Shape Analysis in the Absence of Pointers and Structure

Matthew Might

University of Utah, Salt Lake City, Utah, USA
might@cs.utah.edu
http://matt.might.net/

Abstract. Shape analyses (Chase et al. 1990, Sagiv et al. 2002) discover properties of dynamic and/or mutable structures. We ask, "Is there an equivalent to shape analysis for purely functional programs, and if so, what 'shapes' does it discover?" By treating binding environments as dynamically allocated structures, by treating bindings as addresses, and by treating value environments as heaps, we argue that we can analyze the "shape" of higher-order functions. To demonstrate this, we enrich an abstract-interpretive control-flow analysis with principles from shape analysis. In particular, we promote "anodization" as a way to generalize both singleton abstraction and the notion of focusing, and we promote "binding invariants" as the analog of shape predicates. Our analysis enables two optimizations known to be beyond the reach of control-flow analysis (globalization and super-β inlining) and one previously unknown optimization (higher-order rematerialization).

1 Introduction

Control-flow analysis is not enough. In higher-order programs, the three facets of control, environment and data meet and intertwine in a single construct: λ. Deep static analysis of higher-order programs requires that all three facets be co-analyzed with one another. Yet, to date, static analysis of higher-order programs has focused largely on bounding the control facet [1,12,22,26,27,29].[1] Limited excursions have tamed parts of the environment facet [16,18,20,28], and little work even approaches the data facet [17]. These deficits in reasoning leave higher-order languages at a disadvantage with respect to optimization. Our goal in this work is to address these deficits with a holistic approach to the abstract interpretation [5,6] of higher-order programs.

1.1 Limitations of Control-Flow Analysis

To motivate the kind of analysis we need, we will consider specific problems beyond the reach of the control-flow analysis; we will identify the common thread

[1] Control-flow analyses (CFA) answer the higher-order control-flow question: Given a call site $[\![(f\ e_1 \ldots e_n)]\!]$, which procedures may be invoked here? 0CFA, for instance, answers which λ-terms may have closures invoked at the call site.

G. Barthe and M. Hermenegildo (Eds.): VMCAI 2010, LNCS 5944, pp. 263–278, 2010.

as the "generalized environment problem"; and we will then argue that the higher-order analog of shape analysis is what we need to solve it.

CFA Limitation: Super-β inlining. Inlining a function based on flow information is blocked by the lack of environmental precision in control-flow analysis. Shivers termed the inlining of a function based on flow information *super-β* inlining [27], because it is beyond the reach of ordinary β-reduction. Consider:

```
(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)`, the only procedure ever invoked is a closure over the lambda term `(lambda () x)`. The lambda term's only free variable, `x`, is 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)`, 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 variable, the inlining could be unsafe.

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

CFA Limitation: Globalization. Sestoft identified globalization as a second blindspot of control-flow analysis [25]. Globalization is an optimization that converts a procedure parameter into a global variable when it is safe to do so. Though not obvious, globalization can also be cast as a problem of reasoning about environments: if, for every state of execution, all *reachable* environments which contain a variable are equivalent for that variable, then it is safe to turn that variable into a global.

Specific problem. To determine the safety of globalizing the variable v, we need to know that for each reachable state, for any two environments ρ and ρ' reachable inside that state, it must be that $\rho(v) = \rho'(v)$ if $v \in dom(\rho)$ and $v \in dom(\rho')$.

CFA Limitation: Rematerialization. Compilers for imperative languages have found that it can be beneficial to rematerialize (to recompute) a value at its point of use if the values on which it depends are still available. On modern hardware, rematerialization can decrease register pressure and improve cache

[2] The symbol ρ denotes a conventional variable-to-value environment map.

performance. Functional languages currently lack analyses to drive rematerialization. Consider a trivial example:

```
((let ((z y))
  (lambda () z)))
```

At the top-level call site in this program, only a closure over the lambda term `(lambda () z)` is invoked. Yet, we cannot inline the lambda term, changing the program into `((lambda () z))`, because at the very least, the variable `z` isn't even in scope at the call site. We could, however, rematerialize the lambda term `(lambda () y)` instead. Of course, justifying this transformation goes beyond reasoning about the equivalence of environments. What we need is an analysis that can reason about the equivalence of individual *bindings* between environments, *e.g.*, the equality of the binding to the variable `z` within the closure and the binding to the variable `y` at the call site. At the moment, no such analysis exists for higher-order programs.

Specific problem To rematerialize the expression e' in place of expression e, it must be the case that for every environment ρ that evaluates the expression e into a closure (lam, ρ'), that the environment ρ evaluates the expression e' into a closure (lam', ρ'') such that the terms lam and lam' are equal under a substitution $\sigma \subseteq \mathsf{Var} \times \mathsf{Var}$ and for each $(v, v') \in \sigma$, it must be that $\rho'(v) = \rho''(v')$.

1.2 The Generalized Environment Problem

The brief survey of optimizations beyond the reach of higher-order control-flow analysis highlighted the importance of reasoning precisely about environments, and more atomically, about individual bindings. In fact, Shivers's work on k-CFA [27] classified optimizations beyond the reach of CFA as those which must solve "the environment problem."

The term *environment problem* connotes the fact that control-flow analyses excel at reasoning about the λ-term half of closures, but determine little (useful) information about the environment half. Might refined Shivers's definition of the **environment problem** to be determining the equivalence of a pair of environments, for every pair in a given set of environment pairs [16].[3] Equivalence in this case means showing that the environments agree on some specified subset of their domains. This narrow definition is suitable for enabling super-β inlining and globalization, but it is too limited for higher-order rematerialization.

For example, we could not declare the closures $(\llbracket \texttt{(lambda (z) (f z))} \rrbracket, \rho)$ and $(\llbracket \texttt{(lambda (x) (g x))} \rrbracket, \rho')$ to be equivalent unless we knew that $\rho\llbracket \texttt{f} \rrbracket \equiv \rho'\llbracket \texttt{g} \rrbracket$ as well. In this case, the analysis cares about the equality of bindings to two *different* variables in two *different* environments. Thus, the **generalized environment problem** asks whether two *bindings* are equivalent to one another,

[3] The set of pairs comes from concretizing abstract environments, *i.e.*, $\gamma(\hat{\rho}) \times \gamma(\hat{\rho}')$.

where a binding is a variable *plus* the environment in which it was bound, *e.g.*, "Is $[\![\mathtt{x}]\!]$ in environment ρ equivalent to $[\![\mathtt{y}]\!]$ in environment ρ'?"

1.3 Insight: Environments as Data Structures; Bindings as Addresses

Under the hood, environments are *dynamically allocated data structures* that determine the value of a λ-term's free variables, and as a consequence, the meaning of the function represented by a closure. When we adapt and extend the principles of shape analysis (specifically, singleton abstractions [2,4] and shape predicates [23]) to these environments, we can reason about the meaning of and relationships between higher-order functions. As we adapt, we find that, in a higher-order control-flow analysis, bindings are the proper analog of addresses. More importantly, we will be able to solve the aforementioned problems beyond the reach of traditional CFA.

1.4 Contributions

We define the generalized environment problem. We define higher-order rematerialization as a novel client of the generalized environment problem, and we note that super-β inlining and globalization—both known to be beyond the reach CFA—are also clients of the generalized environment problem. We find the philosophical analog of shape analysis for higher-order programs; specifically, we find that we can view binding environments as data structures, bindings as addresses and value environments as heaps. Under this correspondence, we discover *anodization*, a means for achieving both singleton abstraction and focusing; and we discover *binding invariants* as an analog of shape predicates. We use this analysis to solve the generalized environment problem.

2 Platform: Small-Step Semantics, Concrete and Abstract

For our investigation into higher-order shape analysis, our platform is a small-step framework for the multi-argument continuation-passing-style λ-calculus:

$$
\begin{array}{ll}
f, e \in \mathsf{Exp} = \mathsf{Var} + \mathsf{Lam} & v \in \mathsf{Var} ::= id^{\ell} \\
\ell \in \mathsf{Lab} \text{ is a set of labels} & lam \in \mathsf{Lam} ::= (\lambda^{\ell}\ (v_1 \ldots v_n)\ call) \\
& call \in \mathsf{Call} ::= (f\ e_1 \ldots e_n)^{\ell}.
\end{array}
$$

2.1 State-Spaces

The concrete state-space (Σ in Figure 1) for the small-step machine has four components: (1) a call site *call*, (2) a binding environment β to determine the bindings of free variables, (3) a value environment *ve* to determine the value of bindings, and (4) a time-stamp t to encode the current context/history.

The abstract state-space ($\hat{\Sigma}$ in Figure 1) parallels the structure of the concrete state-spaces. For these domains, we assume the natural partial orders; for example, $\widehat{ve} \sqcup \widehat{ve}' = \lambda\hat{b}.\widehat{ve}(\hat{b}) \cup \widehat{ve}'(\hat{b})$.

Binding environments ($BEnv$), as a component of both machine states and closures, are the environments to which the environment problem refers. In our semantics, binding environments map variables to bindings. A binding b is a commemorative token minted for each instance of a variable receiving a value; for example, in k-CFA, a binding is a variable name paired with the time-stamp at which it was bound. The value environment ve tracks the denotable values (D) associated with every binding. A denotable value d is a closure.

In CFAs, bindings—the atomic components of environments—play the role that addresses do in pointer analysis. Our ultimate goal is to infer relationships between the concrete values behind abstract bindings. For example, we want to be able to show that bindings to the variable `v` at some set of times are equal, under the value environment, to the bindings to the variable `x` at some other set of times. (In the pure λ-calculus, the only obvious relationships between bindings are equality and inequality.)

In CFA theory, time-stamps also go by the less-intuitive name of *contours*. Both the concrete and the abstract state-spaces leave the exact structure of time-stamps and bindings undefined. The choices for bindings determine the polyvariance of the analysis. Time-stamps encode the history of execution in some fashion, so that under abstraction, their structure determines the context in context-sensitivity.

The concrete and abstract state-spaces are linked by a parameterized second-order abstraction map, $\alpha^\eta : \Sigma \to \hat{\Sigma}$, where the parameter $\eta : (Addr \to \widehat{Addr}) \cup (Time \to \widehat{Time})$ abstracts both bindings and times:

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

2.2 Transition Rules

With state-spaces defined, we can specify the concrete transition relation for CPS, $(\Rightarrow) \subseteq \Sigma \times \Sigma$; then we can define its corresponding abstraction under the map α^η, $(\rightsquigarrow) \subseteq \hat{\Sigma} \times \hat{\Sigma}$. With the help of an argument-expression evaluator, $\mathcal{E} : \mathsf{Exp} \times BEnv \times VEnv \rightharpoonup D$:

$$\begin{aligned}
\mathcal{E}\ (v, \beta, ve)\ &= ve(\beta(v)) \\
\mathcal{E}\ (lam, \beta, ve)\ &= (lam, \beta),
\end{aligned}$$

$$
\begin{array}{rl}
\varsigma \in \Sigma = & \mathsf{Call} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Time} \\
\beta \in \mathit{BEnv} = & \mathsf{Var} \rightharpoonup \mathit{Bind} \\
\mathit{ve} \in \mathit{VEnv} = & \mathit{Bind} \rightharpoonup D \\
d \in D = & \mathit{Val} \\
\mathit{val} \in \mathit{Val} = & \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} = & \mathsf{Lam} \times \mathit{BEnv} \\
b \in \mathit{Bind} & \text{ is an } \textbf{infinite} \text{ set of bindings} \\
t \in \mathit{Time} & \text{ is an } \textbf{infinite} \text{ set of times}
\end{array}
\qquad
\begin{array}{rl}
\hat{\varsigma} \in \hat{\Sigma} = & \mathsf{Call} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{VEnv}} \times \widehat{\mathit{Time}} \\
\hat{\beta} \in \widehat{\mathit{BEnv}} = & \mathsf{Var} \rightharpoonup \widehat{\mathit{Bind}} \\
\widehat{\mathit{ve}} \in \widehat{\mathit{VEnv}} = & \widehat{\mathit{Bind}} \to \hat{D} \\
\hat{d} \in \hat{D} = & \mathcal{P}(\widehat{\mathit{Val}}) \\
\widehat{\mathit{val}} \in \widehat{\mathit{Val}} = & \widehat{\mathit{Clo}} \\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} = & \mathsf{Lam} \times \widehat{\mathit{BEnv}} \\
\hat{b} \in \widehat{\mathit{Bind}} & \text{ is a } \textbf{finite} \text{ set of bindings} \\
\hat{t} \in \widehat{\mathit{Time}} & \text{ is a } \textbf{finite} \text{ set of times}
\end{array}
$$

Fig. 1. State-space for the lambda calculus: Concrete (left) and abstract (right)

we can define the single concrete transition rule for CPS:

$$
\begin{aligned}
(\llbracket (f\ e_1 \ldots e_n)^{\ell} \rrbracket, \beta, \mathit{ve}, t) \Rightarrow (\mathit{call}, \beta'', \mathit{ve}', t'), \text{ where:} \\
d_i &= \mathcal{E}(e_i, \beta, \mathit{ve}) \\
d_0 &= (\llbracket (\lambda^{\ell'}\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket, \beta') \\
t' &= \mathit{tick}(\mathit{call}, t) \\
b_i &= \mathit{alloc}(v_i, t') \\
\beta'' &= \beta'[v_i \mapsto b_i] \\
\mathit{ve}' &= \mathit{ve}[b_i \mapsto d_i].
\end{aligned}
$$

With the help of an abstract evaluator, $\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{VEnv}} \to \hat{D}$:

$$
\begin{aligned}
\hat{\mathcal{E}}\ (v, \hat{\beta}, \widehat{\mathit{ve}}) &= \widehat{\mathit{ve}}(\hat{\beta}(v)) \\
\hat{\mathcal{E}}\ (\mathit{lam}, \hat{\beta}, \widehat{\mathit{ve}}) &= \left\{ (\mathit{lam}, \hat{\beta}) \right\},
\end{aligned}
$$

we can define an analogous transition rule for the abstract semantics:

$$
\begin{aligned}
(\llbracket (f\ e_1 \ldots e_n)^{\ell} \rrbracket, \hat{\beta}, \widehat{\mathit{ve}}, \hat{t}) \rightsquigarrow (\mathit{call}, \hat{\beta}'', \widehat{\mathit{ve}}', \hat{t}'), \text{ where:} \\
\hat{d}_i &= \hat{\mathcal{E}}(e_i, \hat{\beta}, \widehat{\mathit{ve}}) \\
\hat{d}_0 &\ni (\llbracket (\lambda^{\ell'}\ (v_1 \ldots v_n)\ \mathit{call}) \rrbracket, \hat{\beta}') \\
\hat{t}' &= \widehat{\mathit{tick}}(\mathit{call}, \hat{t}) \\
\hat{b}_i &= \widehat{\mathit{alloc}}(v_i, \hat{t}') \\
\hat{\beta}'' &= \hat{\beta}'[v_i \mapsto \hat{b}_i] \\
\widehat{\mathit{ve}}' &= \widehat{\mathit{ve}} \sqcup [\hat{b}_i \mapsto \hat{d}_i].
\end{aligned}
$$

2.3 Concrete and Abstract Interpretation

To evaluate a program *call* in the concrete semantics, its meaning is the set of states reachable from the initial state $\varsigma_0 = (call, [], [], t_0)$:

$$\{\varsigma : \varsigma_0 \Rightarrow^* \varsigma\}.$$

A naïve abstract interpreter could behave similarly, exploring the states reachable from the initial state $\hat{\varsigma} = (call, [], \bot, \hat{t}_0)$:

$$\{\hat{\varsigma} : \hat{\varsigma}_0 \leadsto^* \hat{\varsigma}\}.$$

In practice, widening on value environments [5] accelerates convergence [16,27].

2.4 Parameters for the Analysis Framework

Time-stamp incrementers and binding allocators serve as parameters:

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

Time-stamps are designed to encode context/history. Thus, the abstract time-stamp incrementer $\widehat{tick}$ and the abstraction map α^η decide how much history to retain in the abstraction. As a result, the function $\widehat{tick}$ determines the context-sensitivity of the analysis. Similarly, the abstract binding allocator chooses how to allocate abstract bindings to variables, and in doing so, it fixes the polyvariance of the analysis. Once the parameters are fixed, the semantics must obey a straightforward soundness theorem:

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

3 Analogy: Singleton Abstraction to Binding Anodization

Focusing on our goal of solving the generalized environment problem—reasoning about the equality of individual bindings—we turn to singleton abstraction [4]. Singleton abstraction has been used in pointer and shape analyses to drive must-alias analysis; we extend singleton abstraction, and the framework of anodization, to determine the equivalence of bindings to the *same* variable. That is, we will be able to solve the environment problem with our singleton abstraction, but not the *generalized* environment problem. In Section 4, we will solve the generalized problem by bootstrapping binding invariants on top of anodization.

A Galois connection [6] $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$ has a singleton abstraction iff there exists a subset $\hat{X}_1 \subseteq \hat{X}$ such that for all $\hat{x} \in \hat{X}_1$, $size(\gamma(\hat{x})) = 1$. The critical property of singleton abstractions is that equality of abstract representatives implies equality of their concrete constituents. Hence, when the set X contains addresses, singleton abstractions enable must-alias analysis. Analogously, when the set X contains bindings, singleton abstraction enables binding-equality testing.

Example 1. Suppose we have a concrete machine with three memory addresses: `0x01`, `0x02` and `0x03`. Suppose the addresses abstract so that $\alpha(\texttt{0x01}) = \hat{a}_1$ and $\alpha(\texttt{0x02}) = \alpha(\texttt{0x03}) = \hat{a}_*$. The address $\hat{a}_1$ is a singleton abstraction, because it has only one concrete constituent—`0x01`. After a pointer analysis, if some pointer variable `p1` points only to address $\hat{a}'$ and another pointer variable `p2` points only to address $\hat{a}''$ and $\hat{a}' = \hat{a}_1 = \hat{a}''$ then `p1` must alias `p2`.

In order to solve the super-β inlining problem, Shivers informally proposed a singleton abstraction for k-CFA which he termed "re-flow analysis" [27]. In re-flow analysis, the CFA is re-run, but with a "golden" contour inserted at a point of interest. The golden contour—allocated only once—is a singleton abstraction by definition. While sound in theory, re-flow analysis does not work in practice: the golden contour flows everywhere the non-golden contours flow, and inevitably, golden and non-golden contours are compared for equality. Nevertheless, we can salvage the spirit of Shivers's golden contours through *anodization*. Under anodization, bindings are not golden, but may be temporarily gold-plated.

In anodization, the concrete and abstract bindings are split into two halves:

$$Bind = Bind_\infty + Bind_1 \qquad \widehat{Bind} = \widehat{Bind}_\infty + \widehat{Bind}_1,$$

and we assert "anodizing" bijections between these halves:

$$g : Bind_\infty \to Bind_1 \qquad \hat{g} : \widehat{Bind}_\infty \to \widehat{Bind}_1,$$

such that:

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

Every abstract binding has two variants, a summary variant, $\hat{b}$, and an anodized variant, $\hat{g}(\hat{b})$. We will craft the concrete and abstract semantics so that the anodized variant will be a singleton abstraction. We must anodize concrete bindings as well because the concrete semantics have to employ the same anodization strategy as the abstract semantics in order to prove soundness.

The concrete semantics must also obey an abstraction-uniqueness constraint over anodized bindings, so that for any reachable state $(call, \beta, ve, t)$:

$$\text{If } g(b) \in dom(ve) \text{ and } g(b') \in dom(ve) \text{ and } \eta(b) = \eta(b') \text{ then } b = b'. \quad (1)$$

In other words, once the concrete semantics decides to allocate an anodized binding, it must de-anodize existing concrete bindings which abstract to the same abstract binding. Anodization by itself does not dictate *when* a concrete semantics should allocate an anodized binding; this is a *policy* decision; anodization is a *mechanism*. For simple policies, the parameters *alloc* and $\widehat{alloc}$, by selecting anodized or summary bindings, jointly encode the policy.

As an example of the simplest anodization policy, we describe the higher-order analog of Balakrishnan and Reps's recency abstraction in Section 3.3. An example of a more complicated policy is closure-focusing (Section 3.4).

Formally, the concrete transition rule must rebuild the value environment with every transition:

$$\begin{aligned}(\llbracket(f\ e_1 \ldots e_n)^{\ell}\rrbracket, \beta, ve, t) \Rightarrow (call, \beta'', ve', t'), \text{ where:}\\ d_i &= \mathcal{E}(e_i, \beta, ve)\\ d_0 &= (\llbracket(\lambda^{\ell'}\ (v_1 \ldots v_n)\ call)\rrbracket, \beta')\\ t' &= tick(call, t)\\ b_i &= alloc(v_i, t')\\ B &= \{b_i : b_i \in Bind_1\}\\ \beta'' &= (g_B^{-1}\beta')[v_i \mapsto b_i]\\ ve' &= (g_B^{-1}ve)[b_i \mapsto (g_B^{-1}d_i)],\end{aligned}$$

where the de-anodization function $g_B^{-1} : (BEnv \to BEnv) \cup (VEnv \to VEnv) \cup (D \to D) \cup (Bind \to Bind)$ strips the anodization off bindings that abstract to any binding in the set B:

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

The corresponding abstract transition rule must also rebuild the value environment with every transition:

$$\begin{aligned}(\llbracket(f\ e_1 \ldots e_n)^{\ell}\rrbracket, \hat{\beta}, \widehat{ve}, \hat{t}) \rightsquigarrow (call, \hat{\beta}'', \widehat{ve}', \hat{t}'), \text{ where:}\\ \hat{d}_i &= \hat{\mathcal{E}}(e_i, \hat{\beta}, \widehat{ve})\\ \hat{d}_0 &\ni (\llbracket(\lambda^{\ell'}\ (v_1 \ldots v_n)\ call)\rrbracket, \hat{\beta}')\\ \hat{t}' &= \widehat{tick}(call, \hat{t})\\ \hat{b}_i &= \widehat{alloc}(v_i, \hat{t}')\\ \hat{B} &= \left\{\hat{b}_i : \hat{b}_i \in Bind_1\right\}\\ \hat{\beta}'' &= (\hat{g}_{\hat{B}}^{-1}\hat{\beta}')[v_i \mapsto \hat{b}_i]\\ \widehat{ve}' &= (\hat{g}_{\hat{B}}^{-1}\widehat{ve}) \sqcup [\hat{b}_i \mapsto (\hat{g}_{\hat{B}}^{-1}\hat{d}_i)],\end{aligned}$$

where the de-anodization function $\hat{g}_{\hat{B}}^{-1} : (\widehat{BEnv} \to \widehat{BEnv}) \cup (\widehat{VEnv} \to \widehat{VEnv}) \cup (\widehat{D} \to \widehat{D}) \cup (\widehat{Val} \to \widehat{Val}) \cup (\widehat{Bind} \to \widehat{Bind})$ strips the anodization off abstract bindings in the set $\hat{B}$:

$$\hat{g}^{-1}_{\hat{B}}(\hat{b}) = \begin{cases} \hat{b}' & \hat{b} \in \hat{B} \text{ and } \hat{b} = \hat{g}(\hat{b}') \\ \hat{b} & \text{otherwise} \end{cases}$$
$$\hat{g}^{-1}_{\hat{B}}\left\{\hat{d}_1, \ldots, \hat{d}_n\right\} = \left\{\hat{g}^{-1}_{\hat{B}}(\hat{d}_1), \ldots, \hat{g}^{-1}_{\hat{B}}(\hat{d}_n)\right\}$$
$$\hat{g}^{-1}_{\hat{B}}(lam, \hat{\beta}) = (lam, \hat{g}^{-1}_{\hat{B}}(\hat{\beta}))$$
$$\hat{g}^{-1}_{\hat{B}}(\hat{\beta}) = \lambda v.\hat{g}^{-1}_{\hat{B}}(\hat{\beta}(v))$$
$$\hat{g}^{-1}_{\hat{B}}(\widehat{ve}) = \lambda \hat{b}.\hat{g}^{-1}_{\hat{B}}(\widehat{ve}(\hat{b})).$$

Because the concrete semantics obey the uniqueness constraint (Equation 1), the abstract interpretation may treat the set $\widehat{Bind}_1$ as a set of singleton abstractions for the purpose of testing binding equality.

3.1 Solving the Environment Problem with Anodization

Given two abstract environments $\hat{\beta}_1$ and $\hat{\beta}_2$, it is easy to determine whether the concrete constituents of these environments agree on the value of some subset of their domains, $\{v_1, \ldots, v_n\}$:

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* $\beta_1(v) = \beta_2(v)$.

Proof. By the abstraction-uniqueness constraint.

3.2 Implementing Anodization Efficiently

The naïve implementation of the abstract transition rule is inefficient: the de-anodizing function $\hat{g}^{-1}_{\hat{B}}$ must walk the abstract value environment with *every* transition. Even in 0CFA, this walk adds a quadratic penalty to every transition. To avoid this walk, the analysis should use serial numbers on bindings "under the hood," so that:

$$\widehat{Bind} \approx \widehat{Bind}_\infty \times \mathbb{N}.$$

That is, the value environment should be implemented as two maps:

$$\widehat{VEnv} \approx (\widehat{Bind}_\infty \to \mathbb{N} \to \hat{D}) \times (\widehat{Bind}_\infty \to \mathbb{N}).$$

Given a split value environment $\widehat{ve} = (\hat{f}, \hat{h})$, a binding $(\hat{b}, n)$ is anodized only if $n = \hat{h}(\hat{b})$, and it is not anodized if $n < \hat{h}(\hat{b})$. Thus, when the allocator chooses to anodize a binding, it does need to walk the value environment with the function $\hat{g}^{-1}_{\hat{B}}$ to strip away existing anodization; it merely needs to increment the serial number associated with that binding.

3.3 Instantiating Anodization: Recency Abstraction

In recency abstraction [2], the most-recently allocated abstract variant of a resource is tracked distinctly from previously allocated variants. Anodization makes it straightforward to model recency in a higher-order setting. In a language with mutation, recency abstraction solves the initialization problem, whereby addresses are allocated with a default value, but then set to another shortly thereafter. Recency abstraction prevents the default value from appearing as a possibility for every address, which is directly useful in eliminating null-pointer checks. In a higher-order setting, recency permits precise computation of binding equivalence for variables that are bound in non-recursive and tail-recursive procedures or that die before the recursive call.

3.4 Instantiating Anodization: Closure-Focusing

Anodization enables another shape-analytic technique known as focusing [15,23]. In focusing, a specific, previously-allocated variant is split into the singleton variant under focus—and all other variants. In a higher-order language, there is a natural opportunity to focus on all of the bindings of a closure when it is created. Focusing provides a way to solve the environment problem for closures which capture variables which have been re-bound since closure-creation.

4 Analogy: Binding Invariants as Shape Predicates

Anodization can solve the environment problem, but it cannot solve the generalized environment problem, where we need to be able to reason about the equality of bindings to *different* variables in *different* environments. To solve this problem, we cast shape predicates as *binding invariants*. A binding invariant is an equivalence relation over abstract bindings, and it can be considered as a separate, relational abstraction of program state, $\alpha^{\eta}_{\equiv} : \Sigma \to \hat{\Sigma}_{\equiv}$, where:

$$\hat{\Sigma}_{\equiv} = \mathcal{P}\left(\widehat{Bind} \times \widehat{Bind}\right),$$

such that:

$$\alpha^{\eta}_{\equiv}(call, \beta, ve, t) = \left\{(\hat{b}, \hat{b}') : ve(b) = ve(b') \text{ if } \eta(b) = \hat{b} \text{ and } \eta(b') = \hat{b}'\right\}.$$

In contrast with earlier work, binding-invariant abstraction is a relational abstract domain over *abstract bindings* rather than *program variables* [7,8].

Informally, if $(\hat{b}, \hat{b}') \in \alpha^{\eta}_{\equiv}(\varsigma)$, it means that all of the concrete constituents of the bindings $\hat{b}$ and $\hat{b}'$ agree in value. To create the analysis, we can formulate a new abstraction as the direct product of the abstractions α^{η} and $\alpha^{\eta}_{\equiv}$:

$$\dot{\alpha^{\eta}} : \Sigma \to \hat{\Sigma} \times \hat{\Sigma}_{\equiv}$$
$$\dot{\alpha^{\eta}}(\varsigma) = (\alpha^{\eta}(\varsigma), \alpha^{\eta}_{\equiv}(\varsigma)).$$

The constraints of a straightforward soundness theorem (Theorem 1) lead to an abstract transition relation over this space:

$$\begin{aligned}((\llbracket(f\ e_1 \ldots e_n)^{\ell}\rrbracket, \hat{\beta}, \widehat{ve}, \hat{t}), \equiv) \rightsquigarrow ((call, \hat{\beta}'', \widehat{ve}', \hat{t}'), \equiv'), \text{ where:}\\ \hat{d}_i &= \hat{\mathcal{E}}(e_i, \hat{\beta}, \widehat{ve})\\ \hat{d}_0 &\ni (\llbracket(\lambda^{\ell'}\ (v_1 \ldots v_n)\ call)\rrbracket, \hat{\beta}')\\ \hat{t}' &= \widehat{tick}(call, \hat{t})\\ \hat{b}_i &= \widehat{alloc}(v_i, \hat{t}')\\ \hat{B} &= \left\{\hat{b}_i : \hat{b}_i \in \widehat{Bind}_1\right\}\\ \hat{\beta}'' &= (\hat{g}_{\hat{B}}^{-1}\hat{\beta}')[v_i \mapsto \hat{b}_i]\\ \widehat{ve}' &= (\hat{g}_{\hat{B}}^{-1}\widehat{ve}) \sqcup [\hat{b}_i \mapsto (\hat{g}_{\hat{B}}^{-1}\hat{d}_i)],\end{aligned}$$

and singleton bindings are reflexively equivalent:

$$\frac{\hat{b} \in \widehat{Bind}_1}{\hat{b} \equiv' \hat{b},}$$

and bindings between singletons are trivially equivalent:

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

and untouched bindings retain their equivalence:

$$\frac{\hat{b} \equiv \hat{b}' \qquad \hat{b} \notin \hat{B} \qquad \hat{b}' \notin \hat{B}}{\hat{b} \equiv' \hat{b}',}$$

and bindings re-bound to themselves also retain their equivalence:

$$\frac{\hat{\beta}(e_i) \equiv \hat{b}_i}{\hat{\beta}(e_i) \equiv' \hat{b}_i.}$$

4.1 Solving the Generalized Environment Problem

Under the direct product abstraction, the generalized environment theorem, which rules on the equality of individual bindings, follows naturally:

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* $ve(b) = ve(b')$*.*

Proof. By the structure of the direct product abstraction $\dot{\alpha^{\eta}}$.

5 Application: Higher-Order Rematerialization

Now that we have a generalized environment analysis, we can precisely state the condition under which higher-order rematerialization is safe. Might's work on the correctness of super-β inlining formally defined *safe* to mean that the transformed program and the untransformed program maintain a bisimulation in their concrete executions [16].

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 $(v', v) \in \sigma$, $\hat{\beta}'(v') \equiv \hat{\beta}(v)$.*

Proof. The proof of bisimulation has a structure identical to that of the proof correctness for super-β inlining in [16].

6 Related Work

Clearly, this work draws on the Cousots' abstract interpretation [5,6]. Binding invariants succeed the Cousots' work as a relational abstraction of higher-order programs [7,8], with the distinction that binding invariants range over abstract bindings instead of formal parameters. Binding invariants were also inspired by Gulwani *et al.*'s quantified abstract domains [9]; there is an implicit universal quantification ranging over concrete constituents in the definition of the abstraction map $\alpha^{\eta}_{\equiv}$. This work also falls within and retains the advantages of Schmidt's small-step abstract interpretive framework [24]. As a generalization of control-flow analysis, the platform of Section 2 is a small-step reformulation of Shivers's denotational CFA [27], which itself was a extension of Jones's original CFA [13]. Like the Nielsons' unifying work on CFA [22], this work is an implicit argument in favor of the inherent flexibility of abstract interpretation for the static analysis of higher-order programs. In contrast with constraint-based, type-based and model-checking CFAs, small-step abstract interpretive CFAs are easy to extend via direct products and parameterization.

From shape analysis, anodized bindings draw on singleton abstraction while binding invariants are inspired by both predicate-based abstractions [3] and three-valued logic analysis [23]. Chase *et. al* had early work on counting-based singleton abstractions [4], while Hudak's work on analysis of first-order functional programs employed a precursor to counting-based singleton abstraction [10]. Anodization, using factored sets of singleton and non-singleton bindings, is most closely related to the Balakrishnan and Reps's recency abstraction [2], except that anodization works on bindings instead of addresses, and anodization is not restricted to a most-recent allocation policy. Superficially, one might also term Jones and Bohr's work on termination analysis of the untyped λ-calculus via size-change as another kind of shape analysis for higher-order programs [14].

Given the importance of inlining and globalization, the functional community has responded with *ad hoc* extensions to control-flow analyses to support these optimizations. Shivers's re-flow analysis developed the concept of singleton abstraction independently to determine equivalence over environments [27]. Wand and Steckler approached the environment problem by layering a constraint-based environment-equivalence analysis on top of 0CFA [28]. Jagannathan *et al.* developed a counting-based constraint analysis to drive lightweight closure conversion [11]. More recently, Might and Shivers attacked the problem with stack-driven environment-analysis (ΔCFA), but this analysis also proved too brittle for many programs [18]. Might and Shivers' reachability- and counting-driven environment analysis (ΓCFA) provides a scalable analysis which can reason about environment equivalence [19,21]. All of these extensions are capable of solving the environment problem in limited cases; none of them can solve the generalized environment problem, and none take the principled, flexible approach provided by anodization and binding invariants.

7 Conclusion

We motivated the need to reason about the equivalence of environments in higher-order programs by finding optimizations beyond the reach of ordinary control-flow analysis: super-β inlining, globalization and higher-order rematerialization. We distilled the core problem which must be solved in order to enable these optimizations—the generalized environment problem. The generalized environment problem asks whether two variables bound in different environments are equivalent, *e.g.*, "Is $[\![\mathtt{x}]\!]$ in bound in ρ equivalent to $[\![\mathtt{y}]\!]$ bound in ρ'?" We then created an analysis framework for solving the generalized environment problem by considering the analog of shape analysis in terms of control-flow analysis. We rendered the principle of singleton abstraction as anodization, and we rendered the principle of shape predicates as binding invariants. By composing anodization and binding invariants, we arrived at an extended higher-order flow-analysis framework that can solve the generalized environment problem.

8 Future Work

Next steps for this work include folding more language features into the framework, considering the impact of these features on both anodization and binding invariants and integrating Gulwani's techniques for bounding of numeric variables [9]. For instance, once numbers are introduced, we could enrich binding invariants to reason about both equality and inequality among the concrete constituents of abstract bindings. We also expect that when we introduce dynamic allocation, that anodization and binding invariants will naturally morph back into the must-alias analysis and shape predicates from whence they came. This technology is also being introduced into the U Combinator higher-order flow analysis toolkit; the latest beta version of this toolkit is always available from `http://www.ucombinator.org/`.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI 2001: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pp. 203–213. ACM Press, New York (2001)
4. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI 1990: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, pp. 296–310. ACM Press, New York (1990)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 269–282. ACM Press, New York (1979)
7. Cousot, P., Cousot, R.: Relational abstract interpretation of higher-order functional programs. In: JTASPEFL 1991, Bordeaux. BIGRE 74, pp. 33–36 (1991)
8. Cousot, P., Cousot, R.: Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages). In: Proceedings of the 1994 International Conference on Computer Languages, pp. 95–112. IEEE Computer Society Press, Los Alamitos (1994)
9. Gulwani, S., Mccloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL 2008: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 235–246. ACM, New York (2008)
10. Hudak, P.: A semantic model of reference counting and its abstraction. In: LFP 1986: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 351–363. ACM, New York (1986)
11. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: POPL 1998: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 329–341. ACM, New York (1998)
12. Jagannathan, S., Weeks, S.: A unified treatment of flow analysis in higher-order languages. In: POPL 1995: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 393–407. ACM, New York (1995)
13. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Proceedings of the 8th Colloquium on Automata, Languages and Programming, London, UK, pp. 114–128. Springer, Heidelberg (1981)
14. Jones, N.D., Bohr, N.: Call-by-value termination in the untyped lambda-calculus. Logical Methods in Computer Science 4(1), 1–39 (2008)
15. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Finding concurrency-related bugs using random isolation. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 198–213. Springer, Heidelberg (2009)

16. Might, M.: Environment Analysis of Higher-Order Languages. PhD thesis, Georgia Institute of Technology (June 2007)
17. Might, M.: Logic-flow analysis of higher-order programs. In: POPL 2007: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 185–198. ACM Press, New York (2007)
18. Might, M., Shivers, O.: Environment analysis via delta-cfa. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 127–140. ACM, New York (2006)
19. Might, M., Shivers, O.: Improving flow analyses via gamma-cfa: Abstract garbage collection and counting. In: ICFP 2006: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, pp. 13–25. ACM, New York (2006)
20. Might, M., Shivers, O.: Analyzing the environment structure of higher-order languages using frame strings. Theoretical Computer Science 375(1-3), 137–168 (2007)
21. Might, M., Shivers, O.: Exploiting reachability and cardinality in higher-order flow analysis. Journal of Functional Programming 18(special double issue 5-6), 821–864 (2008)
22. Nielson, F., Nielson, H.R.: Infinitary control flow analysis: a collecting semantics for closure analysis. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 332–345. ACM, New York (1997)
23. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
24. Schmidt, D.A.: Abstract interpretation of small-step semantics. In: Dam, M. (ed.) LOMAPS-WS 1996. LNCS, vol. 1192, pp. 76–99. Springer, Heidelberg (1997)
25. Sestoft, P.: Analysis and efficient implementation of functional programs. PhD thesis, University of Copenhagen, Denmark (October 1991)
26. Shivers, O.: Control flow analysis in Scheme. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, vol. 23, pp. 164–174. ACM, New York (1988)
27. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University (1991)
28. Wand, M., Steckler, P.: Selective and lightweight closure conversion. In: POPL 1994: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 435–445. ACM, New York (1994)
29. Wright, A.K., Jagannathan, S.: Polymorphic splitting: An effective polyvariant flow analysis. ACM Transactions on Programming Languages and Systems 20(1), 166–207 (1998)