Stack-Liberated Abstract Garbage Collection

KIMBALL GERMANE, Brigham Young University
MICHAEL D. ADAMS, University of Utah

Properly devised, stack-precise control-flow analyses—those that model control using a pushdown system—exhibit the natural and appealing property of *context irrelevance* under which the result of an evaluation does not depend on its continuation. The addition of abstract garbage collection to such analyses destroys context irrelevance as the precise abstract transitions taken during the evaluation of the expression depend on (the root set of reachability contained in) the continuation. This paper reconciles context irrelevance and abstract garbage collection by narrowing the scope of the store to just the local environment. Within this narrowed scope, garbage collection is trivially straightforward. However, with the store no longer acting as a global repository of effects, we require an explicit channel between evaluations to communicate effects. We define an effect log for each evaluation to act as this channel, which we can apply to explicitly propagate effects. After the dust settles, we're left with a fully-compositional analysis that exhibits the benefits of perfect stack precision, garbage collection, and context irrelevance.

ACM Reference Format:
Kimball Germane and Michael D. Adams. 2019. Stack-Liberated Abstract Garbage Collection. *Proc. ACM Program. Lang.* Scheme, 1 (July 2019), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Abstract garbage collection [Might and Shivers 2006] and the use of pushdown systems [Earl et al. 2010; Vardoulakis and Shivers 2011] each enhance the precision of control-flow analysis (CFA). Abstract garbage collection reduces the use of abstract heap space, thereby avoiding the conflation of distinct heap data by CFA. The use of pushdown systems to model control flow allows abstract returns to be matched precisely with their corresponding calls, thereby avoiding the conflation of distinct control paths by CFA.

A natural question is whether these enhancements can be combined so that a CFA can benefit from both simultaneously. A difficulty arises because a pushdown analysis decreases the information the analysis can obtain about the stack at any given point while abstract garbage collection demands fairly comprehensive access to the stack to calculate the root set of reachability. Earl et al. [2012] overcome this difficulty with an intricate mechanism to provide the garbage collector with an overapproximation of the root set without revealing the precise structure of the stack. Thus, they are able to reconcile the tension between these two enhancements and the resultant analysis simultaneously enjoys the increased precision each offers.

However, this reconciliation comes at the cost of *context irrelevance* [Johnson and Van Horn 2014], the property that a given evaluation, absent control effects, should be independent of its continuation. (In general, stack-precise CFAs—i.e. CFAs that model control flow using pushdown systems—natively exhibit this property.) Context irrelevance allows us to, e.g., reason locally about evaluation and memoize our (abstract) interpreters. The issue is that, contrary to the concrete

Authors' addresses: Kimball GermaneBrigham Young University; Michael D. AdamsUniversity of Utah.

2475-1421/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

setting, garbage collection is semantically relevant in the abstract setting: its presence influences the state space explored by the analysis. Hence, an approach to abstract garbage collection which relies on the stack (e.g. for reachability roots) causes evaluation (state-space exploration) to depend on the continuation (stack) and thereby violate context irrelevance.

This paper introduces an alternative reconciliation of abstract garbage collection and pushdown systems which preserves context irrelevance.

The key idea is to narrow the scope of the store to only its associated configuration—expression and closing environment. So narrowed, the store does not close the continuation and therefore garbage collection does not need to obtain reachability roots from the stack to proceed. Of course, the continuation needs to be closed to preserve future evaluation. Our key insight is that, when control flow is modelled by a pushdown system (so that returns are precisely matched to their calls), the store of the caller is available at the return point. This store closes the caller frame, making it unnecessary for the callee store to close it. By similar reasoning, the caller's store itself need not close the frame of the caller's caller. Inductively, each stack frame is closed by its own store.

When the role of the store is limited in this way, the store of the caller is restored in a particular sense when the call returns. In other words, this role induces a compositional treatment of the store (as opposed to threading it through evaluation). Of course, when the store is no longer threaded through evaluation, it can no longer be used to accumulate bindings and communicate effects, such as mutation. We fill this gap by communicating effects through an effect log produced by each evaluation. This log can be applied to the caller's store to replay the relevant effects performed during evaluation of the call. Effect logs are compositional also: when evaluating compound expressions, we can compose the effect logs of the subexpressions' evaluation to obtain the effect log of the overall expression's evaluation.

Once we have made this change in the analysis, the environment and store are each treated compositionally. But we observe that, almost naturally, we treat the abstract time compositionally as well, rather than threading it through evaluation. The effect of this treatment is a shift from the last-k-call-sites context abstraction of k-CFA [Shivers 1991] to the top-m-stack-frames abstraction of m-CFA [Might et al. 2010]. This appearance of m-CFA's context abstraction is, to our knowledege, the first in a setting with perfect stack precision.

In the end, by treating each of the machine components compositionally, we are able to craft a control-flow analysis that enjoys the benefits of perfect stack precision, garbage collection, context irrelevance, and the highly-effective m-CFA context abstraction.

We introduce the core language we will use throughout the paper in the next section. We review abstract garbage collection in Section 3 by adding it to a small-step semantics for our language. We then transition toward an alternative approach to garbage collection through a series of big-step semantics in Section 4. In Section 5, we abstract the culmination of this series to obtain a sound, computable control-flow analysis using this alternative approach. We discuss related work in Section 6 and conclude in Section 7.

2 A-NORMAL FORM λ-CALCULUS

For presentation, we keep the language small: we use a unary λ-calculus in $\mathcal{A}$-normal form [Flanagan et al. 1993], the grammar of which is given below.

$$\begin{aligned}
\mathit{Exp} \ni e &::= ce \mid \mathsf{let}\, x = ce \,\mathsf{in}\, e \\
\mathit{CExp} \ni ce &::= ae \mid (ae_0\ ae_1) \mid \mathsf{set!}\, x\ ae \\
\mathit{AExp} \ni ae &::= x \mid \lambda x.e \\
\mathit{Var} \ni x & \quad \text{[an infinite set of variables]}
\end{aligned}$$

A proper expression e is a let-expression or a *call expression ce*. A let-expression binds a variable to the result of a call expression. (Restricting the bound expression to a call expression prevents let-expressions from nesting there, a hallmark of $\mathcal{A}$-normal form.) A call expression e is an application, a set!-expression, or an *atomic expression ae*. An atomic expression ae is a variable reference or a λ abstraction.

Atomic expressions are trivial [Reynolds 1998]. We include set!-expressions to produce mutative effects that must be threaded through evaluation. For our purposes, we consider a set!-expression "serious" [Reynolds 1998] since it has an effect on the store.

A program is a closed expression; we assume (without loss of generality) that programs are alphatised—that is, each bound variable has a distinct name.

We will refer to the set of application expressions by *App* at various places throughout the paper.

3 ABSTRACT GARBAGE COLLECTION

Might and Shivers [2006] introduced abstract garbage collection in a stack-imprecise CFA[1] of a small-step semantics. Earl et al. [2012] ported abstract garbage collection to a stack-precise CFA of a small-step semantics. In this section, we define a small-step semantics over our language and add garbage collection to it. The resultant small-step semantics with garbage collection will then serve as a common framework in which to discuss details of stack-imprecise and stack-precise CFAs within it.

We first establish some semantic domains in the next subsection. Elements of these domains appear in the following sections as well as Section 4. We then formally define a small-step semantics over our language using these domains. We then add abstract garbage collection to our semantics. Once we've reached this point, we discuss stack-imprecise and stack-precise CFAs of it.

3.1 Semantic Domains

Following are some semantic components that we will use heavily throughout the rest of the paper.

$$v ::= (\lambda x.e, \rho) \qquad \rho \in \mathit{Env} = \mathit{Var} \rightharpoonup \mathit{BindContext}$$

$$c \in \mathit{BindContext} = \mathit{App}^* \qquad \sigma \in \mathit{Store} = \mathit{Var} \times \mathit{BindContext} \rightharpoonup \mathit{Val}$$

$$\mathit{Cont} \ni \kappa ::= \mathsf{mt} \mid \mathsf{lt}(x, \rho, e, c, \kappa)$$

A value v is closure, a pair of a λ abstraction and an environment which closes it. An environment ρ is a finite map from variables x to *binding contexts* c. Let $\rho|_e$ denote the restriction of the domain of the environment ρ to the free variables of e. A binding context c is a finite sequence of call sites. A store σ is a map from bindings to values where a binding consists of a variable and a context. A continuation κ is either the empty continuation or the continuation of a let binding.

3.2 A Small-Step Abstract Machine Semantics

We define our first concrete semantics as a small-step relation over abstract machine states. The state space of our machine is given formally as follows.

$$\begin{aligned} \varsigma &\in \mathit{State} = \mathit{Eval} + \mathit{Apply} \\ \varsigma_{\mathrm{ev}} &\in \mathit{Eval} = \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Cont} \times \mathit{BindContext} \\ \varsigma_{\mathrm{ap}} &\in \mathit{Apply} = \mathit{Store} \times \mathit{Cont} \times \mathit{Val} \end{aligned}$$

Machine states come in two variants. An *Eval* machine state represents a point in execution in which an expression will be evaluated; it contains registers for an expression e, its closing environment ρ, the store σ, the continuation κ, and the binding context c. An *Apply* machine state represents a

[1]By which we mean a CFA which does not precisely match returns to the corresponding calls.

$$\textsc{Let}\quad \frac{}{\mathsf{ev}(\mathsf{let}\,x = ce\,\mathsf{in}\,e, \rho, \sigma, \kappa, c) \rightarrow_{\mathsf{ev}} \mathsf{ev}(ce, \rho, \sigma, \mathsf{lt}(x, \rho, e, c, \kappa), c)}$$

$$\textsc{Call}\quad \frac{(\lambda x.e, \rho') = \mathsf{aeval}(\sigma, \rho, ae_0) \qquad v = \mathsf{aeval}(\sigma, \rho, ae_1) \qquad c' = (ae_0\; ae_1) :: c \qquad \sigma' = \sigma[(x, c') \mapsto v] \qquad \rho'' = \rho'[x \mapsto c']}{\mathsf{ev}((ae_0\; ae_1), \rho, \sigma, \kappa, c) \rightarrow_{\mathsf{ev}} \mathsf{ev}(e, \rho'', \sigma', \kappa, c')}$$

$$\textsc{Set!}\quad \frac{v = \mathsf{aeval}(\sigma, \rho, ae) \qquad a = (x, \rho(x)) \qquad \sigma' = \sigma[a \mapsto v]}{\mathsf{ev}(\mathsf{set!}\,x\; ae, \rho, \sigma, \kappa, c) \rightarrow_{\mathsf{ev}} \mathsf{ap}(\sigma', \kappa, (\lambda \mathrm{x.x}, \bot))}$$

$$\textsc{Atomic}\quad \frac{v = \mathsf{aeval}(\sigma, \rho, ae)}{\mathsf{ev}(ae, \rho, \sigma, \kappa, c) \rightarrow_{\mathsf{ev}} \mathsf{ap}(\sigma, \kappa, v)}$$

$$\textsc{Apply}\quad \frac{\rho' = \rho[x \mapsto c'] \qquad \sigma' = \sigma[(x, c') \mapsto v]}{\mathsf{ap}(\sigma, \mathsf{lt}(x, \rho, e, c', \kappa), v) \rightarrow_{\mathsf{ap}} \mathsf{ev}(e, \rho', \sigma', \kappa, c')}$$

Fig. 1. Small-step abstract machine semantics

point in execution at which a value is in hand and must be delivered to the continuation; it contains registers for the store σ, the continuation κ, and the value v to deliver.

Figure 1 contains the definitions of two relations over machine states, the union of which constitutes the small-step relation. The $\rightarrow_{\mathsf{ev}}$ relation transitions an *Eval* state to its successor. The Let rule pushes a continuation frame to save the bound variable, environment, body, and binding context. The resultant *Eval* state is poised to evaluate the bound expression *ce*. The Call rule uses aeval to obtain values for each of the operator and argument. After deriving a new binding context, it extends the store and environment with the new context and arranges for evaluation of the operator body in the new context. The Set! rule remaps a location in the store designated by a given variable (which is resolved in the environment) to a value obtained by aeval. It returns the identity function. The Atomic rule evaluates an atomic expression.

$$\mathsf{aeval}(\sigma, \rho, x) = \sigma(x, \rho(x)) \qquad\qquad \mathsf{aeval}(\sigma, \rho, \lambda x.e) = (\lambda x.e, \rho|_{\lambda x.e})$$

The Apply rule applies a continuation to a value, extending the environment and store, restoring the binding context, and arranging for the evaluation of the let body.

It's worth mentioning which quantities we thread through evaluation and which we treat "compositionally". The store and continuation are threaded through evaluation; each transition passes each of them, or some close derivative, to the next state. On the other hand, environments are treated compositionally. This is necessary because the language is lexically-scoped, so the environment within an evaluation context must be restored when evaluation concludes. In a k-CFA, a timestamp which is threaded through evaluation serves as the binding context. Here, rather than threading a timestamp, we introduce a new binding context for the evaluation of procedure bodies but restore the old when the procedure returns. By treating it this way, we obtain the m-CFA context abstraction [Might et al. 2010] which is arguably superior in general to the k-CFA abstraction. We will continue to treat it compositionally even after the transition to a stack-precise CFA, yielding *pushdown m-CFA*, a combination which has not appeared in the literature to our knowledge.

3.3 Garbage-Collecting Concrete Semantics

We now extend the semantics presented in the previous section with garbage collection. This extension will have no effect on the behavior of programs—at least in this exact setting.

We perform garbage collection on *Apply* states where root set of reachability lies in the continuation and value components. We define a family root of functions to extract the reachability root set from these and related components.

$$\mathsf{root}_v(\lambda x.e, \rho) = \mathsf{root}_\rho(\rho) \qquad \mathsf{root}_\kappa(\mathsf{mt}) = \emptyset$$
$$\mathsf{root}_\rho(\rho) = \rho \qquad \mathsf{root}_\kappa(\mathsf{lt}(x, \rho, e, c, \kappa)) = \mathsf{root}_\rho(\rho|_e) \cup \mathsf{root}_\kappa(\kappa)$$

The root_v metafunction extracts the root addresses from a closure by using root_ρ to extract the root addresses from its environment. By the root_ρ metafunction, the root addresses of an environment are simply the variable–context pairs that define it. That is, the definition of root_ρ views its argument ρ extensionally as a set of variable–context pairs. The root_κ metafunction extracts the root addresses from a continuation. The empty continuation has no root addresses whereas the root addresses of a non-empty continuation are those of its stored environment combined with those of the continuation it extends.

We then define a reachability relation $\rightarrow_\sigma$ over addresses parameterized by a store σ defined by

$$a_0 \rightarrow_\sigma a_1 \Leftrightarrow a_1 \in \mathsf{root}_v(\sigma(a_0))$$

We then define the reachability of a root set with respect to a store

$$\mathcal{R}(\sigma, A) = \{a' : a \in A, a \rightarrow_\sigma^* a'\}$$

where $\rightarrow_\sigma^*$ is the reflexive, transitive closure of $\rightarrow_\sigma$. From here, we obtain the transition

$$\text{GC}\quad \frac{A = \mathsf{root}_\kappa(\kappa) \cup \mathsf{root}_v(v) \qquad \sigma' = \sigma|_{\mathcal{R}(\sigma,A)}}{\mathsf{ap}(\sigma, \kappa, v) \rightarrow_{\mathsf{GC}} \mathsf{ap}(\sigma', \kappa, v)}$$

where $\sigma|_{\mathcal{R}(\sigma,A)}$ is σ restricted to the reachable addresses $\mathcal{R}(\sigma, A)$. We compose this garbage-collecting transition with the standard *Apply* transition so that the garbage-collected semantics are $\rightarrow_{\mathsf{ev}} \cup \rightarrow_{\mathsf{GC}} \circ \rightarrow_{\mathsf{ap}}$.

3.4 Stack-Imprecise CFA with Garbage Collection

With a small-step abstract machine semantics with garbage collection, we can apply a systematic abstraction technique such as *Abstracting Abstract Machines* (AAM) [Van Horn and Might 2010] to obtain a stack-imprecise CFA of them. For space reasons, we elide a full formal presentation of the CFA but instead focus on some relevant features.

The AAM abstraction technique largely applies component-wise to the semantic domains of the analysis and leaves the overall structure of the semantic state space intact. The top level in particular is largely preserved and becomes

$$\widehat{State} = \widehat{Eval} + \widehat{Apply}$$
$$\widehat{Eval} = Exp \times \widehat{Env} \times \widehat{Store} \times \widehat{ContAddr} \times \widehat{BindContext}$$
$$\widehat{Apply} = \widehat{Store} \times \widehat{ContAddr} \times \widehat{Val}$$

where each component is replaced by a counterpart abstract component. The two exceptions to this are the expression component *Exp* which remains untouched and the continuation component *Cont* which becomes a continuation address $\widehat{ContAddr}$ identifying a store-allocated continuation. One critical aspect of the abstraction process is the finitization of the store: whereas the domains of concrete stores are unbounded, the domains of abstract stores are explicitly bounded. In consequence, the analysis will often obtain an address at which to map some abstract resource only to find that it is already in use. In this case, the analysis joins the resources, and any access at that address will yield an approximation of each. As we discuss shortly, this phenomenon is strongly intertwined with garbage collection.

The first relevant observation about the abstracted semantics is that the structure of the abstract continuation is embedded completely within the store. Since each $\widehat{Apply}$ contains the entirety of its continuation component, performing abstract garbage collection is largely a matter of porting the machinery from the exact semantics, and is straightforward.

The second relevant observation about the abstracted semantics concerns the abstracted version of the Call rule, defined

$$\textsc{Call}\quad \frac{(\lambda x.e, \hat{\rho}') \in \widehat{aeval}(\hat{\sigma}, \hat{\rho}, ae_0) \qquad \hat{v} = \widehat{aeval}(\hat{\sigma}, \hat{\rho}, ae_1) \qquad \hat{c}' = \lfloor (ae_0\ ae_1) :: \hat{c} \rfloor_m \\ \hat{\sigma}' = \hat{\sigma}[(x, \hat{c}') \mapsto \hat{v}] \qquad \hat{\rho}'' = \hat{\rho}'[x \mapsto \hat{c}']}{\mathsf{ev}((ae_0\ ae_1), \hat{\rho}, \hat{\sigma}, \hat{\alpha}, \hat{c}) \hat{\rightarrow}_{\mathsf{ev}} \mathsf{ev}(e, \hat{\rho}'', \hat{\sigma}', \hat{\alpha}, \hat{c}')}$$

Modulo the metavariable names denoting abstracted elements, this rule is identical to that of the exact semantics except for two aspects: First, the abstract evaluation of the operator ae_0 may yield multiple closures and the analysis considers the application of each.[2] Second, the binding context $\hat{c}'$ is defined as the length-m prefix of its concrete counterpart (obtained via $\lfloor \cdot \rfloor_m$).

Regular application of abstract garbage collection reaps stale bindings from the store. Then, later allocations that would have found an address in use will instead find it free. In effect, garbage collection causes addresses to map to fewer abstract values which we observe in particular when we determine the value of an operator.

3.5 Stack-Precise CFA with Garbage Collection

Stack-imprecise CFAs typically allocate the continaution in the store, subjecting it to the same nondeterminism that afflicts store-allocated values. Where value nondeterminism manifests as evaluation yielding multiple values, continuation nondeterminism manifests as evaluation returning to multiple points. Stack-precise CFAs, on the other hand, do not allocate it in the store but instead factor it out of machine states to manage it globally. (Some CFAs factor the store out of machine states to manage globally, part of *widening the store*. In a sense, factoring out the continuation is part of *widening the continuation*.) Once factored, the state space of machine states is defined

$$\begin{aligned} State &= Eval + Apply \\ Eval &= Exp \times Env \times Store \times BindContext \\ Apply &= Store \times Val \end{aligned}$$

before abstraction. *Eval* states become *evaluation configurations* and *Apply* states become *evaluation results*. Except for the presence of the binding context, these are the same shapes one finds in actual stack-precise CFAs [Darais et al. 2017; Vardoulakis and Shivers 2011; Wei et al. 2018].

However, factoring the continuation out and ceding control of it to the analysis presents an obstacle to abstract garbage collection which needs to extract the root set of reachable addresses from it. Earl et al. [2010] developed a technique by which the analysis could introspect the continuation and offer as a primitive the extraction of the root set of reachable addresses from the continuation. Johnson et al. [2014] refined this "fully-precise" technique and offered an "approximate" technique

[2]Due to the inherent imprecision of analysis, not every abstractly-applied closure will necessarily appear in a corresponding call under the exact semantics. Such closures, initiating spurious control paths, waste analysis effort and this waste compounds as the exploration of spurious paths leads to the discovery of yet more.

that introduced this set of root addresses as a component in the evaluation configuration. Darais et al. [2017] showed that the *Abstracting Definitional Interpreters*-approach is compatible with this latter technique.

These techniques, while effective at admitting abstract garbage collection to stack-precise CFAs, violate context irrelevance (in spirit, if not outright). In terms of the initial semantics of Section 3.2, context irrelevance is the property that $\mathsf{ev}(e, \rho, \sigma, \kappa, c) \rightarrow^{+} \mathsf{ap}(\sigma', \kappa, v)$ if and only if $\mathsf{ev}(e, \rho, \sigma, \kappa', c) \rightarrow^{+} \mathsf{ap}(\sigma', \kappa', v)$ which is to say that the evaluation of a configuration is unaffected by its continuation. A stack-precise CFA of this small-step semantics (but without abstraction garbage collection) exhibits context irrelevance in this same way. Because abstract garbage collection modifies the store non-monotonically, it prevents (spurious) control paths from being explored and changes the store yielded by those that are explored. Thus, the abstract evaluation of a configuration becomes dependent on (the reachability roots embedded in) its continuation. The "approximate" technique that introduces the set of root addresses as a component in the evaluation configuration vacuously restores context irrelevance by distinguishing otherwise-identical configurations based on the continuation. That is, the states $\mathsf{ev}(e, \rho, \sigma, \kappa, c)$ and $\mathsf{ev}(e, \rho, \sigma, \kappa', c)$ with identical configurations but distinct continuations become the continuation-less evaluation configurations $\mathsf{ev}(e, \rho, \sigma, A, c)$ and $\mathsf{ev}(e, \rho, \sigma, A', c)$ with distinct root address sets A and A'. This address set is a close approximation of the continuation and effectively preserves context *relevance* in the semantics.

4 FROM THREADED TO COMPOSITIONAL STORES

In this section, we present a series of four semantics that gradually transition from a threaded treatment of stores without garbage collection to a compositional treatment of stores with garbage collection. We define each of these semantics in terms of big-step judgments of (or close to) the form $\sigma, \rho, c \vdash e \Downarrow (v, \sigma')$. This judgment expresses that the *evaluation configuration* consisting of the expression e under the store σ, environment ρ, and timestamp c evaluates to the *evaluation result* consisting of the value v and the store σ'.

Formulating our semantics in big-step style offers two advantages to our setting: First, we can readily express them by big-step definitional interpreters at which point we can apply systematic abstraction techniques [Darais et al. 2017; Wei et al. 2018] to obtain corresponding CFAs exhibiting perfect stack precision. Second, they emphasize the availability of the configuration store at the delivery point of the evaluation result; this availability is crucial to our ability to shift to a compositional treatment of the store.

4.1 Threaded-Store Semantics

To orient ourselves to the big-step setting, we present the reference semantics for our language in big-step style in Figure 2. This reference semantics is equivalent to the reference semantics given in small-step style in Section 3.2 except that there is no corresponding Apply rule; its responsibility—to deliver a value to a continuation—is handled implicitly by the big-step formulation. In terms of big-step semantics, this reference semantics is characterized by the threading of the store through each rule; the resultant store of evaluation is the configuration store plus the allocation and mutation incurred during evaluation. Hence, we refer to this semantics as the *threaded-store* semantics. We use natural numbers as store subscripts in each rule to emphasize the store's monotonic increase.

A program *pr* is evaluated in an initial configuration with an empty store $\bot$, an empty environment $\bot$, and an empty binding context $\langle\rangle$. In such a configuration, *pr* evaluates to a value v if $\bot, \bot, \langle\rangle \vdash pr \Downarrow (v, \sigma)$.

The Let rule evaluates the bound call expression *ce* under the incoming environment and store. If evaluation results in a value–store pair, this incoming environment is extended with a binding

$$\textsc{Let}\quad \frac{\sigma_0, \rho, c \vdash ce \Downarrow (v_0, \sigma_1) \qquad \rho' = \rho[x \mapsto c] \qquad \sigma_2 = \sigma_1[(x, c) \mapsto v_0] \qquad \sigma_2, \rho', c \vdash e \Downarrow (v, \sigma_3)}{\sigma_0, \rho, c \vdash \mathsf{let}\, x = ce\, \mathsf{in}\, e \Downarrow (v, \sigma_3)}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}((\lambda x.e, \rho_0), \sigma_1) = \mathsf{aeval}(\sigma_0, \rho, ae_0) \qquad (v_1, \sigma_2) = \mathsf{aeval}(\sigma_1, \rho, ae_1) \qquad c' = (ae_0\ ae_1) :: c \\ \rho_1 = \rho_0[x \mapsto c'] \qquad \sigma_3 = \sigma_2[(x, c') \mapsto v_1] \qquad \sigma_3, \rho_1, c' \vdash e \Downarrow (v, \sigma_4)\end{array}}{\sigma_0, \rho, c \vdash (ae_0\ ae_1) \Downarrow (v, \sigma_4)}$$

$$\textsc{Set!}\quad \frac{(v, \sigma_1) = \mathsf{aeval}(\sigma_0, \rho, ae) \qquad \sigma_1 = \sigma_0[(x, \rho(x)) \mapsto v]}{\sigma_0, \rho, c \vdash \mathsf{set!}\, x\ ae \Downarrow ((\lambda \mathsf{x}.\mathsf{x}, \bot), \sigma_1)} \qquad \textsc{Atomic}\quad \frac{}{\sigma, \rho, c \vdash ae \Downarrow \mathsf{aeval}(\sigma, \rho, ae)}$$

Fig. 2. The threaded-store semantics

derived from the bound variable and incoming binding context.[3] The resultant store is extended with mapping from that binding to the resultant value. The body expression is evaluated under the extended environment and store and its result becomes that of the overall expression.

Contrasting the treatment of the environment and the store by the Let rule is instructive. On the one hand, the environment is treated compositionally: the incoming environment of evaluation is restored and extended after evaluation of the bound value. On the other hand, the store is treated non-compositionally: the store resulting from the evaluation of the bound expression is extended after it has accumulated the effects of its evaluation. By this reasoning, we classify the treatment of the binding context as *compositional* rather than *threaded*; this treatment is a departure from typical practice of CFA.

The Call rule evaluates the atomic expressions ae_0 and ae_1 for the operator and argument, respectively. It then derives a new binding context, extends the environment and store with a binding using that context, and evaluates the operator body under the extended environment, store, and derived binding context. The result of evaluation the body is that of the overall expression.

The Set! rule evaluates the atomic body expression ae and updates the binding of the referenced variable in the store. Its result is the identity function paired with the updated store.

The Atomic rule evaluates an atomic expression ae using the aeval atomic evaluation metafunction. Foreshadowing the succeeding semantics, we define aeval to return a pair of its calculated value and the given store. In this semantics, the store is passed through unmodified; in forthcoming semantics, it will be altered according to the calculated value.

$$\mathsf{aeval}(\sigma, \rho, x) = (\sigma(x, \rho(x)), \sigma) \qquad\qquad \mathsf{aeval}(\sigma, \rho, \lambda x.e) = ((\lambda x.e, \rho|_{\lambda x.e}), \sigma)$$

4.2 Threaded-Store Semantics with Effect Log

The second semantics enhances the reference semantics with an *effect log* ξ which explicitly records the mutation that occurs through evaluation. The effect log is considered part of the evaluation result; accordingly the *effect log semantics* are in terms of judgments of the form $\sigma, \rho, c \vdash e \Downarrow_{!} (v, \sigma'), \xi$. Figure 3 presents the effect log semantics, identical to the reference semantics except for (1) the addition of the effect log and (2) the use of the metavariable a to denote an address (x,c). (This usage continues in all subsequent semantics as well.)

[3]Because the program is alphatised, the binding of a let-bound variable in a particular calling context will not interfere with the binding of any other variable.

$$\textsc{Let}\quad \frac{\sigma_0, \rho, c \vdash ce \Downarrow_! (v_0, \sigma_1), \xi_0 \qquad \rho' = \rho[x \mapsto c] \qquad \sigma_2 = \sigma_1[(x, c) \mapsto v_0] \qquad \sigma_2, \rho', c \vdash e \Downarrow_! (v, \sigma_3), \xi_1}{\sigma_0, \rho, c \vdash \mathsf{let}\, x = ce\, \mathsf{in}\, e \Downarrow_! (v, \sigma_3), \xi_1 \circ \mathsf{extend}_{log}((x, c), v_0, \sigma_1) \circ \xi_0}$$

$$\textsc{Call}\quad \frac{((\lambda x.e, \rho_0), \sigma_1) = \mathsf{aeval}(\sigma_0, \rho, ae_0) \qquad (v_1, \sigma_2) = \mathsf{aeval}(\sigma_1, \rho, ae_1) \qquad c' = (ae_0\, ae_1) :: c \qquad \rho_1 = \rho_0[x \mapsto c'] \qquad \sigma_3 = \sigma_2[(x, c') \mapsto v_1] \qquad \sigma_3, \rho_1, c' \vdash e \Downarrow_! (v, \sigma_4), \xi}{\sigma_0, \rho, c \vdash (ae_0\, ae_1) \Downarrow_! (v, \sigma_4), \xi \circ \mathsf{extend}_{log}((x, c'), v_1, \sigma_2)}$$

$$\textsc{Set!}\quad \frac{(v, \sigma_1) = \mathsf{aeval}(\sigma_0, \rho, ae) \qquad a = (x, \rho(x)) \qquad \sigma_1 = \sigma_0[a \mapsto v]}{\sigma_0, \rho, c \vdash \mathsf{set!}\, x\, ae \Downarrow_! ((\lambda \mathsf{x}.\mathsf{x}, \bot), \sigma_1), \mathsf{extend}_{log}(a, v, \sigma_1)} \qquad \textsc{Atomic}\quad \frac{}{\sigma, \rho, c \vdash ae \Downarrow_! \mathsf{aeval}(\sigma, \rho, ae), \lambda\sigma.\sigma}$$

Fig. 3. Threaded-store semantics with an effect log

The effect log is represented by a function from store to store. The definition of each log is given by either a literal identity function, a use of the extend_{log} metafunction, or the composition of effect logs. The extend_{log} metafunction is defined

$$\mathsf{extend}_{log}(a, v, \sigma') = \lambda\sigma.\sigma[a \mapsto v] \cup \sigma'$$

where the union of the extended store $\sigma[a \mapsto v]$ and the value-associated store σ' treats each store as a relation but the result is always a function (in the sense that no binding is related to more than one value). The effect log of the Atomic rule is the identity function, reflecting that no allocation or mutation is performed when evaluating an atomic expression. The effect log of the Set! rule is constructed by the metafunction extend_{log}; the store argument to extend_{log} is the store *after* the mutation has occurred. The use of this store is necessary to propagate the mutative effect and ensures that its union with the store on which this log is replayed agrees on all common bindings. The effect log of the Call rule is composed of the effect log of evaluation of the body and an entry for the allocation of the bound variable. Finally, the effect log of the Let rule is composed of the effect logs of evaluation of both the body and binding expression interposed by an entry for the allocation of the bound variable.

The intended role of the effect log is captured by the following lemma, which states that one may obtain the resultant store by applying the resultant log to the initial store of evaluation.

Lemma 4.1. *If* $\sigma, \rho, c \vdash e \Downarrow_! (v, \sigma'), \xi$, *then* $\sigma' = \xi(\sigma)$.

The proof proceeds straightforwardly by induction on the judgment's derivation.

4.3 Compositional-Store Semantics

The third semantics (seen in Figure 4) shifts the previous semantics from threading the store to treating it compositionally. Under this treatment, evaluation results still consist of a value, store, and effect log, but the store is associated directly to the value—at least conceptually—and not treated as a global effect repository. This alternative role is particularly apparent in the Let rule: the store resulting from evaluation of the bound expression is not extended to be used as the initial store of evaluation of the body. instead, the effect log resulting from evaluation of the bound expression is applied to the initial store (of the overall let expression). We emphasize this compositional treatment

$$\textsc{Let}\quad \frac{\sigma,\rho,c \vdash ce \Downarrow_\circ (v',\sigma_{v'}),\xi' \qquad \sigma' = \xi'(\sigma) \qquad (\rho',\sigma'') = \mathsf{extend}(\rho,\sigma',x,c,v',\sigma_{v'}) \qquad \sigma'',\rho',c \vdash e \Downarrow_\circ (v,\sigma_v),\xi}{\sigma,\rho,c \vdash \mathsf{let}\,x = ce\,\mathsf{in}\,e \Downarrow_\circ (v,\sigma_v),\xi \circ \mathsf{extend}_{log}((x,c),v',\sigma_{v'}) \circ \xi'}$$

$$\textsc{Call}\quad \frac{((\lambda x.e,\rho_0),\sigma_0) = \mathsf{aeval}(\sigma,\rho,ae_0) \qquad (v_1,\sigma_1) = \mathsf{aeval}(\sigma,\rho,ae_1) \qquad c' = (ae_0\ ae_1) :: c \qquad (\rho',\sigma') = \mathsf{extend}(\rho_0,\sigma_0,x,c',v_1,\sigma_1) \qquad \sigma',\rho',c' \vdash e \Downarrow_\circ (v,\sigma_v),\xi}{\sigma,\rho,c \vdash (ae_0\ ae_1) \Downarrow_\circ (v,\sigma_v),\xi \circ \mathsf{extend}_{log}((x,c'),v_1,\sigma_1)}$$

$$\textsc{Set!}\quad \frac{(v,\sigma_v) = \mathsf{aeval}(\sigma,\rho,ae) \qquad a = (x,\rho(x)) \qquad \sigma' = \sigma_v[a \mapsto v]}{\sigma,\rho,c \vdash \mathsf{set!}\,x\ ae \Downarrow_\circ ((\lambda \mathsf{x}.\mathsf{x},\bot),\sigma'),\mathsf{extend}_{log}(a,v,\sigma')} \qquad \textsc{Atomic}\quad \frac{}{\sigma,\rho,c \vdash ae \Downarrow_\circ \mathsf{aeval}(\sigma,\rho,ae),\lambda\sigma.\sigma}$$

Fig. 4. The compositional-store semantics

by no longer using numeric subscripts, suggesting an "evolution" of the store, and instead using ticks, suggesting distinct (but related) instances.

We ensure that the store associated with each value *closes* that value. A store σ closes a value v if σ contains an entry for each address (variable–binding pair) transitively reachable from v. The reachability of addresses is straightforward:

- An address (x,c) is reachable from ρ if x is in the domain of ρ and $\rho(x) = c$.
- An address a is reachable from a value $(\lambda x.e,\rho)$ if it is reachable from ρ.
- An address a is reachable from an address a' with respect to a store σ if a is reachable from $\sigma(a')$.

We use the extend metafunction to bind a value to a variable in a given binding context within a given environment and store, defined

$$\mathsf{extend}(\rho,\sigma,x,c,v,\sigma_v) = (\rho[x \mapsto c], \sigma[(x,c) \mapsto v] \cup \sigma_v)$$

When we extend a store with a mapping for a value, we also copy all of the mappings from that value's associated store. In this semantics, the newly-extended store will always already have them. In the next semantics, in which we introduce garbage collection, it won't necessarily. In both this semantics and the next, however, the newly-extended store and the value's store will agree on any common bindings.

Although the role of the store has changed, the same lemma holds in this semantics as does in the previous. We repeat it in terms of this semantics.

Lemma 4.2. *If* $\sigma,\rho,c \vdash e \Downarrow_\circ (v,\sigma_v),\xi$, *then* $\xi(\sigma) = \sigma_v$.

Like the previous lemma, its proof can be obtained by induction on the judgment's derivation.

4.4 Compositional-Store Semantics with Garbage Collection

Our final semantics (seen in Figure 5) continues the compositional treatment of the store but adds garbage collection to remove bindings unreachable from a root set. Because the role of each store is to close a value (i.e. house values for each binding transitively reachable from its environment), this root set consists solely of the bindings in the value's environment; in particular, the environments residing in the stack are irrelevant to reachability.

$$\textsc{Let}\quad \frac{\begin{array}{c}(\rho_{ce}, \sigma_{ce}) = \mathsf{restrict}(ce, \rho, \sigma) \qquad \sigma_{ce}, \rho_{ce}, c \vdash ce \Downarrow_{gc} (v', \sigma_{v'}), \xi' \qquad \sigma' = \xi'(\sigma) \\ (\rho', \sigma'') = \mathsf{extend}(\rho, \sigma', x, c, v', \sigma_{v'}) \qquad (\rho_e, \sigma_e) = \mathsf{restrict}(e, \rho', \sigma'') \qquad \sigma_e, \rho_e, c \vdash e \Downarrow_{gc} (v, \sigma_v), \xi\end{array}}{\sigma, \rho, c \vdash \mathsf{let}\, x = ce\, \mathsf{in}\, e \Downarrow_{gc} (v, \sigma_v), \xi \circ \mathsf{extend}_{log}((x, c), v', \sigma_{v'}) \circ \xi'}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}((\lambda x.e, \rho_0), \sigma_0) = \mathsf{aeval}_{gc}(\sigma, \rho, ae_0) \qquad (v_1, \sigma_1) = \mathsf{aeval}_{gc}(\sigma, \rho, ae_1) \qquad c' = (ae_0\ ae_1) :: c \\ (\rho', \sigma') = \mathsf{extend}(\rho_0, \sigma_0, x, c', v_1, \sigma_1) \qquad (\rho_e, \sigma_e) = \mathsf{restrict}(e, \rho', \sigma') \qquad \sigma_e, \rho_e, c' \vdash e \Downarrow_{gc} (v, \sigma_v), \xi\end{array}}{\sigma, \rho, c \vdash (ae_0\ ae_1) \Downarrow_{gc} (v, \sigma_v), \xi \circ \mathsf{extend}_{log}((x, c'), v_1, \sigma_1)}$$

$$\textsc{Set!}\quad \frac{\begin{array}{c}(v, \sigma_v) = \mathsf{aeval}_{gc}(\sigma, \rho, ae) \\ a = (x, \rho(x)) \qquad \sigma' = \sigma_v[a \mapsto v]\end{array}}{\sigma, \rho, c \vdash \mathsf{set!}\, x\ ae \Downarrow_{gc} ((\lambda \mathrm{x}.\mathrm{x}, \bot), \bot), \mathsf{extend}_{log}(a, v, \sigma')} \qquad \textsc{Atomic}\quad \frac{}{\sigma, \rho, c \vdash ae \Downarrow_{gc} \mathsf{aeval}_{gc}(\sigma, \rho, ae), \lambda\sigma.\sigma}$$

Fig. 5. The compositional-store semantics with garbage collection

This semantics uses a particular atomic evaluation function aeval_{gc} which garbage collects the store associated with a value. It is defined

$$\mathsf{aeval}_{gc}(\sigma, \rho, x) = (v, \mathrm{gc}(v, \sigma)) \text{ where } v = \sigma(x, \rho(x))$$
$$\mathsf{aeval}_{gc}(\sigma, \rho, \lambda x.e) = (v, \mathrm{gc}(v, \sigma)) \text{ where } v = (\lambda x.e, \rho|_{\lambda x.e})$$

where $\mathrm{gc}(v, \sigma)$ prunes the unreachable bindings from σ with respect to v. Since the reachability calculation and projection is unchanged from the garbage-collection operation of Section 3.3, we elide its definition.

This semantics is careful to ensure that each evaluation is performed under a store with no provably unreachable bindings residing within by frequent use of the restrict metafunction. For a given expression e, closing environment ρ, and closing store σ, the restrict metafunction first determines the restriction of ρ to the free variables of e and then the bindings of σ reachable from that restriction; it then garbage-collects the store by pruning unreachable bindings. Formally, restrict is defined

$$\mathsf{restrict}(e, \rho, \sigma) = (\rho', \mathrm{gc}(\rho', \sigma)) \text{ where } \rho' = \rho|_e$$

where $\mathrm{gc}(\rho, \sigma)$ prunes the unreachable bindings from σ with respect to ρ.

The Let rule proceeds by first obtaining the restriction of the environment and store with respect to the bound expression ce, before evaluating ce under that restriction. The evaluation of ce produces a value v', an associated store $\sigma_{v'}$ which closes only that value, and an effect log ξ'. The Let rule then replays the effect log ξ' on the initial store σ thereby accumulating any mutation (and allocation it depends on) which occurred. After replaying the log, it extends the resultant store σ' and initial environment ρ with a binding for v' and copies the bindings of its associated store $\sigma_{v'}$. Finally, the extended environment and store are restricted with respect to the body expression e before e's evaluation under them.

The Call rule proceeds by atomically evaluating the operator and argument expressions. After calculating the new binding context c', the operator value environment and store are extended with the new binding. Before evaluation of the body e commences, the extended environment and store are restricted with respect to it.

The Set! rule atomically evaluates the expression *ae* producing the assigned value. It returns the identity function which, with an empty environment, is closed by an empty store.

The Atomic rule evaluates an atomic expression with aeval_{gc}.

To connect this semantics to the previous, we show that the addition of garbage collection has no semantic effect by the following lemma.

Lemma 4.3. *If* $\sigma, \rho, c \vdash e \Downarrow_{\circ} (v, \sigma_v), \xi$ *and* $\sigma' = \mathrm{gc}(\rho|_e, \sigma)$ *then* $\sigma', \rho, c \vdash e \Downarrow_{gc} (v, \sigma'_v), \xi'$ *where* $\sigma'_v = \mathrm{gc}(v, \sigma_v)$.

In prose, this lemma states that two evaluation configurations, identical except that one's store is the other's with unreachable bindings pruned, will yield the same evaluation result: their evaluation will produce the same value and, modulo unreachable bindings, the same closing store.

5 ABSTRACT COMPOSITIONAL-STORE SEMANTICS WITH GARBAGE COLLECTION

We now *abstract* the compositional-store semantics with garbage collection—the final semantics of the preceding section. Abstracting the semantics involves (1) defining a finite counterpart of each component of the evaluation configuration and result and (2) defining a counterpart of each semantic rule in terms of these finite components. With each component of the configuration finite, configurations themselves become finite. Then we show that each abstracted rule *simulates* its counterpart—that it admits the full range of its counterpart's behavior. Doing this for each rule ensures that the abstract semantics includes every behavior included by the exact semantics. Once that's complete, we can directly implement our big-step semantics in an abstract definitional interpreter [Darais et al. 2017; Wei et al. 2018] to obtain a control-flow analysis.

We begin by abstracting each configuration component.

$$\begin{aligned}
\widehat{Store} \ni \hat{\sigma} &= \widehat{Address} \to \widehat{Val} & \widehat{Val} \ni \hat{v} &= \mathcal{P}(\widehat{Clo})\\
\widehat{Address} \ni \hat{a} &= Var \times \widehat{BindContext} & \widehat{Clo} &= Lam \times \widehat{Env}\\
\widehat{Log} \ni \hat{\xi} &= \widehat{Address} \to \widehat{Val} & \widehat{BindContext} \ni \hat{c} &= App^{\leq m}\\
\widehat{Env} \ni \hat{\rho} &= Var \rightharpoonup \widehat{BindContext}
\end{aligned}$$

Like its concrete counterpart, an abstract store $\hat{\sigma}$ maps an abstract address to an abstract value. Abstract addresses remain a pair of a variable and binding context, only the context is abstract. An abstract value $\hat{v}$, however, is a *set* of abstract closures rather than a single closure. An abstract closure is a λ paired with an abstract environment $\hat{\rho}$ which itself is a finite map from variables to binding contexts. An abstract binding context $\hat{c}$ is a sequence of at most m application sites, where m is a parameter to the analysis.[4] We represent an abstract log $\hat{\xi}$ with the same structure as an abstract store but interpret it as the mappings we must add to the initial store—a kind of *store delta*. We define abstract join, composition, and application operators by

$$\hat{\sigma}_0 \sqcup \hat{\sigma}_1 = \lambda\hat{a}.\hat{\sigma}_0(\hat{a}) \cup \hat{\sigma}_1(\hat{a}) \qquad \hat{\xi}_0 \hat{\circ} \hat{\xi}_1 = \hat{\xi}_0 \sqcup \hat{\xi}_1 \qquad \hat{\xi}(\hat{\sigma}) = \hat{\sigma} \sqcup \hat{\xi}$$

Formally connecting these abstract domains to their concrete counterparts is critical to establishing the simulation relationship between the exact and abstract semantics. We make this connection by means of a polymorphic abstraction function $|\cdot|$[5], defined for all domains except stores by

$$|\rho| = \lambda x.|\rho(x)| \qquad |c| = \lfloor c \rfloor_m \qquad |(\lambda x.e, \rho)| = \{(\lambda x.e, |\rho|)\} \qquad |\xi| = |\xi(\bot)|$$

[4]The parameter m is used similarly to the parameter k of k-CFA.

[5]The abstraction function is typically accompanied by a complementary *concretization* function to complete a Galois connection. For simplicity here, we leave it incomplete.

Let

$$\frac{\begin{array}{c}(\hat{\rho}_{ce}, \hat{\sigma}_{ce}) = \widehat{\mathsf{restrict}}(ce, \hat{\rho}, \hat{\sigma}) \qquad \hat{\sigma}_{ce}, \hat{\rho}_{ce}, \hat{c} \vdash ce \mathrel{\hat{\Downarrow}} (\hat{v}', \hat{\sigma}_{v'}), \hat{\xi}' \qquad \hat{\sigma}' = \hat{\xi}'(\hat{\sigma}) \\ (\hat{\rho}', \hat{\sigma}'') = \widehat{\mathsf{extend}}(\hat{\rho}, \hat{\sigma}', x, \hat{c}, \hat{v}', \hat{\sigma}'_v) \qquad (\hat{\rho}_e, \hat{\sigma}_e) = \widehat{\mathsf{restrict}}(e, \hat{\rho}', \hat{\sigma}'') \qquad \hat{\sigma}_e, \hat{\rho}_e, \hat{c} \vdash e \mathrel{\hat{\Downarrow}} (\hat{v}, \hat{\sigma}_v), \hat{\xi}\end{array}}{\hat{\sigma}, \hat{\rho}, \hat{c} \vdash \mathsf{let}\, x = ce \,\mathsf{in}\, e \mathrel{\hat{\Downarrow}} (\hat{v}, \hat{\sigma}_v), \hat{\xi} \mathbin{\hat{\circ}} \hat{\xi}'}$$

Call

$$\frac{\begin{array}{c}(\hat{v}_0, \hat{\sigma}_0) = \widehat{\mathsf{aeval}}(\hat{\sigma}, \hat{\rho}, ae_0) \qquad (\lambda x.e, \hat{\rho}_0) \in \hat{v}_0 \\ (\hat{v}_1, \hat{\sigma}_1) = \widehat{\mathsf{aeval}}(\hat{\sigma}, \hat{\rho}, ae_1) \qquad \hat{c}' = \lfloor (ae_0\ ae_1) :: \hat{c} \rfloor_m \\ (\hat{\rho}', \hat{\sigma}') = \widehat{\mathsf{extend}}(\hat{\rho}_0, \hat{\sigma}_0, x, \hat{c}', \hat{v}_1, \hat{\sigma}_1) \qquad (\hat{\rho}_e, \hat{\sigma}_e) = \widehat{\mathsf{restrict}}(e, \hat{\rho}', \hat{\sigma}') \qquad \hat{\sigma}_e, \hat{\rho}_e, \hat{c}' \vdash e \mathrel{\hat{\Downarrow}} (\hat{v}, \hat{\sigma}_v), \hat{\xi}\end{array}}{\hat{\sigma}, \hat{\rho}, \hat{c} \vdash (ae_0\ ae_1) \mathrel{\hat{\Downarrow}} (\hat{v}, \hat{\sigma}_v), \hat{\xi}}$$

Set!

$$\frac{\begin{array}{c}(\hat{v}, \hat{\sigma}_v) = \widehat{\mathsf{aeval}}(\hat{\sigma}, \hat{\rho}, ae) \\ (\_, \hat{\xi}) = \widehat{\mathsf{extend}}(\bot, \bot, x, \hat{\rho}(x), \hat{v}, \hat{\sigma}_v)\end{array}}{\hat{\sigma}, \hat{\rho}, \hat{c} \vdash \mathsf{set!}\, x\ ae \mathrel{\hat{\Downarrow}} (\{(\lambda \mathsf{x}.\mathsf{x}, \bot)\}, \bot), \hat{\xi}}$$

Atomic

$$\frac{}{\hat{\sigma}, \hat{\rho}, \hat{c} \vdash ae \mathrel{\hat{\Downarrow}} \widehat{\mathsf{aeval}}(\hat{\sigma}, \hat{\rho}, ae), \bot}$$

Fig. 6. The abstract compositional-store semantics with garbage collection

and for stores by

$$|\sigma| = \lambda \hat{a}. \bigcup_{|a| = \hat{a}} |\sigma(a)|$$

Abstracting a store groups entries by their abstracted address in a large set. Abstracting an environment ρ abstracts its range. Abstracting a binding context c takes its at-most-m-length prefix. Abstracting a closure produces a singleton of that closure with an abstracted environment. Finally, abstracting a log ξ produces the abstract store that results from apply the log to the empty store $\bot$ and then abstracting.

Figure 6 defines the abstract compositional-store semantics with garbage collection. Structurally, nearly every rule is identical to the exact counterpart that it abstracts; most of the work of abstraction is defining the abstract domains and metafunctions and connecting them to those of the exact semantics. The Call rule differs structurally from its exact counterpart in two notable ways: First, because an abstract value is a set of closures, it applies for each such closure in the operator set. Second, it defines the new binding context $\hat{c}'$ to be the prefix of the application site prepended to the previous abstract time $\hat{c}$ and limited to a length of at most m. The abstract $\widehat{\mathsf{aeval}}$ metafunction is defined

$$\widehat{\mathsf{aeval}}(\hat{\sigma}, \hat{\rho}, x) = (\hat{v}, \widehat{\mathsf{gc}}(\hat{v}, \hat{\sigma})) \text{ where } \hat{v} = \hat{\sigma}(\hat{\rho}(x))$$

$$\widehat{\mathsf{aeval}}(\hat{\sigma}, \hat{\rho}, \lambda x.e) = (\hat{v}, \widehat{\mathsf{gc}}(\hat{v}, \hat{\sigma})) \text{ where } \hat{v} = \{(\lambda x.e, \hat{\rho}|_{\lambda x.e})\}$$

For space, we omit straightforward definitions for the abstract variants of $\widehat{\mathsf{gc}}$, $\widehat{\mathsf{restrict}}$, and $\widehat{\mathsf{extend}}$.

As a final step before we establish the simulation relationship, we define an ordering on stores (and logs, extending it in the natural way):

$$\hat{\sigma}_0 \sqsubseteq \hat{\sigma}_1 \Leftrightarrow \forall \hat{a} \in \widehat{Address}. \hat{\sigma}_0(\hat{a}) \subseteq \hat{\sigma}_1(\hat{a}) \qquad \hat{v}_0 \sqsubseteq \hat{v}_1 \Leftrightarrow \hat{v}_0 \subseteq \hat{v}_1$$

Once again, these semantics maintain the invariant that the resultant store σ' contains only data reachable from v. We establish the invariant by an initial derivation $\bot, \bot, \langle\rangle \vdash pr \hat{\Downarrow} (\hat{v}, \hat{\sigma}), \hat{\xi}$ which obtains when a program pr evaluates to a value v closed by σ.

We formally connect this semantics with the previous by the following abstraction theorem.

Theorem 5.1. *If* $|\sigma| \sqsubseteq \hat{\sigma}$ *and* $|\rho| = \hat{\rho}$ *and* $|c| = \hat{c}$ *and* $\sigma, \rho, c \vdash e \Downarrow_{gc} (v, \sigma_v), \xi$, *then* $\hat{\sigma}, \hat{\rho}, \hat{c} \vdash e \hat{\Downarrow} (\hat{v}, \hat{\sigma}_v), \hat{\xi}$ *where* $|v| \sqsubseteq \hat{v}$ *and* $|\sigma_v| \sqsubseteq \hat{\sigma}_v$ *and* $|\xi| \sqsubseteq \hat{\xi}$.

This theorem states that if the configuration components are related by abstraction, then, for any given derivation in the exact semantics, there is an derivation in the abstract semantics which yields an abstraction of its results. It can be proved by induction on the derivation.

6 RELATED WORK

This work has its roots in abstract interpretation [Cousot and Cousot 1977] and is an instance of control-flow analysis [Shivers 1991].

There have been various efforts to obtain perfect stack precision in control-flow analysis. The first realizations were CFA2 [Vardoulakis and Shivers 2011] and PDCFA [Earl et al. 2010]. Successive realizations [Gilray et al. 2016; Johnson and Van Horn 2014] sought to decrease both its time complexity and implementation burden. Current realizations [Darais et al. 2017; Wei et al. 2018] are formulated as abstract *definitional interpreters* [Reynolds 1998]. These realizations are particularly apt as they provide the most straightforward implementation technique for the big-step abstract semantics we introduced.

Might and Shivers [2006] introduced abstract garbage collection to Shivers-style control-flow analysis. Earl et al. [2012] reconcile the technique with pushdown systems using an approach whereby each control state includes a set of reachability roots as a component. Johnson et al. [2014] refine the technique, calling it "fully-precise", and introduce an "approximate" variant in which the analyzer traverses the abstract stack to gather reachability roots. Darais et al. [2017] show that abstract definitional interpreters easily accommodate abstract garbage collection; essentially they use the fully-precise technique within a state monad. Each of these three combinations of abstract garbage collection and pushdown systems sacrifices context irrelevance; the primary purpose of this work is to combine the two while preserving it.

7 CONCLUSION

In this paper, we exploited the precise call—return matching of stack-precise CFA to treat the store compositionally in evaluation rather than threading it through each step. We recovered the ability to propagate effects through evaluation (e.g. mutation) by associating an effect log with each computation. Because the effect log behaves compositionally, we end up with an association between configurations and the values they yield that is completely independent of the control context of those configurations, thereby restoring the context irrelevance enjoyed by non-GC stack-precise CFA without sacrificing GC itself.

We observed that this compositional treatment can be generalized somewhat: when we treat the binding context (used in the allocation of abstract resources) compositionally rather than threading it through evaluation, the analysis exhibits the m-CFA context abstraction instead of that of k-CFA.

The complement to abstract garbage collection is abstract counting [Might and Shivers 2006] which keeps track of the number of concrete resources that correspond to an abstract resource and enables certain abstract transitions, such as a strong store update. It would be interesting to apply abstract counting to compositional stores to perform strong updates or effect logs to record when a strong update of a binding is possible. This application might connect to the *stack environments* of CFA2 [Vardoulakis and Shivers 2011].

REFERENCES

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 238–252.

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 12 (Aug. 2017), 25 pages. https://doi.org/10.1145/3110256

Christopher Earl, Matthew Might, and David Van Horn. 2010. Pushdown Control-Flow Analysis of Higher Order Programs. (2010).

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2364527.2364576

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. 2014. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming* 24 (May 2014), 218–283. https://doi.org/10.1017/s0956796814000100

James Ian Johnson and David Van Horn. 2014. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 11–22. https://doi.org/10.1145/2661088.2661098

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/1159803.1159807

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397.

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

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science* 7, 2 (2011), 39. https://doi.org/10.2168/LMCS-7(2:3)2011

Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 105 (July 2018), 28 pages. https://doi.org/10.1145/3236800