Visualizing Abstract Abstract Machines

KYLE HEADLEY, University of Alabama, Birmingham
CLARK REN, University of Alabama, Birmingham
KRISTOPHER MICINSKI, Syracuse University
THOMAS GILRAY, University of Alabama, Birmingham

We present an approach for interactively visualizing static analyses built using the abstracting abstract machines (AAM) methodology—a process that yields a static program analysis by *abstract interpretation* of an *abstract machine*. The resulting analysis is a state graph of all *possible* machine states—with paths through this graph encoding possible executions of the program—combined with a model of the heap. To understand or audit the results of such an analysis (e.g., for debugging or improving the analysis) can become a laborious process of stepping from state to state, building an intuition for each, while considering valid executions that are missing and spurious executions that are included. Finding states relevant to some program property, on its own, can involve writing a custom predicate to match such states at the REPL.

In this paper, we explore an approach to concisely visualizing AAM-based analyses of Scheme programs by decomposing the analysis into its functional components and displaying nested graphs for inter- and intra-procedural control flow. We allow interactive visualization in that the user can focus on specific functions or lines of code to discover if they're reachable, in what contexts, atop what stacks, and with what values bound to variables in scope, in terms of states in an abstract abstract machine.

Additional Key Words and Phrases: program analysis, visualization, program understanding, abstract interpretation

1 INTRODUCTION

Abstract interpretation (AI) is a powerful technique for reasoning about the behavior of programs [Cousot and Cousot 1977a]. For example, AI can verify the absence of null pointer errors, be used to inline polymorphic function calls, eliminate runtime array-bounds checks, and more. In this paper, we focus on understanding the results of abstract interpretation, applied to abstract machines—a common approach to programming language semantics (we present background on this approach in Section 2).

While static analysis tools have varied and important applications, they are often seen as complex and hard-to-understand. In particular, when the analysis produces an unexpected result, it can be exceptionally challenging to understand what led to that result. Our goal is to develop an effective visualization that allows an analysis user or designer to systematically understand an analysis result. We foresee several potential users of such a visualization including analysis developers, students seeking to understand abstract interpretation, and static analysis researchers attempting to understand the effects of various precision-enhancing techniques on the final analysis results.

Our visualization builds upon the *abstracting abstract machines* (AAM) program analysis methodology [Might 2010; Van Horn and Might 2010]. In AAM, the result of the abstract interpretation is a control-flow graph (representing explored program behavior) and a model of the heap. This leads to a straightfoward visualization via the control-flow graph produced by the analysis. However, as we observe in Section 3, this is not necessarily an ideal visualization. Instead, we develop an analysis presenting the user with two views: one for intraprocedural

Authors' addresses: Kyle Headley, University of Alabama, Birmingham, kheadley@uab.edu; Clark Ren, University of Alabama, Birmingham, cren@uab.edu; Kristopher Micinski, Syracuse University, kkmicins@syr.edu; Thomas Gilray, University of Alabama, Birmingham, gilray@uab.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

behavior and one for interprocedural behavior. This allows the user to focus on the execution of an individual function while also seeing a summary of the whole-program, interprocedural behavior in a side-by-side view.

We have implemented a web-based prototype visualization for Shivers' k-CFA [Shivers 1988] using our technique. Our implementation analyzes a subset of Scheme and allows for easy tuning of analysis sensitivity. Our system presents a visualization to the user and allows them to step through the analysis results, inspecting both the control-flow and values of variables, alongside the program's source. We see this as a promising step that we hope can lead to improvements in the way users and designers interact with the results of static analysis.

Contributions. In this paper we make the following contributions.

- We present an AAM-style analysis for a Scheme IR that is flexible enough to reach a variety of different analysis goals by adjusting a small number of parameters.
- We describe an algorithm for segmenting AAM-style analyses so that a visualization may focus on individual functions and their interactions.
- We implement our approach as a web application (https://analysisviz.gilray.net) to visualize and explore analyses of Scheme programs.

The rest of the paper is organized as follows: we develop an abstract abstract machine (AAM) for Scheme in section 2. We start with a concrete semantics of a small intermediate language λ_{scm} defined as operational rules for advancing a machine state in section 2.1. This is used to develop an abstract semantics in section 2.2, and provided with features for efficiency and tunability in section 2.3. Section 3 presents the trouble with exploring the results of an AAM analysis naïvely, motivating our work on an AAM visualizer. We walk through a demo in section 3.1, discuss potential pitfalls of visualizations in section 3.2, and introduce our strategy for dealing with these issues in section 3.3. We cover our implementation in section 4, which includes a novel *segmentation* algorithm (presented in section 4.2) to assemble the different views used in our visualizer.

2 ABSTRACT ABSTRACT MACHINES

The theory of **abstract interpretation** (AI) [Cousot and Cousot 1976, 1977a,b, 1992] gives us a toolset for modeling the behavior of a program (explicating and proving its properties) by over- or under-approximating the set of possible execution traces for the program. This is accomplished by modifying a **concrete semantics** for the target programming language to produce an **abstract semantics** that faithfully models the original, maintaining some essential accuracy while trading away precision in return for computability and an upper-bound on analysis complexity. With *sufficient* precision, computable static analyses constructed in this way are able to verify important safety properties, empower compiler optimizations, and assist in auditing a program's information flows and behavior.

Abstracting abstract machines (AAM) [Might 2010; Van Horn and Might 2010] is a general methodology for applying abstract interpretation to abstract-machine-based semantics. **Abstract machines** are a type of a structural operational semantics [Hennessy 1990; Plotkin 1981; Winskel 1993] relating a succession of machine states, each encoded as a tuple of machine components such as a control expression or program counter, a store/heap, and a call stack. Abstract machines give semantics engineers fine-grained control to represent all important aspects of program interpretation (such as control-flow, mutation, first-class control, exceptions, first-class functions, etc) using arbitrary, nested mathematical entities such as tuples, sets, and maps, and to describe precisely how the machine in one state advances to its succeeding state (potentially stepping from state to state forever, or until terminating properly, or getting into a *stuck* error state that cannot be advanced).

The AAM methodology allows a high degree of control over how program states are represented and makes it easy to instrument with additional information as necessary. In a functional setting, for example, it is natural to construct a **control-flow analysis** (CFA) [Might 2007; Shivers 1991] that models the possible control-flow paths a program may follow in any concrete evaluation. CFA requires **closure analysis**—that is, tracking which

$$
\begin{aligned}
e \in \textbf{Exp} &::= (\texttt{let}\ ([x_0\ e_0][x_1\ e_1]\ldots)\ e_b) \\
&\mid (\texttt{e}_0\ \texttt{e}_1\ldots) \\
&\mid \texttt{ae} \\
\texttt{ae} \in \textbf{AExp} &::= x \mid \texttt{lam} \\
\texttt{lam} \in \textbf{Lam} &::= (\lambda\ (x\ y\ldots)\ e) \\
x, y \in \textbf{Var} &\text{ is a set of identifiers}
\end{aligned}
$$

Fig. 1. Our target language λ_{scm}

$$
\begin{aligned}
\mathcal{A}(x, \rho, \sigma) &\triangleq \sigma(\rho(x)) \\
\mathcal{A}((\lambda\ (x_0\ldots)\ e), \rho, \sigma) &\triangleq \langle \textbf{clo}\ (\lambda\ (x_0\ldots)\ e), \rho \rangle
\end{aligned}
$$

Fig. 2. Concrete atomic-expression evaluation

$$
\varsigma \in \Sigma \triangleq \begin{cases} \text{eval: } \textbf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \\ \text{apply: } \mathit{Clo}^* \times \mathit{Store} \times \mathit{Kont} \end{cases}
$$

$$
\begin{aligned}
\rho \in \mathit{Env} &\triangleq \textbf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &\triangleq \mathit{Addr} \rightharpoonup \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} &\triangleq \textbf{Lam} \times \mathit{Env} \\
\kappa \in \mathit{Kont} &\triangleq \mathit{Frame}^* \\
\psi \in \mathit{Frame} &\triangleq \mathit{Clo}^* \times \mathit{Exp}^* \times \mathit{Env} \\
a \in \mathit{Addr} &\text{ is an infinite set}
\end{aligned}
$$

Fig. 3. CESK machine domains

closures can flow to which program variables or heap addresses—in order to bound the possible the callees invoked at any particular call site.

In this section, we review the principles of AAM, including how to tune precision and complexity, how to perform standard structural simplifications, and how to plug-in different approximations for program values. We develop an abstract machine (concrete semantics) and abstract abstract machine (abstract semantics) for a Scheme intermediate representation that we will visualize in following sections.

2.1 Abstract Machines (CESK)

In this section we define a concrete semantics for the language λ_{scm}, shown in figure 1. We demonstrate our general approach to analysis using this core intermediate language, deriving an abstract semantics from its concrete semantics. Figure 3 defines a CESK machine whose states are generally comprised of four components: a control expression, a binding environment, a value store, and a continuation (model of the stack). More specifically, machine states are one of two kinds: an **eval** state or an **apply** state. **eval** states have expression, environment, store, and continuation components. **apply** states have a list of values (a function being applied followed by its arguments) and a store and continuation.

We denote domains of unbounded sequences using a star operator; for example, a continuation (κ) is defined as a sequence of frames, (ψ). Each of these frames consists of a sequence of values (closures), a sequence of expressions to be evaluated, and the environment to evaluate them under. Closures (*clo*) are the only type of value in λ_{scm}, represented as a lambda term and the environment it was closed under. Environments (ρ) are each a mapping from variables to addresses, which the Store (σ) maps to values. We separate these domains in preparation for abstraction, where having a distinguished set of addresses to finitize plays an important part. Here, the the set of addresses (*Addr*) is some infinite set (such as $\mathbb{N}$).

Figure 2 shows the atomic-expression evaluator we use as a helper function in the following semantics. It either looks up a variable in the environment, and its address in the store, or it packages up the lambda expression and environment into a closure.

Figure 4 shows five small-step rules defining evaluation. The **[Let]** rule handles let forms as equivalent to an immediate application of a lambda ($\lambda\ (x\ y\ldots)\ e_b$). The rule moves to evaluate the first right-hand-side (RHS), e_0, while pushing a new frame onto the current continuation (stack). Each frame contains three components: a list

$$
\begin{array}{rll}
\langle \textbf{eval}\ (\texttt{let}\ ([x\ e_0][y\ e_1]\ldots)\ e_b), \rho, \sigma, \kappa\rangle \to_\Sigma \langle \textbf{eval}\ e_0, \rho, \sigma, \langle \textbf{frame}\ (\langle \textbf{clo}\ (\lambda\ (x\ y\ldots)\ e_b), \rho\rangle), (e_1\ldots), \rho\rangle :: \kappa\rangle & & \textbf{[Let]}\\
\langle \textbf{eval}\ (e_0\ e_1\ldots), \rho, \sigma, \kappa\rangle \to_\Sigma \langle \textbf{eval}\ e_0, \rho, \sigma, \langle \textbf{frame}\ (), (e_1\ldots), \rho\rangle :: \kappa\rangle & & \textbf{[App]}\\
\langle \textbf{eval}\ ae, \rho, \sigma, \langle \textbf{frame}\ (v_0\ldots), (), \rho_\kappa\rangle :: \kappa\rangle \to_\Sigma \langle \textbf{apply}\ (v_0\ldots\ v_n), \sigma, \kappa\rangle, \text{ where} & & \textbf{[EvalApply]}\\
v_n = \mathcal{A}(ae, \rho, \sigma) & & \\
\langle \textbf{apply}\ (\langle \textbf{clo}\ (\lambda\ (x_0\ldots x_n)\ e), \rho\rangle\ v_0\ldots v_n), \sigma, \kappa\rangle \to_\Sigma \langle \textbf{eval}\ e, \rho', \sigma', \kappa\rangle, \text{ where} & & \textbf{[ApplyEval]}\\
\rho' = \rho[x_i \mapsto a_i] & & \\
\sigma' = \sigma[a_i \mapsto v_i] & & \\
a_i \notin \text{dom}(\sigma) & & \\
\langle \textbf{eval}\ ae, \rho, \sigma, \langle \textbf{frame}\ (v_0\ldots), (e_0\ e_1\ldots), \rho_\kappa\rangle :: \kappa\rangle \to_\Sigma \langle \textbf{eval}\ e_0, \rho_\kappa, \sigma, \langle \textbf{frame}\ (v_0\ldots\ v_n), (e_1\ldots), \rho_\kappa\rangle :: \kappa\rangle, \text{ where} & & \textbf{[Ret]}\\
v_n = \mathcal{A}(ae, \rho, \sigma) & &
\end{array}
$$

Fig. 4. CESK machine operational semantics

of values, a list of unevaluated expressions, and an environment to evaluate them under. This particular frame contains the let body in a closure (with formal parameters from the LHS of the let form) as the first element in its value list, and any additional rhs expressions in the following list of expressions yet to be evaluated.

The **[App]** rules handles an application form by initiating evaluation of its first expression. The rest of the expressions are added to a new continuation frame, along with the current environment, to be evaluated later.

The **[EvalApply]** rule completes the evaluation of application sub-expressions. This case is distinguished from the later **[Ret]** rule by having a continuation with an *empty* list of unevaluated expressions atop the current stack. The list of values in the continuation is moved into a new apply state, extended by the final value being returned, and the top continuation frame is popped. The return (atomic) expression is evaluated with the atomic-expression evaluator. This rule does not step into a lambda expression, which simplifies the logic by separating evaluation from variable binding.

Binding formal variables to addresses in the environment, and addresses to values in the store, is the role of the **[ApplyEval]** rule. This identifies the formal parameters from the lambda expression of the first element in its list of values—the closure being invoked. The rest of the apply state's values are bound to these variables by extending the applied closure's environment and the apply state's current store by a set of *fresh* (never before used) addresses. The rule steps the apply state to an eval state for the lambda body under the updated environment and store.

The final rule is **[Ret]**, for the case where an eval state is returning a value to an incomplete continuation (a continuation with further expressions yet to be evaluated). The return expression is atomically evaluated and appended to the top continuation frame's value list, the next unevaluated expression becomes the new control expression, and the current environment is reverted to the one stored in the continuation.

To fully evaluate a program e_0 using these rules, we inject the program into an initial state $\varsigma_0 = (e_0, \varnothing, \varnothing, \epsilon)$. We perform the standard lifting of $(\to_\Sigma)$ to a collecting semantics over sets of reachable states $s \in S \triangleq \mathcal{P}(\Sigma)$. This collecting relation $(\to_S)$ is a monotonic, total function that yields a set of the trivially reachable initial state ς_0, plus the set of all states immediately succeeding those in its input.

$$s \to_S s' \iff s' = \{\varsigma' \mid \varsigma \in s \land \varsigma \to_\Sigma \varsigma'\} \cup \{\varsigma_0\}$$

If the program e_0 terminates, then iteration of $(\to_S)$ from $\bot$ (i.e., $\varnothing$) does as well. That is, $(\to_S)^n(\bot)$ is a fixed point containing e_0's full execution trace for some $n \in \mathbb{N}$ whenever e_0 is a terminating program. No such n

$$\hat{\varsigma} \in \hat{\Sigma} \triangleq \begin{cases} \text{eval: } \mathbf{Exp} \times \widehat{Env} \times \widehat{Instr} \times \widehat{Store} \times \widehat{KStore} \times \widehat{Kont} \\ \text{apply: } \hat{D}^* \times \widehat{Instr} \times \widehat{Store} \times \widehat{KStore} \times \widehat{Kont} \end{cases}$$

$$\hat{\rho} \in \widehat{Env} \triangleq \mathbf{Var} \rightharpoonup \widehat{Addr} \qquad \hat{k} \in \hat{K} \triangleq \mathcal{P}(\widehat{Kont})$$
$$\hat{\sigma} \in \widehat{Store} \triangleq \widehat{Addr} \to \hat{D} \qquad \hat{\kappa} \in \widehat{Kont} \triangleq \widehat{Frame}^* \times \widehat{Addr}$$
$$\hat{d} \in \hat{D} \triangleq \mathcal{P}(\widehat{Clo}) \qquad \hat{\psi} \in \widehat{Frame} \triangleq \hat{D}^* \times \mathbf{Exp}^* \times \widehat{Env} \times \widehat{Instr}$$
$$\widehat{clo} \in \widehat{Clo} \triangleq \mathbf{Var}^* \times \mathbf{Exp} \times \widehat{Env} \qquad \hat{\iota} \in \widehat{Instr} \text{ is a finite set}$$
$$\hat{\sigma}_\kappa \in \widehat{KStore} \triangleq \widehat{Addr} \to \hat{K} \qquad \hat{a}, \hat{a}_\kappa \in \widehat{Addr} \text{ is a finite set}$$

Fig. 5. Abstract abstract machine domains for λ_{sch}

is guaranteed to exist in the general case as our language is Turing complete, our semantics precise, and our state-space Σ is infinite.

2.2 Abstracting Abstract Machines

With a precise (incomputable) semantics of λ_{scm} in hand, we may design a computable approximation using abstract interpretation. Broadly, this process simultaneously finitizes the domains of our machine while introducing nondeterminism into the transition relation (a state can step to more than one successor state) and the store (an address can refer to more than one possible value). A finite domain of states (Σ) and state-spaces (S) means that our fixed-point calculation would have to terminate after some finite number of steps, however we require a defined notion of abstraction in order to show that a sound (valid) approximation of the original program is maintained by the abstract semantics.

Figure 5 shows domains for our abstract abstract machine. Typographically we add hats to domains that have changed so it is easy to see which have been abstracted. There were two sources of unboundedness in our concrete CESK machine: there were an unbounded number of addresses and so an unbounded value store, and the stack was modeled directly as an unbounded list. Both of these may be handled in the same way if we first store-allocate continuations [Van Horn and Might 2010], and then finitize our address space. Addresses must be abstracted to some finite set, a choice made by the allocator $\widehat{alloc}$ (discussed further in section 2.3), and so domains which contain abstract addresses (such as the environment and store) are now abstract by virtue of containing abstract addresses. We use a continuation store (from $\widehat{KStore}$) to map continuation addresses, $\hat{a}_\kappa$, to sets of continuations, which are now a list of frames followed by a continuation address. This address is allocated only during the **[AbsApplyEval]** rule, which allows us to retain precision for frames that do not involve function calls (and thus cannot possibly lead to unbounded extension of the stack). The other difference in the $\widehat{Frame}$ domain is the inclusion of instrumentation ($\widehat{Instr}$) as a general tunable parameter for extending states with contextual information (explained more below), but this parameter must be some finite set.

Figure 8 shows two important helper functions. The first is abstract atomic evaluation. As before, variables are accessed from the environment and store, however the store now maps addresses to sets of values, so atomic evaluation may also result in a set of values. Closure creation results in a singleton set containing one abstract closure. The second is a helper to retrieve the top frame of a given continuation—useful now that the continuation may either directly contain a topmost frame or may be an address referencing the rest of the stack via the continuation store.

$$\begin{aligned}
&\langle \textbf{eval}\ (\texttt{let}\ ([x_0\ e_0][x_1\ e_1]\ldots)\ e_b), \hat\rho, \hat\imath, \hat\sigma, \hat\sigma_\kappa, \hat\kappa\rangle \rightsquigarrow_\Sigma && \textbf{[AbsLet]}\\
&\quad\langle \textbf{eval}\ e_0, \hat\rho, \hat\imath', \hat\sigma, \hat\sigma_\kappa, \langle \textbf{frame}\ (\{\langle \textbf{clo}\ (x_0\ x_1\ldots), e_b, \hat\rho\rangle\}), (e_1\ldots), \hat\rho, \hat\imath'\rangle :: \hat\kappa\rangle, \text{where}\\
&\quad\hat\imath' = \widehat{\mathrm{tick1}}(\hat\imath, \hat\varsigma)\\
&\langle \textbf{eval}\ (e_0\ e_1\ldots), \hat\rho, \hat\imath, \hat\sigma, \hat\sigma_\kappa, \hat\kappa\rangle \rightsquigarrow_\Sigma \langle \textbf{eval}\ e_0, \hat\rho, \hat\imath', \hat\sigma, \hat\sigma_\kappa, \langle \textbf{frame}\ (), (e_1\ldots), \hat\rho, \hat\imath'\rangle :: \hat\kappa\rangle, \text{where} && \textbf{[AbsApp]}\\
&\quad\hat\imath' = \widehat{\mathrm{tick1}}(\hat\imath, \hat\varsigma)\\
&\langle \textbf{eval}\ ae, \hat\rho, \hat\imath_0, \hat\sigma, \hat\sigma_\kappa, \hat\kappa\rangle \rightsquigarrow_\Sigma \langle \textbf{apply}\ (\hat d_0\ldots\ \hat d_n), \hat\imath_2, \hat\sigma, \hat\sigma_\kappa, \hat\kappa'\rangle, \text{where} && \textbf{[AbsEvalApply]}\\
&\quad(\langle \textbf{frame}\ (\hat d_0\ldots), (), \rho_\kappa, \hat\imath_1\rangle, \hat\kappa') \in \widehat{\mathrm{lookup}}_\kappa(\hat\kappa, \hat\sigma_\kappa)\\
&\quad\hat d_n = \mathcal{A}(ae, \hat\rho, \hat\sigma)\\
&\quad\hat\imath_2 = \widehat{\mathrm{tick2}}(\hat\imath_0, \hat\imath_1, \hat\varsigma)\\
&\langle \textbf{apply}\ (\hat d_\lambda\ \hat d_0\ldots\hat d_n), \hat\imath, \hat\sigma, \hat\sigma_\kappa, \hat\kappa\rangle \rightsquigarrow_\Sigma \langle \textbf{eval}\ e, \hat\rho', \hat\imath, \hat\sigma', \hat\sigma_\kappa', \hat a_\kappa\rangle, \text{where} && \textbf{[AbsApplyEval]}\\
&\quad\langle \textbf{clo}\ (x_0\ldots x_n), e, \hat\rho\rangle \in \hat d_\lambda\\
&\quad\hat\rho' = \hat\rho[x_i \mapsto \hat a_i]\\
&\quad\hat\sigma' = \hat\sigma \sqcup [\hat a_i \mapsto \hat d_i]\\
&\quad\hat\sigma_\kappa' = \hat\sigma_\kappa \sqcup [\hat a_\kappa \mapsto \widehat{\mathrm{lookup}}_\kappa(\hat\kappa, \hat\sigma_\kappa)]\\
&\quad\hat a_i = \widehat{alloc}(\hat\varsigma, x_i)\\
&\quad\hat a_\kappa = (e,\ \hat\rho')\\
&\langle \textbf{eval}\ ae, \hat\rho, \hat\imath_0, \hat\sigma, \hat\sigma_\kappa, \hat\kappa\rangle \rightsquigarrow_\Sigma && \textbf{[AbsRet]}\\
&\quad\langle \textbf{eval}\ e_0, \hat\rho_\kappa, \hat\imath_2, \hat\sigma,\hat\sigma_\kappa, \langle \textbf{frame}\ (\hat d_0\ldots\ \hat d_n), (e_1\ldots), \hat\rho_\kappa, \hat\imath_1\rangle :: \hat\kappa'\rangle, \text{where}\\
&\quad(\langle \textbf{frame}\ (\hat d_0\ldots), (e_0\ e_1\ldots), \hat\rho_\kappa, \hat\imath_1\rangle, \hat\kappa') \in \widehat{\mathrm{lookup}}_\kappa(\hat\kappa, \hat\sigma_\kappa)\\
&\quad\hat d_n = \mathcal{A}(ae, \hat\rho, \hat\sigma)\\
&\quad\hat\imath_2 = \widehat{\mathrm{tick2}}(\hat\imath_0, \hat\imath_1, \varsigma)
\end{aligned}$$

Fig. 6. Abstract transition rules, $\hat\varsigma \rightsquigarrow_\Sigma \hat\varsigma'$, for λ_{scm}

$$\widehat{alloc}((e, \hat\rho, \hat\imath, \hat\sigma, \hat\sigma_\kappa, \hat\kappa), x) \triangleq (x, \hat\imath)$$

$$\widehat{tick1}(\hat\imath, \langle \textbf{eval}\ e, \hat\rho, (e_1, \ldots, e_k), \hat\sigma, \hat\sigma_\kappa, \hat\kappa\rangle) \triangleq \begin{cases}(e, e_1, \ldots, e_{k-1}) & e = (e_f\ \ldots)\\ (e_1, \ldots, e_k) & otherwise\end{cases}$$

$$\widehat{tick2}(\hat\imath_0, \hat\imath_1, \hat\varsigma) \triangleq \hat\imath_1$$

Fig. 7. Tunable Analysis Parameters

$$\hat{\mathcal{A}}(x, \hat\rho, \hat\sigma) \triangleq \hat\sigma(\hat\rho(x))$$
$$\hat{\mathcal{A}}((\lambda\ (x_0\ldots)\ e), \hat\rho, \hat\sigma) \triangleq \{\langle \textbf{clo}\ (x_0\ldots), e, \hat\rho\rangle\}$$

$$\widehat{\mathrm{lookup}}_\kappa(\hat\psi :: \hat\kappa, \hat\sigma_\kappa) \triangleq \{\hat\psi :: \hat\kappa\}$$
$$\widehat{\mathrm{lookup}}_\kappa(a, \hat\sigma_\kappa) \triangleq \bigcup_{\kappa \in \hat\sigma_\kappa(a)} \widehat{\mathrm{lookup}}_\kappa(\kappa, \hat\sigma_\kappa)$$

Fig. 8. Abstract atomic evaluation and continuation lookup

Figure 6 shows the transition rules for our abstract abstract machine. Each of the five rules corresponds to one of the concrete transition rules and serves the same purpose. Along with the abstract domains, there are several other differences. Instrumentation has been added to each state and continuation frame. An instrumentation is specific to a particular analysis, adding contextual information that can be used to increase analysis precision on a per-context basis. Instrumentation may be updated at each evaluation step through the $\widehat{\text{tick1}}()$ and $\widehat{\text{tick2}}()$ helpers (that take the entire state as input for flexibility/generality). Transition rules that use information from their continuation must first look it up in a continuation store. The result is a set of continuations, so the transition rule is followed for each one, producing a set of successor states. Likewise, where the closure being invoked at a call site is uncertain, one successor state will result for each possible closure. Addresses are generated from state data rather than being generated fresh. This provides a crucial opportunity to manage imprecision—more precise and unique addresses will yield a more precise model of the store and thus of the program as a whole. Finally, as stores no longer bind addresses to values, but to sets of values, new values to be bound are included along with any other values previously bound to that address. That is to say, join ($\sqcup$) between stores distributes point-wise.

Figure 7 shows tunings for our analysis parameters that correspond to Shivers' k-CFA family of analyses. This tracks the top-most k call sites reached on the stack via the instrumentation, and differentiates bindings for variables by both the variable name and this current calling context.

To analyze a program e_0 using these rules, we inject the program into an initial state $\hat{\varsigma}_0 = (e_0, \varnothing, (), \bot, \bot, \hat{a}_{\mathbf{halt}})$. We perform the standard lifting of $(\rightsquigarrow_\Sigma)$ to a collecting semantics over sets of reachable states $\hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$. This collecting relation $(\rightsquigarrow_S)$ is a monotonic, total function that yields a set of the trivially reachable initial state $\hat{\varsigma}_0$, plus the set of all states immediately succeeding those in its input.

$$s \rightsquigarrow_S \hat{s}' \iff s' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \land \hat{\varsigma} \rightsquigarrow_\Sigma \hat{\varsigma}'\} \cup \{\hat{\varsigma}_0\}$$

Iteration of $(\rightsquigarrow_S)$ from $\bot$ (i.e., ς) is guaranteed to terminate with a sound analysis of program e_0. That is, $(\rightsquigarrow_S)^n(\bot)$ is a fixed point containing a sound analysis of e_0 for some $n \in \mathbb{N}$.

Soundness. An analysis is called **sound** if the bound it provides on program behavior is accurate. The kind of control-flow analysis we've developed is a conservative over-approximation of program behavior that places an upper bound on the propagation of closures through the program and on edges in the control-flow graph. That our analysis is *sound* thus entails that if a closure can flow to a concrete address a, our analysis must indicate this same closure can flow to the abstract address for a, $\hat{a}$. Likewise, our final control-flow graph (CFG) cannot be missing any edges that are followed in any execution of the program—the CFG represents a *correct* upper bound on control-flow. To prove our abstract semantics soundly models our concrete semantics, we would define a family of abstraction functions α that map concrete entities to their most precise abstract corespondent, and would use this formal notion of abstraction to show that simulation is preserved across every transition:

$$\alpha(s) \sqsubseteq \hat{s} \land s \rightarrow_S s' \implies \hat{s} \rightsquigarrow_S \hat{s}' \land \alpha(s') \sqsubseteq \hat{s}'$$

2.3 Store Widening, IR, and Tunability

Global store widening. Various forms of widening and further approximations may be layered on top of the above analysis $(\rightsquigarrow)$. One such approximation is store widening, which is necessary for our analysis to be polynomial-time in the size of the program. This structurally approximates the analysis above, where each state contains a whole store, by pairing a set of reachable states without stores, with a single, global value store and continuation store that over approximates all possible bindings. These global stores are maintained as the least-upper-bound of all bindings that are encountered in the course of analysis.

$$\hat{\xi} \in \hat{\Xi} \triangleq \hat{R} \times \widehat{Store} \times \widehat{KStore} \qquad \hat{r} \in \hat{R} \triangleq \mathcal{P}(\hat{C}) \qquad \hat{c} \in \hat{C} \triangleq \mathbf{Exp} \times \widehat{Env} \times \widehat{Instr} \times \widehat{Kont}$$

We may formalize a widened analysis result as a 3-tuple, $\hat{\xi}$, that contains a set of reachable $(e, \hat{\rho}, \hat{\iota}, \hat{\kappa})$ 4-tuples, paired with a global value store $\hat{\sigma}$ and continuation store $\hat{\sigma}_\kappa$. We may interpret such an analysis result as a set of states $(e, \hat{\rho}, \hat{\iota}, \hat{\sigma}', \hat{\sigma}'_{\kappa}, \hat{\kappa})$ for all $e, \hat{\rho}, \hat{\iota}, \hat{\sigma}' \sqsubseteq \hat{\sigma}, \hat{\sigma}'_{\kappa} \sqsubseteq \hat{\sigma}_{\kappa}$, and $\hat{\kappa}$: each configuration, sans store, is implicitly paired with the two global stores (and all stores *less than* these).

Intermediate language. Setting up a semantics for other language features such as conditionals, primitive operations, first class continuations, or exceptions, is no more difficult, if somewhat more verbose. For example, supporting first-class continuations would require allowing $\hat{\kappa}$ values in the store. Handling new forms is often as straightforward as including an additional transition rule for each.

Allocation-based Polyvariance. Our previous sections formalized k-CFA, a *call sensitive* family of analyses that track the previous k call sites visited at each point (using our parametric instrumentation component, $\widehat{Instr}$) and differentiate bindings by this call history for added precision in each of these possible *contexts*. More generally, our instrumentation component, $\widehat{Instr}$, $\widehat{tick1}$ and $\widehat{tick2}$ functions for advancing this instrumentation at each step, and allocation function $\widehat{alloc}$ that generates addresses for each new binding, comprise a set of tunable parameters we may use to vary the style of polyvariance or context sensitivity used by the analysis [Gilray et al. 2016a]. By varying these components, we can recapitulate a wide variety of analysis styles and render each using our visualizer. We then use a very specific continuation address $a_{\kappa} = (e, \hat{\rho}')$ that is known to yield maximal stack-precision for whatever instrumentation and allocation strategy are chosen [Gilray et al. 2016b].

3 VISUALIZING AAM'S

Our goal is to use a visualization to understand the results of an AAM-based static analysis. In this section we will examine our goals and build up the intuition for our methodology. We first look at the interaction with a raw analysis output (represented as a data structure in Racket). We then work towards visualizing the analysis results as a directed graph (rather than textually), and highlight inherent tradeoffs as we build visualizations of the state space (Section 3.2). We conclude this section by presenting a conceptual sketch for how an analysis user might interact with our ideal visualization (Section 3.3).

3.1 Demo: Examining Variable Bindings

We must first understand how our visualization is to be used. An abstract interpretation (as in section 2.2) will eventually result in some final analysis output, a fixed point. In our case, this is a set of abstract states paired with a global value store and continuation store (a widened analysis result, $\hat{\xi}$). We study the execution of our abstract interpreter on the following small program:

```
((λ (x) x) (λ (y) y))
```

In particular, we want to use our analyzer to approximate which terms get bound to `x` within the first lambda expression. To do so, we invoke our evaluator by calling the function `explore`, which accepts two arguments: an analysis sensitivity (parameter k for k-CFA), and a term (represented as an S-expression) to explore.

```
(match-define (cons states data) (explore (cfa 0) ; specifies 0-CFA
                                          `((λ (x) x) (λ (y) y))))
```

The function `explore` performs fixed-point iteration to compute a set of final states and a collection of data about the analysis (including the final store). As a first step, we bind these results to the `states` and `data` variables. At this point, it is worth pointing out that the simplest possible visualization of all is perhaps simply just a textual representation of the analysis results:

```
> states
(set (list 'eval 'x #hash() '() 'halt)
     (list 'eval '(λ ...) '#hash() '() 'halt)
     (list 'eval '((λ (x) ...) ...) ...) ...)
```

States are represented as lists of a tag (`'eval` or `'apply`) specifying the type of state, followed the state's components. A textual representation allows us to recover the answer to anything the analysis can tell us, however, extracting useful information this way is particularly laborious. For example, we might want to know which closures eventually reach `x` within the above program (the closure for `(λ (y) y)` in this case). To recover this from the textual analysis results, we could fold over the set of states produced by the analysis and examine the states in which the control expression is `x`. From there we could we could examine the environment to determine the address for `x`, which we would then use as an index into the global store.

3.2 Challenges and Tradeoffs in Visualizing AAMs

Using a textual interface to examine properties of an abstract interpretation is useful in some scenarios; for example, debugging an analysis. However, a textual representation is often cumbersome to interact with and is less enabling of open-ended exploration of the analysis results. This is because the textual representation flattens and obfuscates the inherently graph-based and relational structure in th

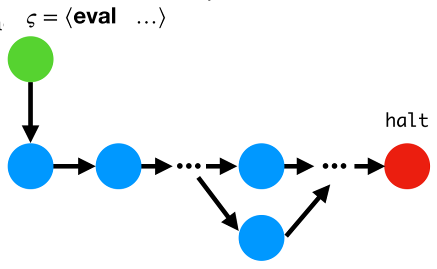
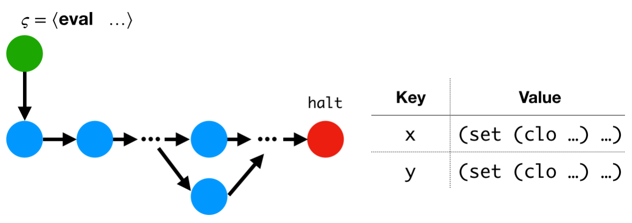


Fig. 9. Visualization mockup

A First Attempt. As a first attempt we generate a graph like the one on the right, which is simply the control-flow graph produced by the analysis. The visualization is rooted at the initial state (colored green, under ς), an eval state for the top-level expression. But there is still a problem: how do we usefully render the data at each state? The challenge lies in the fact that much of the information stored However—as each state in the graph it is not enough to look at the state graph in isolation.

Challenge: Visualizing the Store. There are several options available to us. For example, we could visualize the store alongside the graph: However, this requires the user to manually inspect the store and perform lookup on-demand. While this is an obvious inefficiency, visualizing the store is challenging due to its tightly interwoven nature. Another solution is to have the visualization look up values from the store in certain scenarios. For example, when visualizing the result of an analysis we likely prefer a visualization that displays environments in the form $\left\{x \mapsto \left\{\left\langle(\lambda(x)\ x), \ldots\right\rangle, \left\langle(\lambda(y)\ z), \ldots\right\rangle\right\}, z \mapsto \ \ldots\right\}$ rather than $\{x \mapsto \alpha_1, z \mapsto \alpha_2\}$, leaving the user to lookup each address manually.

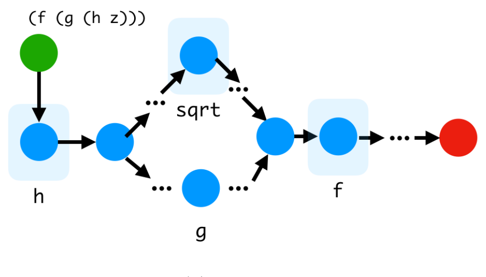


Key	Value
x	(set (clo …) …)
y	(set (clo …) …)

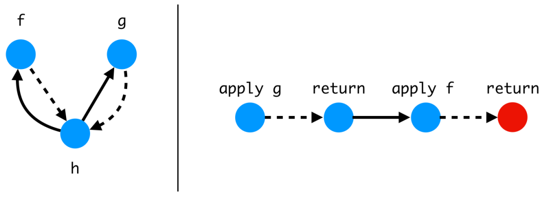
Fig. 10. Visualization mockup with environment

Tradeoff: When Should we Inline? A trade-off has been made in the first attempt: the closures contained within the set of results for `x` also contain environments. We must ask whether to further inline environments, and if so to what depth. In general inlining environments will not terminate (as the store may contain closures with loops or mutual dependences). Another strategy might allow the user to interactively inline environments (e.g., via clicking) or automatically unroll top-level environments and allow the user to interactively inline subsequently encountered nested environments.

Challenge: Controlling State Explosion. For relatively small programs—such as the one in our above example—the state graph is also small. The approach we described above rapidly breaks down for programs beyond a few



(a)



(b)

Fig. 11. Visualization mockup (functional components, 11a), and Panes for intra- and inter-procedural state views (11b).

lines. The main problem is that the state graph shows a global view of the program's execution: states from every function in the program are shown together into a single graph. This can be useful in some scenarios, however, execution graphs will, in general, become overwhelmingly large. This problem is compounded when the analysis is made more precise (e.g., by selecting 2-CFA rather than 0-CFA). One key trade-off for our visualization is deciding to what degree parts of the graph are elided to focus on a specific component.

One strategy is to visualize the execution of each function in isolation. This allows us to leverage the procedural abstraction inherent in the analyzed program's structure. For example:

```
(define (f x) ((λ (y) y) ((λ (x) x) x)))
(define (g x) (... (sqrt (max x) ...)))
(define (h x) (...))
(f (g (h z)))
```

If we visualize the above program and focus only on the execution of g, we can collapse states from other functions into to a single point (see figure 11a). Isolating our visualization of the analysis to a specific function at a time allows the user to focus on how each functional component interacts with the program at large.

3.3 Our Strategy for Visualizing AAMs

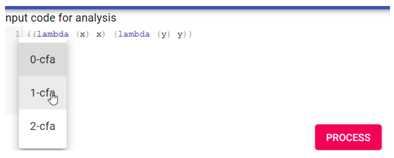
Our visualization consists of two high-level views presented in separate physical panes. The first view is a per-function control-flow graph that shows the intraprocedural analysis of a selected function. The second view is an interprocedural call graph. A mockup of said approach appears in Figure 11b.

In this visualization, h's intraprocedural execution graph is shown on the right. Because h calls both f and g, its intraprocedural execution is fairly straightforward (we elide the states which simply evalulate the variables f and g): first apply g, then apply f on the result, then return.

Our visualization includes a key interactive feature that allows switching between functions: clicking on a dotted call edge will highlight the corresponding target function in the interprocedural call graph. Clicking a different function in the interprocedural call graph will swap the current function being displayed in the interprocedural graph. For example, clicking on the edge for g will highlight g's vertex in the call graph, and clicking on g in the call graph will swap to displaying the execution of g (which simply returns x).

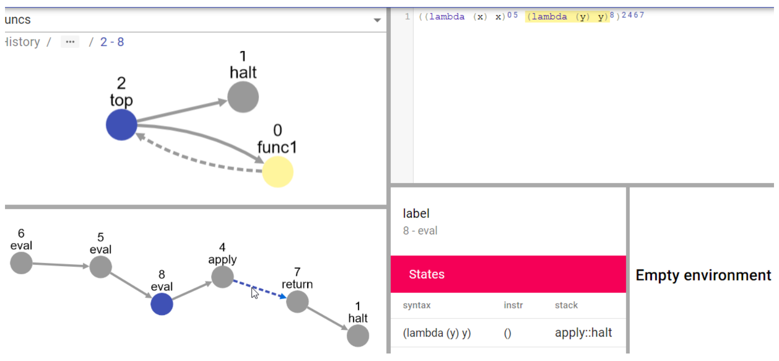
4 IMPLEMENTATION

The visualizer is a client web application that interfaces with a server to handle API requests and analysis processing. Users interact with the client application to submit code to the server for processing and to visualize the resulting analysis. To edit code, users can fork a project, make changes, and resubmit modified code.



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

(a)



```
((lambda (x) x)05 (lambda (y) y)8)2467
```

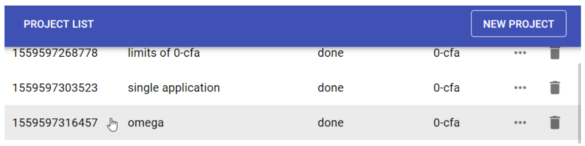
syntax	instr	stack
(lambda (y) y)	()	apply::halt

(b)

Fig. 13. Edit and submit a new project (13a), Project view (13b).

4.1 Application workflow

The web client is a reactive JavaScript application using React.js. The open-source Cytoscape.js and CodeMirror libraries are used for graph visualization and code syntax highlighting, respectively. Our NodeJS-powered server utilizes Express.js for routing and stores each project in memory as well as on the disk in a local `/data/` folder. We use a custom Racket codebase to run the analysis, which is then cached to be delivered to the client on request.



1559597268778	limits of 0-cfa	done	0-cfa
1559597303523	single application	done	0-cfa
1559597316457	omega	done	0-cfa

Fig. 12. Project list view

The client application has three main views: project list, project editing, and project visualization. The project list view shows an overview of all projects on the server. When a project is created, it may be edited in an editing view before being submitted for processing. Once a project is submitted it becomes immutable and the client application switches to a visualization mode, then downloads and displays the project's analysis using a split-pane view.

When the web app loads, it begins at the project list view, shown in figure 12. Here we see each project as well as options to delete or fork it. To create an empty project, we click on the top right button. The new project will then be added to the list. We can select the project to enter it's editing page.

Because the project has not been submitted yet for processing, we are able to edit the project and provide code for analysis. We can return to the project list view at any time by clicking the top left button. Existing code will be saved to the server. The main text box is for input of code and analysis options are available on the bottom toolbar, as seen in figure 13a. To initiate analysis of the code, we can click on the bottom right button. The project list will update when the project has finished analysis.

Figure 13b shows the project view of an identity function applied on an identity function. All panes within this view are resizeable. The top-left pane shows an interprocedural call graph; the bottom-left pane shows a function graph for the component selected in the top-left pane. Clicking on dashed edges in the function graph will highlight called nodes in the call graph. We can switch to the whole-program CFG by clicking the selection button at the top of the graphs.

The code pane at the top right, while immutable to changes, is still interactive. Clicking on the numbered superscript marks will select the corresponding graph node. Likewise, selecting nodes in the graphs will highlight relevant code. The bottom right pane visualizes the selected node's subsumed states and environment.

$$
\begin{aligned}
\textbf{output} &\triangleq \mathit{fid} \times \mathit{MGraph} \times \mathit{Graphs} & \mathit{FGraph} &\triangleq \mathit{FState} \rightarrow \mathcal{P}(\mathit{FTrans}) \\
\mathit{MGraph} &\triangleq \mathit{fid} \rightarrow \mathcal{P}(\mathit{MTrans}) & \mathit{FTrans} &\triangleq \mathit{FState} \times \mathit{Tran} \\
\mathit{MTrans} &\triangleq \mathit{Final} \mid \mathit{fid} \times \mathit{Tran} & \mathit{Final} &\triangleq \textbf{finalinfo} \times \mathcal{P}(\mathit{fid}) \\
\mathit{Graphs} &\triangleq \mathit{fid} \rightarrow \mathcal{P}(\mathit{FState} \times \mathit{FGraph}) & \mathit{Tran} &\triangleq \textbf{transitioninfo} \times \mathcal{P}(\mathit{fid}) \\
\mathit{FState} &\triangleq \varsigma \mid \mathcal{P}(\varsigma) \mid \mathit{Final} & &\mathit{fid} \text{ is a label for a function}
\end{aligned}
$$

Fig. 14. Segmentation output

4.2 Segmentation Algorithm

After our AAM-based analysis runs on the server, we post-processing an analysis in phase we call *segmentation*. Segmentation uses the main analysis as its input data and, along with information about the analysis implementation, produces multiple graphs. One graph is produced for each lambda expression that is the call-target of an application expression. Another graph is produced with nodes representing these lambda expressions and edges representing their calls to, and returns from, one another.

Segmentation proceeds in four main stages: identifying calls, identifying returns, building intraprocedural graphs, and compiling info from these into a summary graph. The identification stages each make a pass over the analysis states, caching relevant information. Calls are entry points to functions, the state following the completion of an application. They are identified by the function they enter. Returns are states following atomic expression states, the continuation of which serves as their identification.

To build our intraprocedural graphs we process entries from our cached calls. Each provides a set of states that entered a function. We consider this a single super-state in our graph. In the general case, we step each element of this set to the next state in the main analysis, collecting the results into a set that serves as the next super-state. Special cases are applications and atomic expressions, which will mark the entries and exits to other functions. This data is collected to be used in the summary graph. Atomic expressions are always function exits, and we step their main analysis states to determine the function they exit to (or identify a halt state instead). Applications evolve by stepping into another function and then (potentially) returning from it. We step the main analysis states to determine which function they enter and to retrieve the continuation from that entry. We look up the continuation in our global returns cache to get our next super-state. If there are no returns we produce a special state to signify this fact. Each intraprocedural graph is produced in turn and their entry points are used as nodes in our interprocedural CFG of components. We make edges from their calls and returns. We also produce special nodes for each exit point, either halt states or error states.

The output of this algorithm is formalized in figure 14. It is a tuple of top-level function id, the main (interprocedural) graph, and the function graphs. Function ids (*fid*) are used throughout to select a particular function (interprocedural) graph. The main graph (*MGraph*) is a mapping from function ids to a set of transitions (*MTrans*). These include the id of the next state as well as some transition info (*Tran*). In the main graph we currently only use the **transitioninfo**, an identifier for the transition (e.g. whether it's a call or return). The rest of the *Tran* info is more useful in the function graphs. Main graph transitions can also be to a *Final* state, which serves as its own transition information as well. Final states may be a halt or stuck state, for example. Along with the main graph in the output we have a number of function *Graphs*, any one of which can be selected with a function id. This id maps to a pair of the initial state of the graph and the graph itself. As with the main graph, the function graphs (*FGraph*) are mappings from states to a set of states with transition information. Here we can make use of the set of function ids included with *Tran*. Transitions from one state to the next in a function graph could be a call and/or return from other functions or exit to another function, so we include their ids here.

A state in a function graph, shown in figure 14 as *FState*, can be a single state from our main AAM analysis, a set of those states, or a final state. Final states here are similar to the main graph final states described above, and are shown in the function graphs (to show where in the evolution of states they occur). In some cases we use single states ς as final states in function graphs, to provide more information about this result. In most cases, however, we subsume a sets of states ς with one node in the function graph. This is possible because the syntax of a function leads its development regardless of, e.g., the information in its store or the instrumentation used. This specific information is still available in the individual states, but they flow together until reaching a branch point (such as a function call). Our current λ_{scm} does not have intra-procedural branching, but including it is future work, and would require segmenting the set of states.

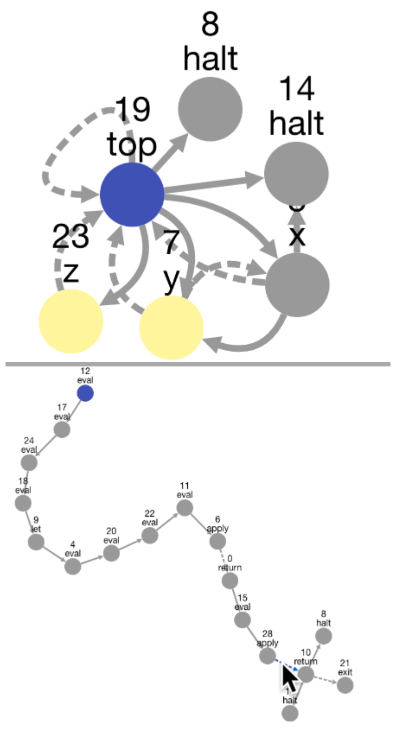


Fig. 15. View of analysis with highlighted functions

4.2.1 Function graph transitions. Generating transitions between functions involves several cases. These states comprise a set of states ς, a super-state, of our primary AAM analysis. All eval states should have the same expression, though imprecision in the analysis may interfere with this at branch points. In cases that are not call or return states, we generate the state transitions, unioning the results to produce the new super-state.

Return states are those that evaluate atomic expressions, or are labeled as returning through a tail-call. For these we generate the next states and compute the id of the function they are part of (this data is cached during parsing and the main analysis). We generate an exit transition for each function id. Some of these function ids may indicate that the exit also concludes the top-level function, so we generate a halt transition. We may also generate a stuck state transition here if the id cannot be computed.

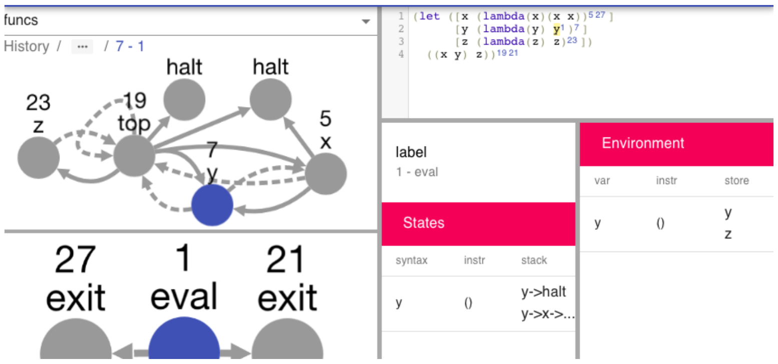
Call states are those that would transition with the **[AbsApplyEval]** rule. We generate all the next states and then separate them into those that continue and those that do not. Stuck states do not continue, but for the rest we must check with our global cache of return states described above. The goal here is to transition directly to the return of the function called in the state, bypassing all intermediate super-states. We use the continuation address of the state we called into as the key in the return cache, providing a set of states returning here. If it is an empty set, then there was no return to this function (generally the result of a non-terminating loop) and we create a no-return transition. Otherwise, we union all the return states as the new super-state, and all the function ids we called into as transition data. This data can be read by our visualizer to highlight called functions.

4.3 Analysis walkthrough

Lets take a look at an example expression and explore its analysis with our visualization. We'll use the code below because it's a simple illustration of imprecision in an analysis affecting results.

```
(let ([x (lambda (x) (x x))]
      [y (lambda (y) y)]
      [z (lambda (z) z)])
  ((x y) z))
```

Figure 15 shows the view of the left panes after analyzing with 0-CFA (highlighting is explained below). The top-left pane has the central node selected (representing the top-level expression). The bottom pane shows a



```
(let ([x (lambda(x)(x x))5 27 ]
      [y (lambda(y) y1 )7 ]
      [z (lambda(z) z)23 ])
  ((x y) z))19 21
```

label
1 - eval

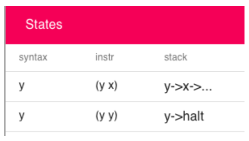
States

syntax	instr	stack
y	()	y->halt y->x->...

Environment

var	instr	store
y	()	y z

(a)



States

syntax	instr	stack
y	(y x)	y->x->...
y	(y y)	y->halt

(b)

Fig. 16. Full view of visualizing "y" (16a) and States in 2-cfa (16b).

progression of sequential states, starting at the top and ending with multiple exits. The first few states are labeled as "eval" followed by a "let". These correspond to evaluating each of the lambdas in the let-form into closures, followed by binding them to the given (LHS) variables. This particular syntax also provides names to these closures which the visualization uses to represent them (e.g., `x` denotes `(λ (x) ...)`). Next, the body of the let-form is evaluated in multiple steps before applying `x` to `y`. Here we see a dotted arrow leaving an "apply" state, connecting it to a "return" state. Selecting this edge (or any dotted arrow) will highlight the invoked function (or functions) in the upper pane. After a few more "eval" states, we find another "apply"-"return" pair, corresponding to the second application in the code. Figure 15 shows the visualizer when we click on the dotted arrow between them, highlighting the two functions on the upper pane. We could have also clicked on the small number "1" in the text pane in the upper right—clicking these markers selects states in either graph.

Here we may note, via manual evaluation of the code, that the identity function labeled "z" is never called, but the analysis highlights it as if it were. This is an artifact of imprecision in 0-CFA. Since the function called here is the result of "y", an identity function, *any* result of "y", and therefore parameter of "y", could be called. "z" is a parameter of "y" at some point, so it appears it could be called here. We can see this information by clicking on "y" in the visualization, shown in figure 16a. In this expanded view, we see the sub-states composing the entry point to "y", along with the environment of one of them. It shows this function sitting atop multiple stacks (corresponding to both calls in the concrete evaluation), which gives two possible values for `y` in the environment. These multiple elements are on different lines in the last column of one row in their table. This lack of precision can be fixed by forking the project and running a more polyvariant analysis, such as 2-CFA. Looking at the states of "y" at this point will show what is in figure 16b, two different states, each of which has only one continuation.

5 CONCLUSION

This paper focuses on exploring AAM-style program analyses. While very useful, they can be difficult to understand in their raw form. Analysis designers and others who can't rely on the simplest outputs need methods of studying whole analyses. We present one such method, building on tunable AAM for maximum flexibility. We build a visualization focused on individual functions. This presents the user with multiple forms of information in small chunks, from standard code view to summary graphs and state-specific environments. We believe this to be useful now, and look forward to extending its capability in the future.

REFERENCES

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Paris, France, 106–130.

Patrick Cousot and Radhia Cousot. 1977a. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, Los Angeles, CA, 238–252.

Patrick Cousot and Radhia Cousot. 1977b. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *ACM Sigplan Notices* 12, 8 (1977), 1–12.

Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming (Leuven, Belgium, 13-17 August 1992, Lecture Notes in Computer Science 631)*. Springer-Verlag, Berlin, Germany, 269–295.

Thomas Gilray, Michael D. Adams, and Matthew Might. 2016a. Allocation Characterizes Polyvariance: a unified methodology for polyvariant control-flow analysis. *Proceedings of the International Conference on Functional Programming (ICFP)* (September 2016).

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016b. Pushdown Control-Flow Analysis For Free. *Proceedings of the Symposium on the Principals of Programming Languages (POPL)* (January 2016).

Matthew Hennessy. 1990. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons.

Matthew Might. 2007. *Environment Analysis of Higher-Order Languages*. Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA.

Matthew Might. 2010. Abstract Interpreters for free. In *Static Analysis Symposium*. 407–421.

Gordon D Plotkin. 1981. *A structural approach to operational semantics*. Technical Report. DAIMI Arhus, Denmark.

Olin Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York, NY, 164–174.

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie-Mellon University, Pittsburgh, PA.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *International Conference on Functional Programming*. 51.

Glynn Winskel. 1993. *The formal semantics of programming languages: an introduction*. MIT press.