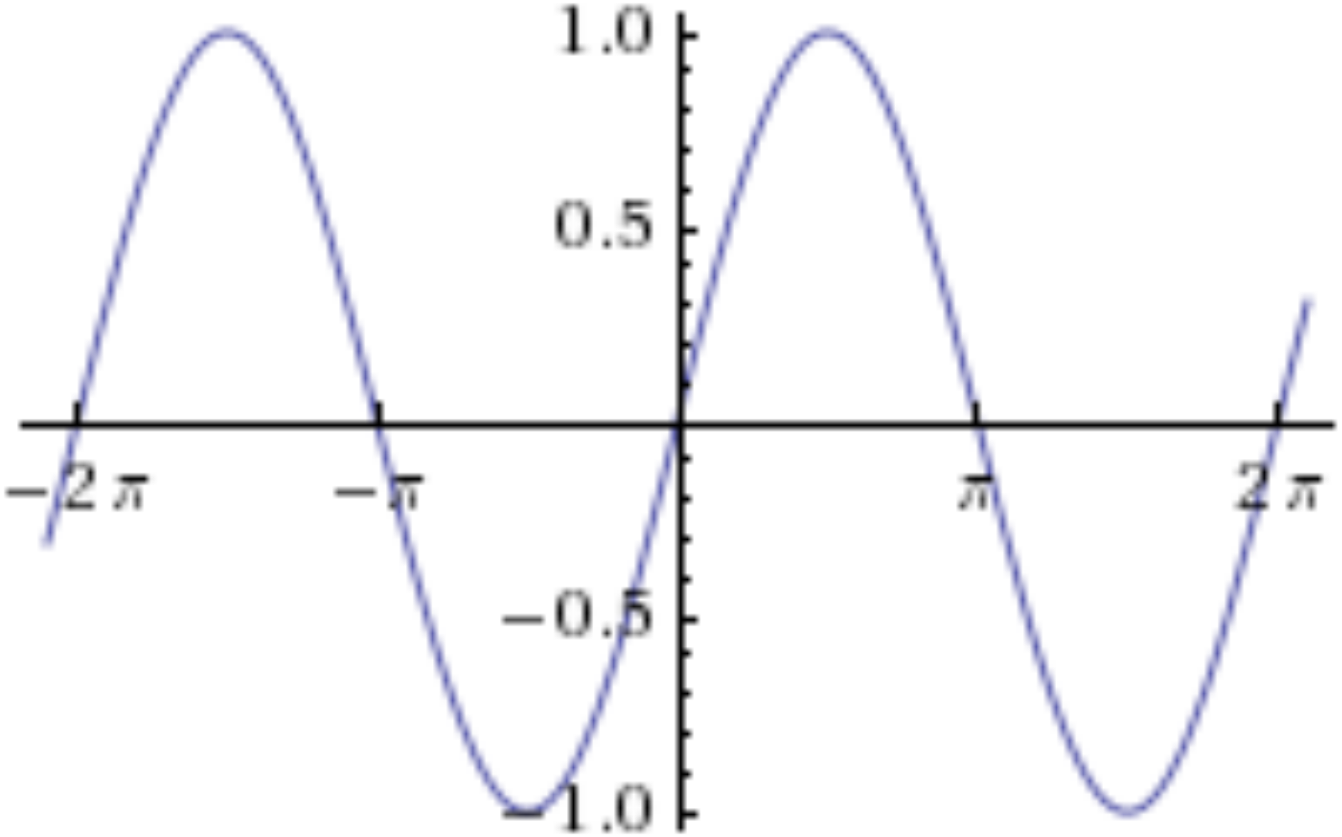
Lambda-calculus lacks linked, mutable, dynamic structures.

Shape analysis studies linked, mutable, dynamic structures.

So, does shape analysis of the λ-calculus mean anything?

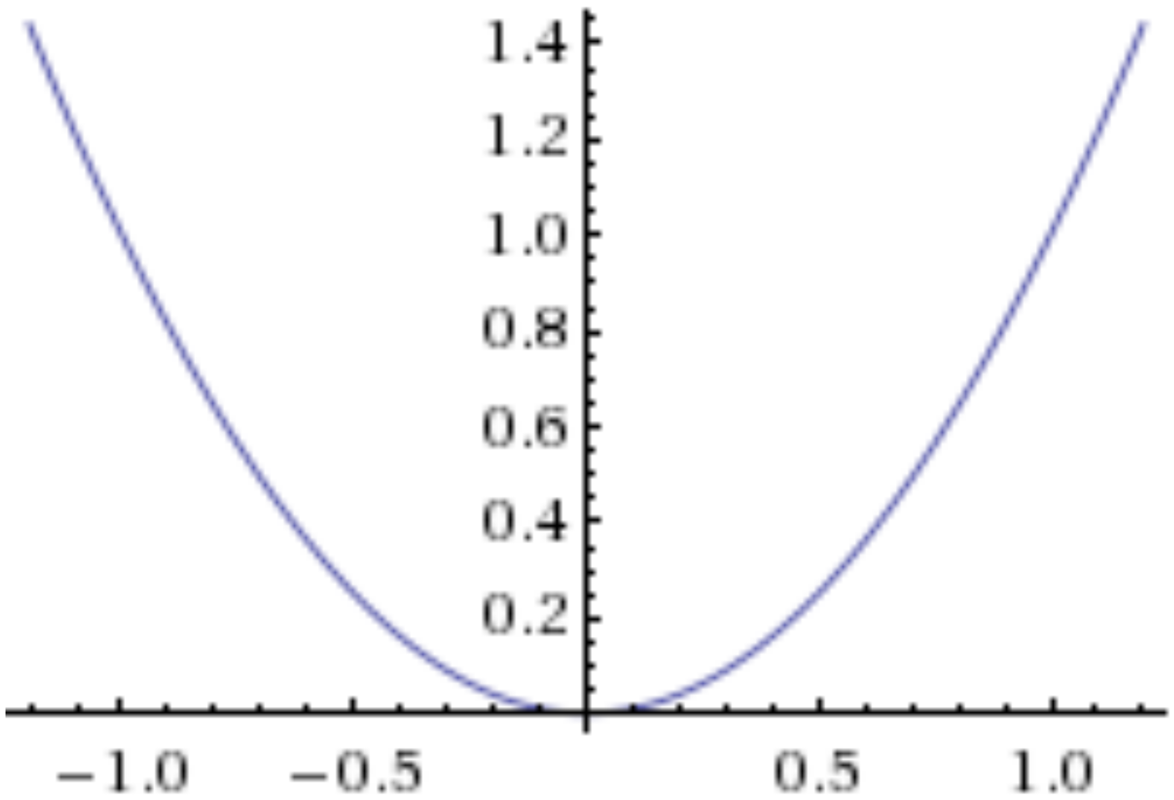
Do functions have shape?


1.4
1.2
1.0
0.8
0.6
0.4
0.2
−1.0
−0.5
0.5
1.0

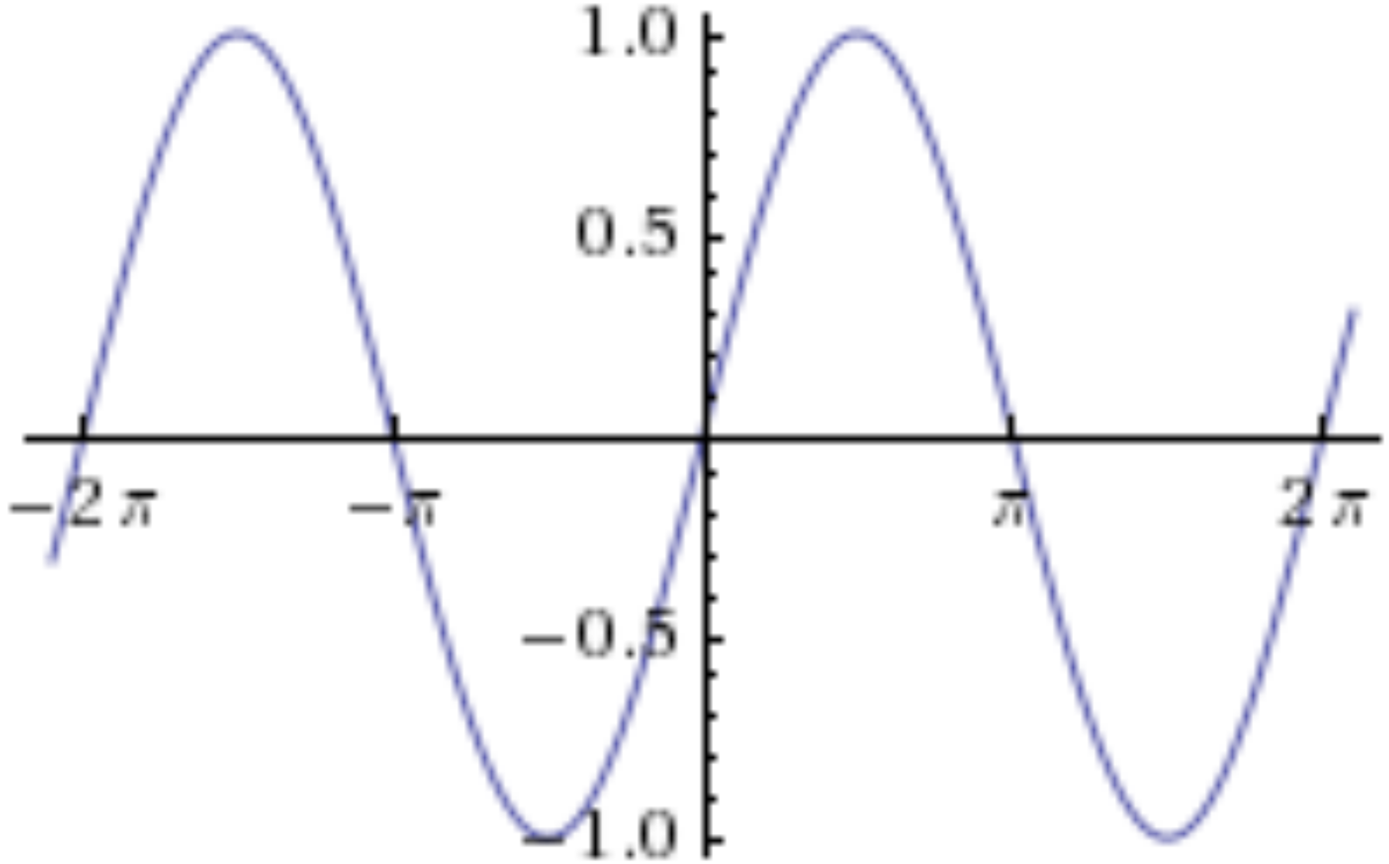

1.0
0.5
−0.5
−1.0
−2 π
−π
π
2 π

What determines the shape of these functions?

Parameters.



$$f(x) = \mathbf{a}x^2 + \mathbf{b}x + \mathbf{c}$$



$$f(x) = \mathbf{A} \sin(\boldsymbol{\omega} x + \boldsymbol{\varphi})$$

$$f(x) = \mathbf{A} \sin(\boldsymbol{\omega} x + \boldsymbol{\varphi})$$

$$f = \lambda x. \mathbf{A} \sin(\boldsymbol{\omega} x + \boldsymbol{\varphi})$$

Free variables determine function shape.

What determines the value of free variables?

Environments.

Function = Closure = Lambda-term + Environment

$$\lambda x.\mathbf{A}\sin(\boldsymbol{\omega} x + \boldsymbol{\varphi})$$

$$(\lambda x.\mathbf{A}\sin(\boldsymbol{\omega} x + \boldsymbol{\varphi}),[\mathbf{A}=1,\boldsymbol{\omega}=1,\boldsymbol{\varphi}={}^{\pi}/_{2}])$$

cos

Environments are linked, mutable, dynamic data structures.

Shape analysis studies linked, mutable, dynamic structures.

Shape analysis of the λ-calculus is environment analysis.

Shape analysis determines the meaning of functions.

Same tools apply

- Singleton abstraction
- Relational abstraction
- Heap/shape predicates

But first,
do environments matter?

Application: Inlining

```
(let ((f (lambda (x h)
           (if x
               (h)
               (lambda () x)))))
  (f #t (f #f nil)))
```

Application: Inlining

```
(let ((f (lambda (x h)
           (if x
               (h)
               (lambda () x)))))
  (f #t (f #f nil)))
```

Environment in closure must match environment at call.

Special environment problem

"Does $env_1(\mathsf{X}) = env_2(\mathsf{X})$?"

Application: Rematerialization

```
(f)

(lambda () z)
```

Compiler wants to inline, but `z` is out of scope at the call!

Application: Rematerialization

```
((lambda () y))
```

```
(lambda () z)
```

Compiler wants to inline, but `z` is out of scope at the call!

General environment problem

"Does $env_1(\mathsf{z}) = env_2(\mathsf{y})$?"

Approach: Build general solution atop special solution.

Starting point:
k-CFA for CPS

In CPS, all calls must be tail calls.

Functions never return, so no stack required.

Small-step state-space

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

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

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

Small-step state-space

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

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

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

Split environments (Shivers, 1991)

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

Split environments (Shivers, 1991)

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

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

Split environments (Shivers, 1991)

$$\beta \in BEnv = \mathsf{Var} \rightharpoonup Bind$$
$$ve \in VEnv = Bind \rightharpoonup Clo$$

$$
\begin{aligned}
\varsigma \in \mathit{State} &= \mathsf{Call} \times \mathit{BEnv} \times \mathit{VEnv} \\
\beta \in \mathit{BEnv} &= \mathsf{Var} \rightharpoonup \mathit{Bind} \\
\mathit{ve} \in \mathit{VEnv} &= \mathit{Bind} \rightharpoonup \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{BEnv} \\
b \in \mathit{Bind} & \text{ is some infinite set}
\end{aligned}
$$

$$\varsigma \in \mathit{State} = \mathsf{PC} \times \mathit{Struct} \times \mathit{Heap}$$

$$s \in \mathit{Struct} = \mathsf{Var} \rightharpoonup \mathit{Addr}$$

$$h \in \mathit{Heap} = \mathit{Addr} \rightharpoonup \mathit{Tagged}$$

$$t \in \mathit{Tagged} = \mathsf{Type} \times \mathit{Struct}$$

$$a \in \mathit{Addr} \text{ is some infinite set}$$

Solving the special problem

Special problem

$\hat{\beta}$ $\hat{\beta}'$

Special problem

$$
\begin{tikzcd}
\hat{\beta} & & \hat{\beta}' \\
\beta \arrow[u, "\alpha"] & & \beta' \arrow[u, "\alpha"']
\end{tikzcd}
$$

Special problem

$$\begin{tikzcd}
\hat{\beta}(v) & = & \hat{\beta}'(v) \\
\beta \arrow[u, "\alpha"] & & \beta' \arrow[u, "\alpha"']
\end{tikzcd}$$

Special problem

$$\begin{tikzcd}
\hat{\beta}(v) \arrow[r, phantom, "="] & \hat{\beta}'(v) \\
\beta(v) \arrow[u, "\alpha"] \arrow[r, phantom, "="] & \beta'(v) \arrow[u, "\alpha"'] \text{ ?}
\end{tikzcd}$$

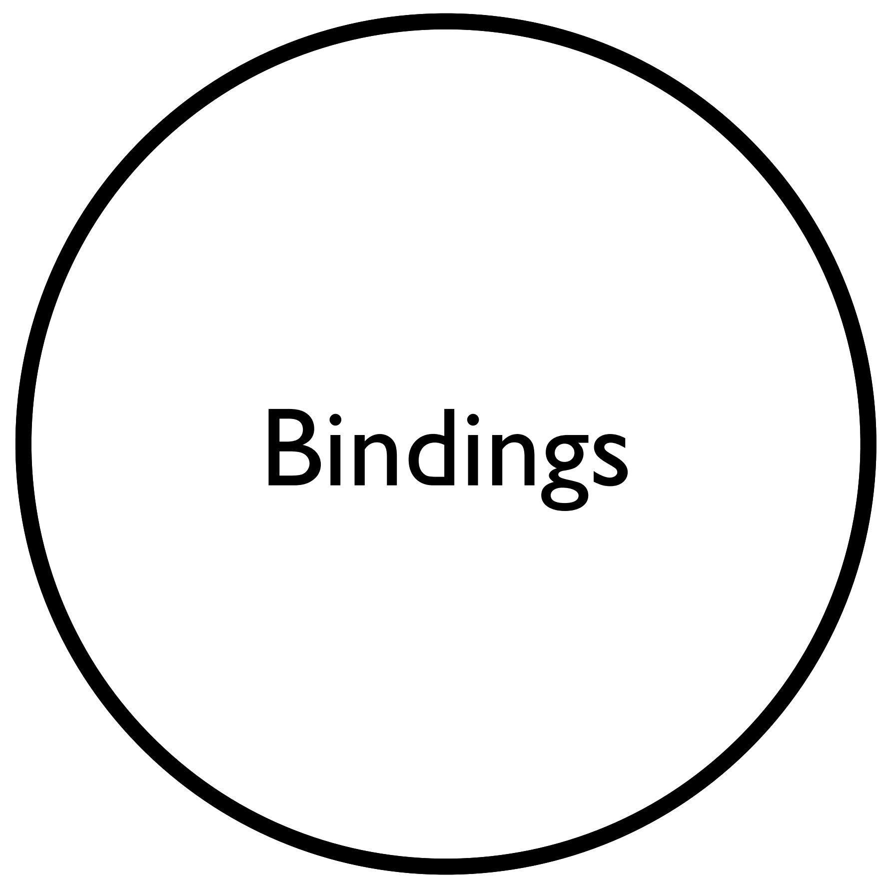
When does $\alpha(b) = \alpha(b')$ imply $b = b'$?

When the abstract bindings are singleton abstractions.

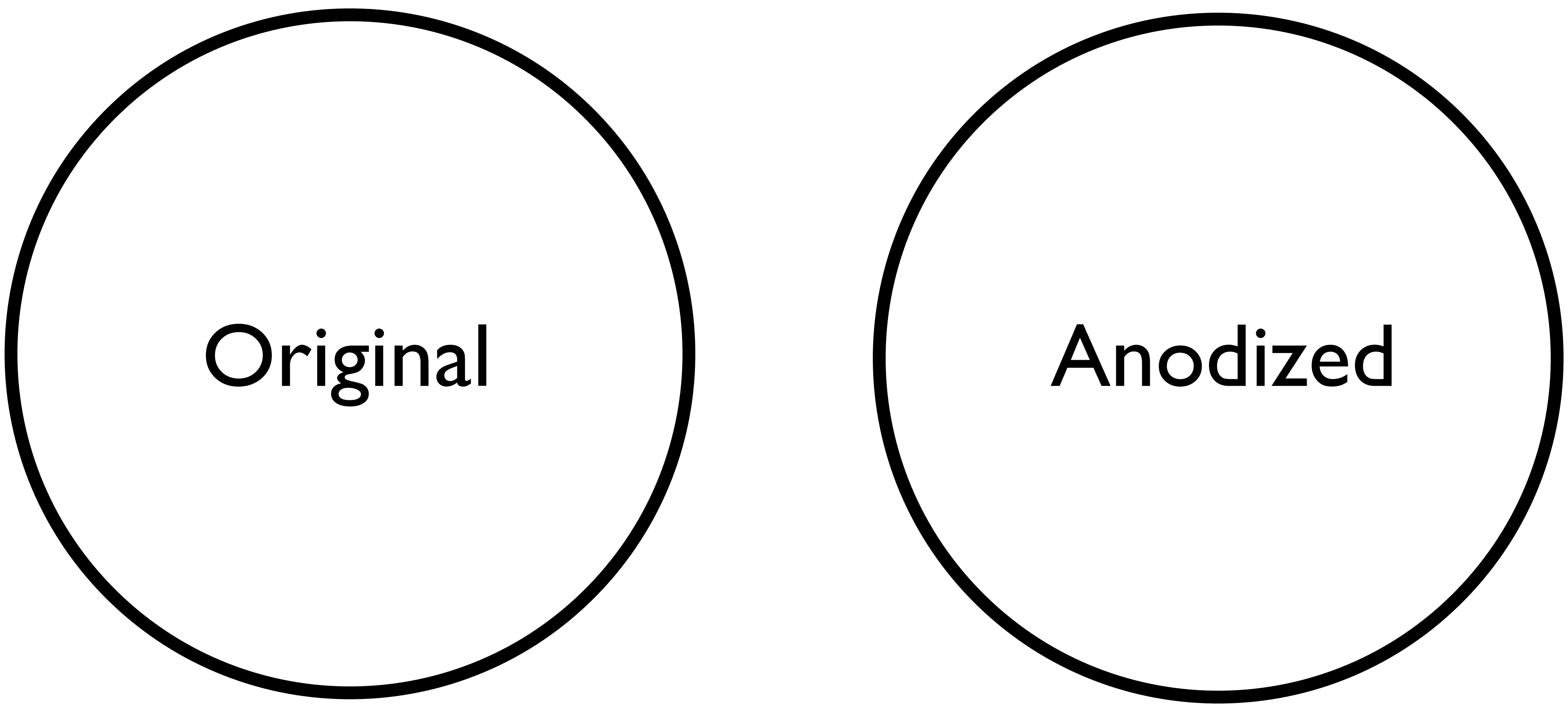
A singleton abstraction has only one concrete constituent.

Next step: Engineer a singleton abstraction into semantics.

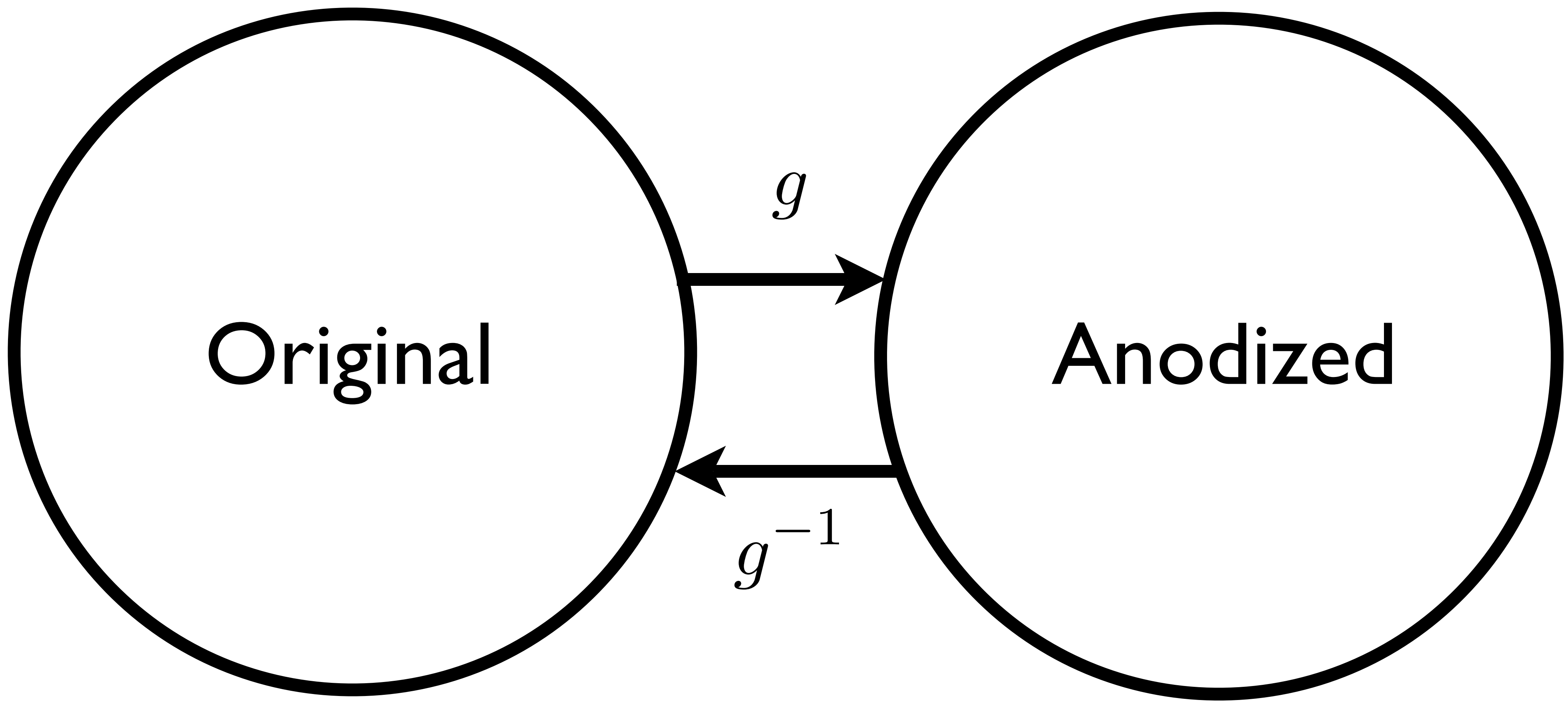
Anodized bindings



Anodized bindings



Anodized bindings



Anodization constraint

If $g(b)$ and $g(b')$ are reachable and $\alpha(b) = \alpha(b')$, then $b = b'$.

Policy example: Recency (Balakrishnan & Reps, 2006)

Anodize most-recently allocated binding.

Solving the general problem

What implies $ve(b) = ve(b')$?

Fact 1: $ve(b) = ve(b)$

Fact 2: $ve(b) = ve(b')$ and $ve(b') = ve(b'')$ implies $ve(b) = ve(b'')$.

When will we know that $ve(b') = ve(b'')$?

When b is bound to b' during function call.

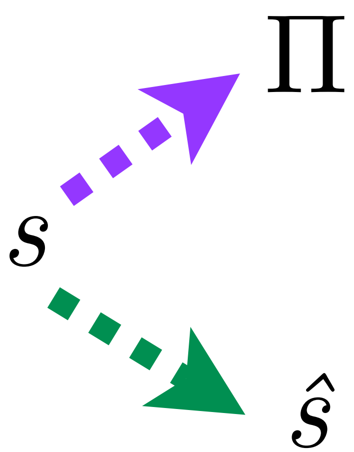
When `(`f `x)` calls `(λ (v)` $call$`)`, we know $ve(\beta(\mathtt{x})) = ve(\beta'(\mathtt{v}))$.

Solution: Track binding invariants as separate abstraction.

Binding invariants

$$\Pi \in State_{\equiv} \subseteq \widehat{Bind} \times \widehat{Bind}$$

Relational



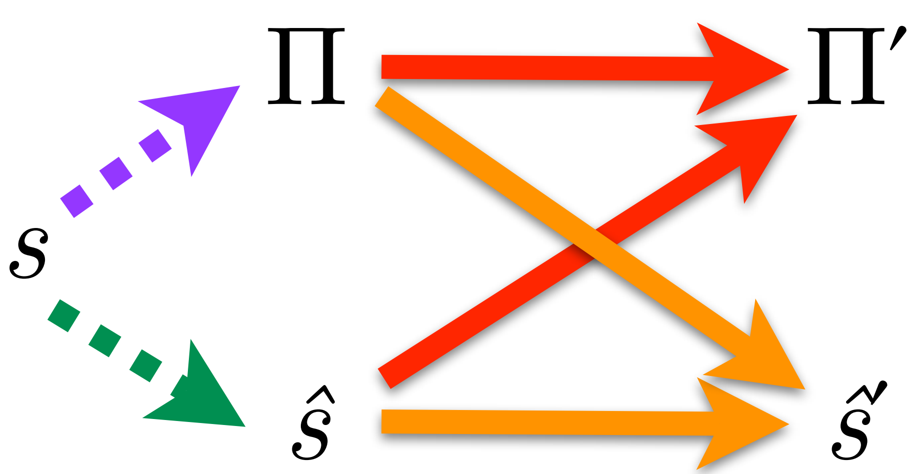
Mechanical

Relational

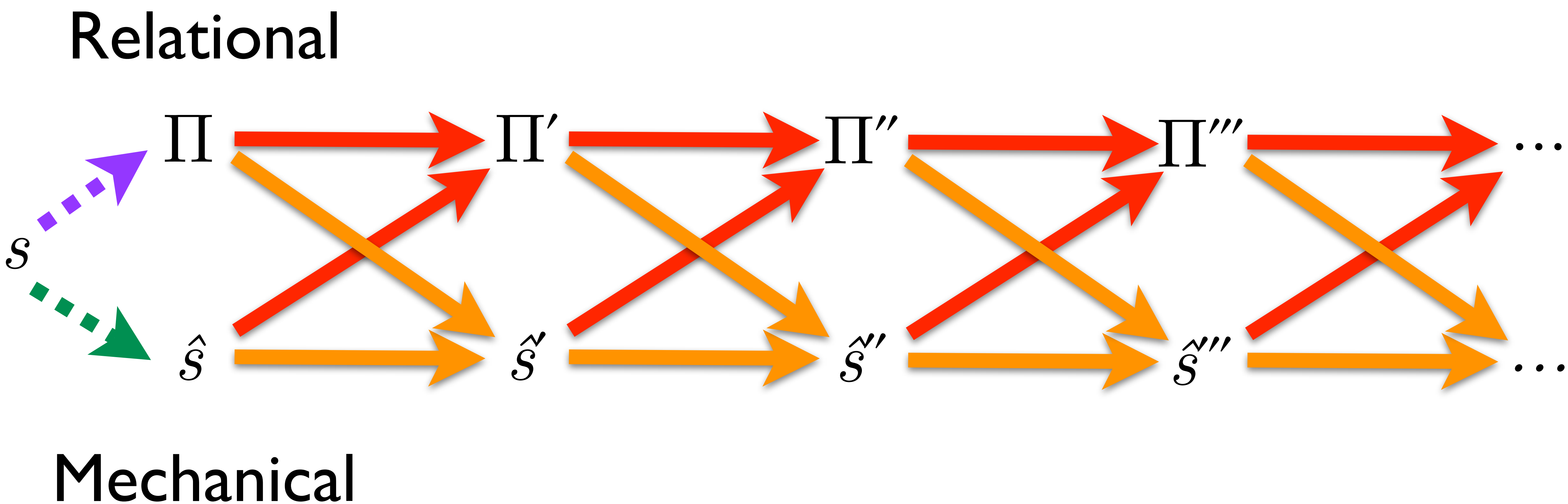
$$\begin{tikzcd} & \Pi \arrow[r] & \Pi' \\ s \arrow[ur, dashed] \arrow[dr, dashed] & & \\ & \hat{s} \arrow[r] & \hat{s}' \end{tikzcd}$$

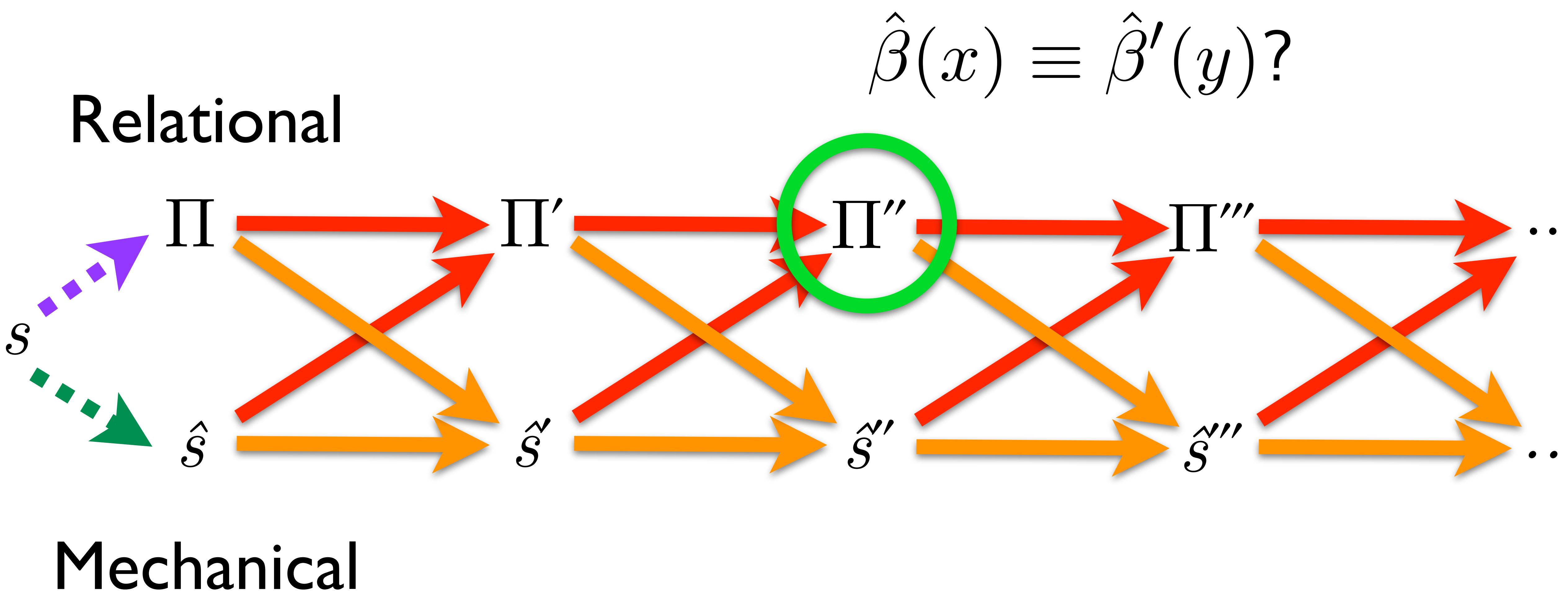
Mechanical

Relational



Mechanical


Relational
Π
Π'
Π''
Π'''
$\cdots$
s
$\hat{s}$
$\hat{s}'$
$\hat{s}''$
$\hat{s}'''$
$\cdots$
Mechanical


$\hat{\beta}(x) \equiv \hat{\beta}'(y)?$
Relational
Π
Π'
Π''
Π'''
$\dots$
s
$\hat{s}$
$\hat{\tilde{s}}'$
$\hat{\tilde{s}}''$
$\hat{\tilde{s}}'''$
$\dots$
Mechanical

Related work

- Cousot & Cousot, 1977, 1979, 1991, 1994.
- Sagiv, Reps, & Wilhelm, 2002.
- Ball et al., 2001.
- Hudak et al., 1985.
- Chase et al., 1990.
- Shivers, 1988, 1991.
- Jagannathan et al., 1998.

More in paper

Specific problem To determine the safety of inlining the lambda term *lam* at the call site $[\![$`(f ...)`$]\!]$, we need to know that for every environment ρ in which this call is evaluated, that $\rho[\![\mathtt{f}]\!] = (lam, \rho')$ and $\rho(v) = \rho'(v)$ for each free variable v in the term *lam*.[2]

More in paper

Specific problem To determine the safety of inlining the lambda term *lam* at the call site $[\![$`(f ...)`$]\!]$, we need to know that for every environment ρ in which this call is evaluated, that $\rho[\![\mathtt{f}]\!] = (lam, \rho')$ and $\rho(v) = \rho'(v)$ for each free variable v in the term *lam*.[2]

$$\eta(b) = \hat{b} \text{ iff } \eta(g(b)) = \hat{g}(\hat{b}).$$

$$\frac{\hat{\beta}(e_i) \in \widehat{Bind}_1 \qquad \hat{b}_i \in \widehat{Bind}_1}{\hat{\beta}(e_i) \equiv' \hat{b}_i,}$$

$$\begin{aligned}
\alpha^\eta(call, \beta, ve, t) &= (\alpha^\eta(V), \alpha^\eta(\beta), \alpha^\eta(ve), \eta(t)) \\
\alpha^\eta_{BEnv}(\beta) &= \lambda v.\eta(\beta(v)) \\
\alpha^\eta_{VEnv}(ve) &= \lambda \hat{b}. \bigsqcup_{\eta(b)=\hat{b}} \alpha^\eta(ve(b)) \\
\alpha^\eta_D(d) &= \{\alpha^\eta_{Val}(d)\} \\
\alpha^\eta_{Val}(lam, \beta) &= (lam, \alpha^\eta(\beta)).
\end{aligned}$$

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\
\hat{\beta} \in \widehat{BEnv} &= \mathsf{Var} \rightharpoonup \widehat{Bind} \\
\widehat{ve} \in \widehat{VEnv} &= \widehat{Bind} \to \hat{D} \\
\hat{d} \in \hat{D} &= \mathcal{P}(\widehat{Val}) \\
\widehat{val} \in \widehat{Val} &= \widehat{Clo} \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{BEnv}
\end{aligned}$$

$\hat{b} \in \widehat{Bind}$ is a **finite** set of bindings

$\hat{t} \in \widehat{Time}$ is a **finite** set of times

Theorem 4. *It is safe to rematerialize the expression e' in place of the expression e in the call site call iff for every reachable compound abstract state of the form $((call, \hat{\beta}'', \widehat{ve}, \hat{t}), \equiv)$, it is the case that $\hat{\mathcal{E}}(e', \hat{\beta}'', \widehat{ve}) = (lam', \hat{\beta}')$ and $\hat{\mathcal{E}}(e, \hat{\beta}'', \widehat{ve}) = (lam, \hat{\beta})$ and the relation $\sigma \subseteq \mathsf{Var} \times \mathsf{Var}$ is a substitution that unifies the free variables of lam' with lam and for each $(v', v) \in \sigma$, $\hat{\beta}'(v') \equiv \hat{\beta}(v)$.*

$$\begin{aligned}
g_B^{-1}(b) &= b \\
g_B^{-1}(g(b)) &= \begin{cases} b & \eta(b) = \eta(b') \text{ for some } g(b') \in B \\ g(b) & \text{otherwise} \end{cases} \\
g_B^{-1}(lam, \beta) &= (lam, g_B^{-1}(\beta)) \\
g_B^{-1}(\beta) &= \lambda v. g_B^{-1}(\beta(v)) \\
g_B^{-1}(ve) &= \lambda b. g_B^{-1}(ve(b)).
\end{aligned}$$

$$(([\![(f\ e_1 \ldots e_n)^\ell]\!], \hat{\beta}, \widehat{ve}, \hat{t}), \equiv) \rightsquigarrow ((call, \hat{\beta}'', \widehat{ve}', \hat{t}'), \equiv'), \text{ where:}$$

$$\begin{aligned}
\hat{d}_i &= \hat{\mathcal{E}}(e_i, \hat{\beta}, \widehat{ve}) \\
\hat{d}_0 &\ni ([\![(\lambda^{\ell'}\ (v_1 \ldots v_n)\ call)]\!], \hat{\beta}') \\
\hat{t}' &= \widehat{tick}(call, \hat{t}) \\
\hat{b}_i &= \widehat{alloc}(v_i, \hat{t}') \\
\hat{B} &= \left\{\hat{b}_i : \hat{b}_i \in \widehat{Bind}_1\right\} \\
\hat{\beta}'' &= (\hat{g}_{\hat{B}}^{-1}\hat{\beta}')[v_i \mapsto \hat{b}_i] \\
\widehat{ve}' &= (\hat{g}_{\hat{B}}^{-1}\widehat{ve}) \sqcup [\hat{b}_i \mapsto (\hat{g}_{\hat{B}}^{-1}\hat{d}_i)],
\end{aligned}$$

Theorem 3. *Given a compound abstract state $((call, \hat{\beta}, \widehat{ve}, \hat{t}), \equiv)$ and two abstract bindings, $\hat{b}$ and $\hat{b}'$, if $\dot{\alpha}^\eta(call, \beta, ve, t) \sqsubseteq ((call, \hat{\beta}, \widehat{ve}, \hat{t}), \equiv)$ and $\eta(b) = \hat{b}$ and $\eta(b') = \hat{b}'$ and $\hat{b} \equiv \hat{b}'$, then $ve(b) = ve(b')$.*

Theorem 2. *If $\alpha^\eta(\beta_1) = \hat{\beta}_1$ and $\alpha^\eta(\beta_2) = \hat{\beta}_2$, and $\hat{\beta}_1(v) = \hat{\beta}_2(v)$ and $\hat{\beta}_1(v) \in \widehat{Bind}_1$, then $\beta_1(v) = \beta_2(v)$.*

Theorem 1. *If $\alpha^\eta(\varsigma) \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, then there exists a state $\hat{\varsigma}'$ such that $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ and $\alpha^\eta(\varsigma') \sqsubseteq \hat{\varsigma}'$.*

$$\begin{aligned}
alloc &: \mathsf{Var} \times Time \to Bind & \widehat{alloc} &: \mathsf{Var} \times \widehat{Time} \to \widehat{Bind} \\
tick &: \mathsf{Call} \times Time \to Time & \widehat{tick} &: \mathsf{Call} \times \widehat{Time} \to \widehat{Time}
\end{aligned}$$

$$([\![(f\ e_1 \ldots e_n)^\ell]\!], \hat{\beta}, \widehat{ve}, \hat{t}) \rightsquigarrow (call, \hat{\beta}'', \widehat{ve}', \hat{t}'), \text{ where:}$$

$$\begin{aligned}
\hat{d}_i &= \hat{\mathcal{E}}(e_i, \hat{\beta}, \widehat{ve}) \\
\hat{d}_0 &\ni ([\![(\lambda^{\ell'}\ (v_1 \ldots v_n)\ call)]\!], \hat{\beta}') \\
\hat{t}' &= \widehat{tick}(call, \hat{t}) \\
\hat{b}_i &= \widehat{alloc}(v_i, \hat{t}') \\
\hat{B} &= \left\{\hat{b}_i : \hat{b}_i \in Bind_1\right\} \\
\hat{\beta}'' &= (\hat{g}_{\hat{B}}^{-1}\hat{\beta}')[v_i \mapsto \hat{b}_i] \\
\widehat{ve}' &= (\hat{g}_{\hat{B}}^{-1}\widehat{ve}) \sqcup [\hat{b}_i \mapsto (\hat{g}_{\hat{B}}^{-1}\hat{d}_i)],
\end{aligned}$$

$$([\![(f\ e_1 \ldots e_n)^\ell]\!], \beta, ve, t) \Rightarrow (call, \beta'', ve', t'), \text{ where:}$$

$$\begin{aligned}
d_i &= \mathcal{E}(e_i, \beta, ve) \\
d_0 &= ([\![(\lambda^{\ell'}\ (v_1 \ldots v_n)\ call)]\!], \beta') \\
t' &= tick(call, t) \\
b_i &= alloc(v_i, t') \\
B &= \{b_i : b_i \in Bind_1\} \\
\beta'' &= (g_B^{-1}\beta')[v_i \mapsto b_i] \\
ve' &= (g_B^{-1}ve)[b_i \mapsto (g_B^{-1}d_i)],
\end{aligned}$$

Shape analysis of higher-order programs exists.

Shape analysis is useful.

¡Gracias!

matt.might.net

I don’t know.

Yes.

No.

Widening?

Narrowing?