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

- CFA
- Pointer analysis
Bottom-up reason

- CFA
- Pointer analysis
- Shape analysis
Bottom-up reason

- CFA
- Pointer analysis
- ?
- Shape analysis
What is “higher order?”
The essence of higher-order: Lambda calculus.
Syntax

Variables; function abstractions; applications.
Syntax

Variables; function abstractions; applications.

\[ v \; (\lambda \; (v) \; e) \; (e_1 \; e_2) \]
Semantics

Value = Value → 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 \( \lambda \)-calculus mean anything?
Do functions have shape?
What determines the shape of these functions?
Parameters.
\[ f(x) = ax^2 + bx + c \]
\[ f(x) = A \sin(\omega x + \varphi) \]
\[ f(x) = A \sin(\omega x + \varphi) \]
\[ f = \lambda x. A \sin(\omega x + \varphi) \]
Free variables determine function shape.
What determines the value of free variables?
Environments.
Function = Closure = Lambda-term + Environment
\[ \lambda x. A \sin(\omega x + \phi) \]
\((\lambda x. A \sin(\omega x + \varphi), [A=1, \omega=1, \varphi=\pi/2])\)
cos
Environments are linked, mutable, dynamic data structures.
Shape analysis studies linked, mutable, dynamic structures.
Shape analysis of the $\lambda$-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)))

Nearly any CFA will find that at the call site
(h) is the only procedure ever
invoked is a closure over the lambda term
(lambda () x) u The lambda term's
only free variables x are in scope at the invocation site. It feels
safe to inline. Yet, if the compiler replaces the reference to
h with the lambda term
(lambda () x) s the meaning of the program will change from
#f to #t. This happens because the closure that gets invoked was closed over an earlier binding of
x to #f whereas the inlined lambda term closes over the binding of
x currently in scope which is to #t. Programs like this mean that functional compilers must
be conservative when they inline based on information obtained from a CFA. If
the inlined lambda term has a free variables the inlining could be unsafe.

Specific problem To determine the safety of inlining the lambda term
lam at the
call site [(f ...)] s we need to know that for every environment
ρ in which this
call is evaluated that
ρ[[(f)]] = o
lam, ρ and
ρ o vρ = ρ o vρ for each free variable
v in the term
lam.
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 $env_1(x) = env_2(x)$?”
Application: Rematerialization

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 $env_1(z) = env_2(y)$?”
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

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

$\rho \in Env = \text{Var} \rightarrow Clo$

$clo \in Clo = \text{Lam} \times Env$
Small-step state-space

\[ \varsigma \in \text{State} = \text{Call} \times \text{Env} \]
\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Clo} \]
\[ \text{clo} \in \text{Clo} = \text{Lam} \times \text{Env} \]
Split environments (Shivers, 1991)

\[ \rho \in Env = Var \rightarrow Clo \]
Split environments (Shivers, 1991)

\[ \rho \in Env = Var \rightarrow Clo \]

\[ \rho \in Env = Var \rightarrow Clo \]
Split environments
(Shivers, 1991)

\[ \beta \in BEnv = \text{Var} \rightarrow \text{Bind} \]

\[ ve \in VEnv = \text{Bind} \rightarrow \text{Clo} \]
\[ \varsigma \in State = \text{Call} \times \text{BEnv} \times \text{VEnv} \]

\[ \beta \in \text{BEnv} = \text{Var} \rightarrow \text{Bind} \]

\[ ve \in \text{VEnv} = \text{Bind} \rightarrow \text{Clo} \]

\[ clo \in \text{Clo} = \text{Lam} \times \text{BEnv} \]

\[ b \in \text{Bind} \text{ is some infinite set} \]
\( \varsigma \in \text{State} = \text{PC} \times \text{Struct} \times \text{Heap} \)
\( s \in \text{Struct} = \text{Var} \rightarrow \text{Addr} \)
\( h \in \text{Heap} = \text{Addr} \rightarrow \text{Tagged} \)
\( t \in \text{Tagged} = \text{Type} \times \text{Struct} \)
\( a \in \text{Addr} \) is some infinite set
Solving the special problem
Special problem

\( \hat{\beta} \) \( \hat{\beta}' \)
Special problem

\[ \hat{\beta}(v) = \hat{\beta}'(v) \]

\[ \beta(v) = \beta'(v) \]
Special problem

\[ \hat{\beta}(v) = \hat{\beta}'(v) \]

\[ \alpha \]

\[ \beta \]

\[ \beta' \]
Special problem

\[ \hat{\beta}(v) = \hat{\beta}'(v) \]

\[ \beta(v) = \beta'(v) \]

\[ \alpha \]

\[ \alpha \]
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

Original

Anodized
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 \((\lambda \ (v) \text{ call})\), we know \(ve(\beta(x)) = ve(\beta'(v))\).
Solution: Track binding invariants as separate abstraction.
Binding invariants

\[ \Pi \in \text{State} \cong \subseteq \text{Bind} \times \text{Bind} \]
Relational

Mechanical

\[ \Pi \]

\[ s \]

\[ \hat{s} \]
Relational

\[ \Pi \rightarrow \Pi' \]

Mechanical

\[ S \rightarrow \hat{S} \rightarrow \hat{S}' \]
Relational

Mechanical
Relational

Mechanical

$\Pi \rightarrow \Pi'$

$\hat{\Pi} \rightarrow \hat{\Pi}'$

$\Pi'' \rightarrow \Pi'''$

$\hat{\Pi}'' \rightarrow \hat{\Pi}'''$

$\ldots$

$s \rightarrow \hat{s}$

$\ldots$
\[ \hat{\beta}(x) \equiv \hat{\beta}'(y) \]
Related work

- Sagiv, Reps, & Wilhelm, 2002.
- Ball et al., 2001.
- Chase et al., 1990.
- Jagannathan et al., 1998.
More in paper

Specific problem To determine the safety of inlining the lambda term $lam$ at the call site $[(f \ldots)]$, we need to know that for every environment $\rho$ in which this call is evaluated, that $\rho[f] = (lam, \rho')$ 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 \( \lambda n. (\text{ anyway }) \) at the call site \( \{ \times \ldots \} \), we need to know that for every environment \( \rho \) in which this call is evaluated, that \( \rho [x] = (\lambda n. \rho ') \) and \( \rho (v) = \rho (v) \) for each free variable \( v \) in the term \( n \).

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

\[
\hat{\beta}(e_i) \in \widehat{\text{Bind}}_1 \quad \hat{b}_i \in \widehat{\text{Bind}}_1
\]

\[
\hat{\beta}(e_i) \equiv \hat{\beta}(e_i) = \hat{b}_i,
\]

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

\[
\alpha^\eta (\text{call}, \beta, ve, t) = (\alpha^\eta (V), \alpha^\eta (\beta), \alpha^\eta (ve), \eta (t))
\]

\[
\alpha^\eta_{B\text{Env}} (\beta) = \lambda v. \eta (\beta (v))
\]

\[
\alpha^\eta_{\text{Call}} (\beta) = \lambda b. \bigcup_{\eta (b) = b} \alpha^\eta (ve (b))
\]

\[
\alpha^\eta_{\text{Call}} (\lambda b. \eta (b)) = \{ \alpha^\eta (ve (b)) \}
\]

\[
\alpha^\eta_{\text{Call}} (\lambda m. \beta) = \lambda b. \alpha^\eta (\beta (ve (b)))
\]

Theorem 3. Given a compound abstract state \( ((\text{call}, \beta, \hat{v}_e, \hat{t}), \equiv) \) and two abstract bindings, \( \hat{b} \) and \( \hat{b}' \), if \( \alpha^\eta (\text{call}, \beta, ve, t) \subseteq ((\text{call}, \beta, \hat{v}_e, \hat{t}), \equiv) \) and \( \eta (b) = \hat{b} \) and \( \eta (b') = \hat{b}' \) and \( b \equiv b' \), then \( ve (b) = ve (b') \).

\[
\xi \in \hat{\Sigma} = \text{Call} \times \hat{B\text{Env}} \times \hat{\text{Env}} \times \hat{\text{Time}}
\]

\[
\hat{\beta} \in \hat{B\text{Env}} = \text{Var} \times \hat{\text{Bind}}
\]

\[
\hat{ve} \in \hat{\text{Env}} = \text{Bind} \times \hat{\text{D}}
\]

\[
\hat{\text{val}} \in \hat{\text{Val}} = \text{Clo}
\]

\[
\hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{B\text{Env}}
\]

\[
\hat{b} \in \hat{\text{Bind}} \text{ is a finite set of bindings}
\]

\[
\hat{t} \in \hat{\text{Time}} \text{ is a finite set of times}
\]

\[
(((f e_1 \ldots e_n)^\prime, \hat{\beta}, \hat{v}_e, \hat{t}), \equiv) \leadsto ((\text{call}, \beta', \hat{v}_e', \hat{t}'), \equiv'), \text{ where:}
\]

\[
\hat{d}_i = \hat{\beta} (e_i, \hat{v}_e)
\]

\[
\hat{d}_0 \equiv (\hat{\text{Call}} (v_1 \ldots v_n) \text{ call}) \hat{\beta}'
\]

\[
\hat{c} = \hat{\text{tick}} \text{ call}, \hat{t}
\]

\[
\hat{b}_i = \text{alloc} (v_i, \hat{c})
\]

\[
\hat{\beta}' = \{ \hat{b}_i : b_i \in \hat{\text{Bind}}_1 \}
\]

\[
\hat{v}_e' = (\hat{g}_B^{-1} \hat{v}_e) [v_i \mapsto \hat{b}_i]
\]

\[
\hat{ve}' = (\hat{g}_B^{-1} \hat{ve}) \cup \{ b_i \mapsto (\hat{g}_B^{-1} d_i) \},
\]

Theorem 1. If \( \alpha^\eta (\xi) \subseteq \xi \) and \( \xi \Rightarrow \xi' \), then there exists a state \( \xi'' \) such that \( \xi \sim \xi'' \) and \( \alpha^\eta (\xi'') \subseteq \xi'' \).

alloc : Var × Time → Bind

\[
\text{alloc} : \text{Var} \times \hat{\text{Time}} \rightarrow \hat{\text{Bind}}
\]

\[
tick : \text{Call} \times \hat{\text{Time}} \rightarrow \hat{\text{Time}}
\]

\[
tick : \text{Call} \times \hat{\text{Time}} \rightarrow \hat{\text{Time}}
\]
Shape analysis of higher-order programs exists.
Shape analysis is useful.
¡Gracias!

matt.might.net
I don’t know.
Yes.
No.
Widening?
Narrowing?