Delta Store Semantics: Abstract Garbage Collection for Abstract Definitional Interpreters

Noah Van Es ✉, Bram Vandenbogaerde, and Coen De Roover

Vrije Universiteit Brussel, Belgium
{noah.van.es, bram.vandenbogaerde, coen.de.roover}@vub.be

Abstract. Both the precision and performance of abstract interpreters can be improved greatly through the integration of abstract garbage collection (GC). Unfortunately, for abstract interpreters that do not explicitly model the stack (e.g., abstract definitional interpreters), this integration has proven cumbersome. Existing approaches either fail to exploit the full precision and performance benefits of abstract GC and pushdown control flow, and/or require complicated modifications to the abstract interpreter. In addition, the lack of global store widening, which is incompatible with abstract GC, often remains an obstacle for scalability. In this work, we present delta store semantics (DSS), offering a novel yet simple approach to integrate abstract GC into big-step abstract definitional interpreters. DSS makes a simple change to the standard big-step language semantics, returning a delta store (representing changes to the original store) instead of an updated store, enabling the integration of a single evaluation rule to interleave GC into its semantics. Importantly, we show that DSS not only preserves the advantages of big-step abstract interpreters and abstract GC, but in fact can exploit greater precision benefits (due to more aggressive GC). We formulate this claim as a theorem, for which we provide both a mechanised proof in Rocq, as well as empirical evidence. Finally, we propose a new form of store widening for DSS, which tackles the scalability issues of abstract interpreters employing abstract GC without store widening. The result is similar to the traditional notion of flow sensitivity in data-flow analyses.

Keywords: Abstract Interpretation · Abstract Garbage Collection

SAS Artifact ★ Validated

SAS Artifact ★ Extensible

SAS Artifact ★ Available

1 Introduction

Abstract garbage collection is the application of garbage collection (GC) to an abstract interpreter. Similar to concrete GC, it reclaims memory locations that are no longer reachable. Its purpose, however, is entirely different: for a concrete interpreter, GC simply frees up memory resources, whereas in an abstract interpreter, its main purpose is to improve precision and scalability [21,22].

Abstract GC has frequently been integrated into abstract interpreters that employ *small-step* operational semantics, such as those obtained by following the *Abstracting Abstract Machines* (AAM) approach to abstract interpretation [30,21]. In contrast, for abstract interpreters that employ *big-step* definitional interpreters [28], this integration is inherently more challenging. The reason for this challenge is that for such *abstract definitional interpreters* [2], there is no explicit (abstraction of the) continuation. A direct application of GC requires access to the continuation to ensure that all addresses that are reachable therefrom are not accidentally reclaimed, as they might still be needed later on. That is, addresses reachable from the continuation are traditionally part of the GC root set.

Yet, at the same time, the lack of an explicit continuation on its own brings about key benefits to the precision and scalability of the abstract interpreter. These benefits are widely recognized under the umbrella of *pushdown control flow* models [3,11]. It should come as no surprise that various attempts [4,2,8] have been made to combine the benefits of such models with the benefits of abstract GC, aiming to realize “the best of both worlds”. Unfortunately, these attempts often fall short of that promise, sacrificing some benefits of both abstract GC and pushdown control flow to enable their co-existence. Another issue is that these existing approaches overcomplicate the underlying machinery of the abstract interpreter, requiring significant engineering efforts and rendering formal reasoning about the interpreter’s abstract semantics more challenging.

1.1 Motivation: The Best of Both Worlds

Both abstract GC and the pushdown control flow of abstract definitional interpreters offer significant precision and performance improvements.

Specifically, for abstract GC, we identify the following advantages:

Avoiding Garbage-Induced Imprecision. In order to ensure termination, an abstract interpreter needs to use a finite number of addresses. As such, multiple allocations may need to reuse the same address. All values allocated at the same address are *joined* together, causing a loss in precision. Abstract GC can prevent some of those precision-detrimental joins: by freeing up addresses that are unreachable, it prevents future allocations at those addresses from having to join the newly allocated value with the old “garbage values”. When using abstract counting [16,21], which counts the number of concrete bindings for each abstract address, abstract GC can also prevent counting dead bindings.

Without abstract GC, these sources of imprecision in the heap are propagated, bringing garbage “back to life” and disturbing not only the data but also the control flow of the interpreter due to *spurious paths* [21]. Existing work has therefore shown that abstract GC can lead to order-of-magnitude improvements in precision (and performance, as the exploration of spurious paths is avoided).

Garbage Irrelevance. Another advantage of abstract GC relates not directly to the interpreter's precision, but to the scalability of the underlying fixpoint algorithm. Specifically, by removing garbage from the heap, the states explored by the abstract interpreter become smaller (i.e., the abstract interpreter only needs to explore the subset of the state space where all garbage is removed). This causes the interpreter to more frequently end up in the same states, allowing the reuse of previously computed analysis results for such states. In contrast, without abstract GC, the interpreter often has to analyze multiple states that are equivalent, but not equal, as they differ only in unreachable (garbage) bindings.

We refer to the examples given in [21,4,29] to illustrate the impact of these benefits. Only an interpreter that is entirely *garbage-free* [21,29] (i.e., eliminates all garbage immediately) enjoys the full benefit of both these advantages.

Similarly, we enumerate the advantages that are inherent to the pushdown control flow of abstract definitional interpreters.

Perfect Stack Precision. The big-step semantics of abstract definitional interpreters no longer model the continuation (i.e., "the stack"), and instead rely on the underlying continuation of the host language (or formalism) for recursive evaluation rules. Since the continuation is no longer modelled in the semantics, an abstraction of these semantics no longer requires an abstraction (i.e., approximation) of the continuation either. Such an abstraction of the continuation would otherwise cause imprecision in control flow, incorrectly matching callers and returns. In contrast, the pushdown control flow of abstract definitional interpreters entirely avoids this precision loss, resulting in "perfect stack precision" [2].

Context Irrelevance. Germane et al. [8] point out that big-step semantics, such as those of abstract definitional interpreters, also enjoy the benefit of context irrelevance. Analogous to garbage irrelevance, context irrelevance ensures that states are not differentiated based on their continuation component (i.e., their "context"). Therefore, when the same evaluation state is encountered in multiple contexts, the analysis results for that state can be reused in every context, improving scalability. The rationale is that just as garbage is not relevant to the evaluation of a current state, in many languages the continuation component also does not contribute to the result of an evaluation. Of course, the latter assumption does not hold when, for instance, first-class continuations are used.[1]

We refer to the examples in [3,4] to illustrate the impact of these benefits.

In this work, we show that the combination of abstract GC and the pushdown control flow of abstract definitional interpreters can not only keep the advantages of both techniques, but can in fact do even better than "the best of both worlds". That is, we identify an additional benefit that can be exploited by abstract GC when applied to abstract definitional interpreters:

[1] However, it can be argued that a CPS transformation of the program under analysis can be used to support such language features.

Stackless Abstract GC. As exemplified by the work of Germane et al. [8], abstract GC for big-step abstract interpreters can (temporarily) reclaim addresses that are still reachable from the (implicit) continuation. That is, such "stackless abstract GC" does not take references from the continuation into account, resulting in a smaller GC root set, and therefore also a more aggressive form of abstract GC (and therefore in turn amplifying its aforementioned advantages). We can illustrate this using the following Scheme program as an example:

```
(define (f n) (if (even? n)
                  (/ n 2)
                  (let ((r (f (+ n 1)))) (+ r n))))
(f 5)
```

For a context-insensitive abstract interpreter without abstract GC, the recursive call to `f` would allocate the argument `(+ n 1)` at the same address for the parameter `n` as the original argument `5`, therefore joining both values at this address and losing precision. However, when abstract GC is applied to a small-step abstract interpreter with explicit continuations (e.g., à la AAM), this precision loss would also not be avoided. The reason is that `n` is still reachable by the continuation when the recursive call to `f` happens, and therefore its corresponding address can not be reclaimed. In contrast, when stackless abstract GC is applied to a big-step abstract interpreter, references from the stack do not contribute to the GC root set (i.e., only addresses reachable from the current environment are preserved). As such, `n` is not kept in scope, and its address can be reclaimed before it is reallocated for the recursive call to `f`, avoiding this precision loss.

1.2 Approach: Delta Store Semantics

In this work, we present delta store semantics (DSS), a new formalism to integrate abstract GC into abstract definitional interpreters, exploiting all the aforementioned advantages. DSS closely resembles the original big-step abstract semantics, making only a simple change by returning a *delta store* (containing all changes to the original store) instead of directly returning an updated store. This modification opens up two interesting opportunities:

- It enables a simple and efficient integration of abstract GC into the big-step abstract semantics (similar in spirit to the *effect logs* of Germane et al. [8,9]). That is, when abstract GC is enabled, the delta store that is computed represents changes w.r.t. a minimal, garbage-collected store. Subsequently, we can *replay* those changes on a non-garbage-collected store, where the bindings that are necessary for the continuation are still present.
- A delta store is a more efficient, minimal representation for the result of an evaluation compared to the full updated store. Just like regular stores, delta stores are joinable. We propose a novel form of store widening for DSS, which still allows for strong updates and (limited) abstract GC. Such store widening involves joining the results of evaluations at the same program point, which is more efficient using delta stores (as we only have to join the changed bindings instead of the entire stores).

These two opportunities form the outline for the core of this paper. Section 3 presents the big-step abstract semantics with delta stores; Section 4 and Section 5 show how abstract GC and flow-sensitive store widening can be integrated into this semantics, respectively. We evaluate the impact of both in Section 6.

Contributions. The contributions of this work are as follows:

– We present delta store semantics (DSS), a novel formulation of big-step abstract semantics where evaluation steps return *minimal* and *composable* delta stores (representing changes to a store) instead of full updated stores. Our formalism resembles the heap fragment semantics of Germane et al. [8,9], but is arguably simpler and closer to the original big-step semantics.
– We show how abstract GC can be integrated into DSS, exploiting the full potential of combining abstract GC with pushdown control flow (i.e., realizing all benefits listed in Section 1.1). As such, this integration achieves better precision than classical abstract GC for small-step abstract interpreters. We formulate this claim as a theorem, and provide a mechanised proof in Rocq.
– We show how a new form of store widening can be integrated into DSS, further exploiting the minimal and compositional nature of delta stores. This form of store widening closely resembles the traditional notion of flow sensitivity, and still allows for strong updates and (a limited form of) abstract GC.
– We provide an implementation of DSS (including the integrations of abstract GC and flow-sensitive store widening) in the MAF framework to abstract interpretation. Using this implementation, we conduct an empirical evaluation to measure the impact of abstract GC (specifically, the impact of the theorem formally shown in Rocq) and flow-sensitive store widening for DSS.

2 Background

We present the formal specification of a minimal higher-order language λ_{ANF} with support for mutable variables (to model strong updates). Below, we define the syntax of λ_{ANF}, based on the λ-calculus in A-Normal Form [5] (ANF).

$$\begin{array}{rlcrl}
e \in \mathsf{Exp} ::= & ae \mid f(ae) & \quad & f, ae \in \mathsf{Atom} ::= & x \mid lam \\
\mid & \textbf{let } x = e_1 \textbf{ in } e_2 & & lam \in \mathsf{Lam} ::= & \lambda x.e \\
\mid & \textbf{set } x := ae \textbf{ then } e & & x \in \mathsf{Var} & \text{(a set of identifiers)}
\end{array}$$

ANF is a syntactic form restricting operators and operands to atomic expressions *ae* which can be evaluated immediately without impacting program state. This simplification can be automated, is purely cosmetic, and without loss of generality.

We define both the abstract small-step and big-step semantics for λ_{ANF}, and discuss the integration of abstract counting and abstract GC.

2.1 Small-Step Abstract Semantics of λ_{ANF}

We present the small-step abstract semantics of λ_{ANF} (the "operational semantics") using the AAM technique [30] to abstract interpretation.

State Space. The state space Σ of the abstract interpreter is given below.[2][3]

$$
\begin{array}{rlrl}
\varsigma \in \Sigma & ::= \langle c, \sigma, \sigma_k, a_k \rangle & \sigma_k \in \mathsf{KStore} & = \mathsf{KAddr} \to \mathsf{Kont} \\
c \in \mathsf{Control} & ::= \mathbf{ev}(e, \rho) \mid \mathbf{ap}(v) & \kappa \in \mathsf{Kont} & = \mathcal{P}(\mathsf{Frame}) \\
\rho \in \mathsf{Env} & = \mathsf{Var} \rightharpoonup \mathsf{Addr} & \mathit{frm} \in \mathsf{Frame} & ::= \mathbf{letk}(x, e, \rho, a_k) \\
\sigma \in \mathsf{Store} & = \mathsf{Addr} \to \mathsf{Val} & l \in \mathsf{Loc} & = \mathsf{Addr} \cup \mathsf{KAddr} \\
v \in \mathsf{Val} & = \mathcal{P}(\mathsf{Clo}) & a \in \mathsf{Addr} & \text{(a finite set)} \\
\mathit{clo} \in \mathsf{Clo} & = \mathsf{Lam} \times \mathsf{Env} & a_k \in \mathsf{KAddr} & \text{(a finite set)}
\end{array}
$$

A state ς consists of a control component (either an expression e under evaluation paired with an environment ρ, or a resulting value v), a store σ (to model "the heap"), a continuation store σ_k (to model "the stack"), and the current continuation address a_k (pointing to "the top of the stack"). While the sets Addr and KAddr are infinite in a concrete interpreter (e.g., by picking $\mathsf{Addr} = \mathsf{KAddr} = \mathbb{N}$), for an abstract interpreter both sets should be finite. One can easily show that this restriction ensures that the state space Σ also remains finite.
As the abstract interpreter can only use a finite number of addresses, it may need to reuse the same address for multiple allocations. Closures and continuation frames that end up at the same address need to be *joined* together to obtain a sound, but finite approximation. Therefore, values and continuations are represented as *sets* of closures and continuation frames, respectively. We define the *join operator* ($\sqcup$), the *subsumption relation* ($\sqsubseteq$) and the *global lower bound* ($\bot$) for Val (and analogously for Kont): $v_1 \sqcup v_2 = v_1 \cup v_2, v_1 \sqsubseteq v_2 \iff v_1 \subseteq v_2$ and $\bot = \emptyset$. We trivially extend the definitions of ($\bot$), ($\sqcup$) and ($\sqsubseteq$) to functions and pairs. For example, $(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup a_2, b_1 \sqcup b_2)$ and $(f_1 \sqcup f_2)(a) = f_1(a) \sqcup f_2(a)$, while ($\bot$) is defined so that $\bot \sqcup v = v \sqcup \bot = v$ and ($\sqsubseteq$) so that $a \sqsubseteq b \iff a \sqcup b = b$.

Evaluation Rules. Atomic expressions can be evaluated in a single step, without making any modifications to the store. To evaluate such atomic expressions, we introduce an auxiliary function $\mathcal{A} : \mathit{Atom} \times \mathsf{Env} \times \mathsf{Store} \to \mathsf{Val}$.

$$\mathcal{A}(x, \rho, \sigma) = \mathsf{lookup}(\sigma, \rho(x)) \qquad \mathcal{A}(\mathit{lam}, \rho, \sigma) = \{\langle \mathit{lam}, \rho \rangle\}$$

[2] Note that it is often customary to put $\widehat{hats}$ on abstracted components, in order to distinguish them from their concrete counterparts. We only present the abstract semantics, and therefore omit these hats to improve readability.

[3] For a state ς, we implicitly use subscripted notations so that $\varsigma = \langle c_\varsigma, \sigma_\varsigma, \sigma_{k_\varsigma}, a_{k_\varsigma} \rangle$. For (partial) maps, $[]$ denotes the empty map, $[a \mapsto b]$ denotes a map m with a single binding (i.e., $dom(m) = \{a\}$ and $m(a) = b$), and the notation $m[a \mapsto b]$ extends the map m so that $m[a \mapsto b](a) = b$ and $m[a \mapsto b](x) = m(x)$ for $x \neq a$.

where currently, lookup is simply defined as $\mathsf{lookup}(\sigma, a) = \sigma(a)$.

Below, we define the small-step transition relation $(\rightarrow) \subseteq \Sigma \times \Sigma$ for λ_{ANF}. Note that $(\rightarrow)$ is not deterministic due to its over-approximating behaviour.

$$\frac{v = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(ae, \rho), \sigma, \sigma_k, a_k \rangle \rightarrow \langle \mathbf{ap}(v), \sigma, \sigma_k, a_k \rangle} \quad (\textsc{St-Atom})$$

$$\frac{\langle \lambda x.e, \rho' \rangle \in \mathcal{A}(f, \rho, \sigma) \qquad v = \mathcal{A}(ae, \rho, \sigma) \qquad a = \mathsf{alloc}(x)}{\langle \mathbf{ev}(f(ae), \rho), \sigma, \sigma_k, a_k \rangle \rightarrow \langle \mathbf{ev}(e, \rho'[x \mapsto a]), \mathsf{extend}(\sigma, a, v), \sigma_k, a_k \rangle} \quad (\textsc{St-App})$$

$$\frac{a'_k = \mathsf{alloc}_k(e_1) \qquad \kappa = \{\mathbf{letk}(x, e_2, \rho, a_k)\}}{\langle \mathbf{ev}(\mathbf{let}\ x = e_1\ \ \mathbf{in}\ e_2, \rho), \sigma, \sigma_k, a_k \rangle \rightarrow \langle \mathbf{ev}(e_1, \rho), \sigma, \mathsf{extend}_k(\sigma_k, a'_k, \kappa), a'_k \rangle} \quad (\textsc{St-Let1})$$

$$\frac{\mathbf{letk}(x, e, \rho, a'_k) \in \sigma_k(a_k) \qquad a = \mathsf{alloc}(x)}{\langle \mathbf{ap}(v), \sigma, \sigma_k, a_k \rangle \rightarrow \langle \mathbf{ev}(e, \rho[x \mapsto a]), \mathsf{extend}(\sigma, a, v), \sigma_k, a'_k \rangle} \quad (\textsc{St-Let2})$$

$$\frac{v = \mathcal{A}(ae, \rho, \sigma) \qquad a = \rho(x) \qquad \sigma' = \mathsf{update}(\sigma, a, v)}{\langle \mathbf{ev}(\mathbf{set}\ x := ae\ \mathbf{then}\ e, \rho), \sigma, \sigma_k, a_k \rangle \rightarrow \langle \mathbf{ev}(e, \rho), \sigma', \sigma_k, a_k \rangle} \quad (\textsc{St-Set})$$

We leave the choice of sets Addr and KAddr, as well as the allocation functions alloc and alloc_k open as configuration parameters of the abstract interpreter (resulting in a particular *allocation policy*). Any allocation policy yields a sound and decidable analysis [30] (as long as the sets chosen for Addr and KAddr are finite). However, the choice is not arbitrary, as the allocation policy decides how often (determined by the size of Addr and KAddr) and when (determined by alloc and alloc_k) addresses need to be reused. This choice therefore affects the precision and *polyvariance* [10] of the abstract interpreter.[4]

Currently, we define functions extend (analogously, extend_k) and update as:

$$\mathsf{extend}(\sigma, a, v) = \mathsf{update}(\sigma, a, v) = \sigma[a \mapsto \sigma(a) \sqcup v]$$

Abstract Counting. We can further improve the precision of the abstract interpreter using abstract counting [19,1]. With abstract counting, one keeps track for every allocated address a in σ whether it has been allocated (concretely) only once or possibly multiple times. Doing so can avoid joining the old and new value in the St-Set rule in certain cases. Formally, we define

$$n \in \mathsf{Count} := 0 \ \mid\ 1 \ \mid\ \infty$$

$$n_1 \sqcup n_2 = \begin{cases} n_2 & \text{if } n_1 \sqsubseteq n_2 \\ n_1 & \text{otherwise} \end{cases} \qquad \mathsf{inc}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \infty & \text{otherwise} \end{cases}$$

[4] For the sake of simplicity, our abstract interpreter does not include a timestamp component (as in [30]), which could be used to express more complex allocation policies such as k-CFA with $k > 1$.

and the (reflexive and transitive) subsumption relation ($\sqsubseteq$) as $0 \sqsubseteq 1 \sqsubseteq \infty$.

We then modify the definition of Store to keep track of abstract counts[5]:

$$\sigma \in \mathsf{Store} = \mathsf{Addr} \to (\mathsf{Val} \; \times \mathsf{Count}\;)$$

Extending the store with a newly-allocated address (using extend) increases the abstract count at that location. Updating the address at an existing address (using update) does not increase the abstract count. If the existing abstract count for that address is ∞, we are not able to replace the value at that location (because that address may represent multiple allocations, and we are only updating one of them); in this case, the new value is still joined together with an old value (called a *weak update*). However, if the abstract count at this location is 1, then we know that we can safely replace the old value with the new one (called a *strong update*), thereby increasing the precision of the St-Set rule. If $\langle v_a, c_a\rangle = \sigma(a)$, then:

$$\mathsf{lookup}(\sigma, a) = v_a \qquad \mathsf{extend}(\sigma, a, v) = \sigma[a \mapsto \langle v_a \sqcup v, \; \mathsf{inc}(c_a)\; \rangle]$$

$$\mathsf{update}(\sigma, a, v) = \begin{cases} \;\sigma[a \mapsto \langle v, 1\rangle]\; & \mathsf{if}\ c_a = 1 \\ \sigma[a \mapsto \langle v_a \sqcup v, c_a\rangle] & \mathsf{otherwise} \end{cases}$$

Abstract Garbage Collection. We now add garbage collection to further improve precision. We use notation and definitions similar to those used by Might and Shivers [21]. First, we define a family of auxiliary functions $\mathcal{T}_X : X \to \mathcal{P}(\mathsf{Loc})$ that return all addresses that are referenced directly by some element of type X.

$$\begin{aligned} \mathcal{T}_\Sigma(\langle c, \sigma, \sigma_k, a_k\rangle) &= \mathcal{T}_{\mathsf{Control}}(c) \cup \{a_k\} & \mathcal{T}_{\mathsf{Clo}}(\langle \lambda x.e, \rho\rangle) &= \mathcal{T}_{\mathsf{Env}}(\rho) \\ \mathcal{T}_{\mathsf{Control}}(\mathbf{ev}(e, \rho)) &= \mathcal{T}_{\mathsf{Env}}(\rho) & \mathcal{T}_{\mathsf{Env}}(\rho) &= \mathsf{range}(\rho) \\ \mathcal{T}_{\mathsf{Control}}(\mathbf{ap}(v)) &= \mathcal{T}_{\mathsf{Val}}(v) & \mathcal{T}_{\mathsf{Frame}}(\mathbf{letk}(x, e, \rho, a_k)) &= \mathcal{T}_{\mathsf{Env}}(\rho) \cup \{a_k\} \\ \mathcal{T}_{\mathsf{Val}}(v) &= \bigcup_{clo \in v} \mathcal{T}_{\mathsf{Clo}}(clo) & \mathcal{T}_{\mathsf{Kont}}(\kappa) &= \bigcup_{frm \in \kappa} \mathcal{T}_{\mathsf{Frame}}(frm) \end{aligned}$$

Next, we introduce the adjacency relation between addresses $(\leadsto_\varsigma) \subseteq \mathsf{Loc} \times \mathsf{Loc}$, where intuitively $l \leadsto_\varsigma l'$ means that there is a reference from l to l'. We define $(\leadsto_\varsigma) = (\leadsto_{\sigma_\varsigma}) \cup (\leadsto_{\sigma_{k_\varsigma}})$, using the following auxiliary adjacency relations:

$$a \leadsto_\sigma l \iff l \in \mathcal{T}_{\mathsf{Val}}(\mathsf{lookup}(\sigma, a)) \qquad a_k \leadsto_{\sigma_k} l \iff l \in \mathcal{T}_{\mathsf{Kont}}(\sigma_k(a_k))$$

The auxiliary function $\mathcal{R}_{(\leadsto)} : \mathcal{P}(\mathsf{Loc}) \to \mathcal{P}(\mathsf{Loc})$ computes all addresses transitively reachable from a given root set using a transition relation ($\leadsto$):

$$\mathcal{R}_{(\leadsto)}(roots) = lfp(f) \;\; \mathbf{where} \;\; f(S) = roots \cup \{l' \in \mathsf{Loc} \mid \exists l \in S : l \leadsto l'\}$$

The function $\mathcal{R} : \Sigma \to \mathcal{P}(\mathsf{Loc})$ computes all reachable addresses of a state:

$$\mathcal{R}(\varsigma) = \mathcal{R}_{(\leadsto_\varsigma)}(\mathcal{T}_\Sigma(\varsigma)).$$

[5] Important changes with respect to previous definitions are highlighted in *gray* .

Using these definitions, we can define garbage collection for a state ς as a function $\Gamma : \Sigma \to \Sigma$ that restricts the stores of ς to its reachable addresses $\mathcal{R}(\varsigma)$.

$$\Gamma(\varsigma) = \langle c_\varsigma, \sigma_\varsigma|_{\mathcal{R}(\varsigma)}, \sigma_{k\varsigma}|_{\mathcal{R}(\varsigma)}, a_k \rangle$$

where $\sigma|_R(a) = \sigma(a)$ if $a \in R$ and $\langle \bot, 0 \rangle$ otherwise (analogously for $\sigma_k|_R$).

To incorporate garbage collection into our evaluation rules, we define a new transition relation $(\rightarrow_\Gamma)$ as a composition of the garbage collection function Γ and the existing transition relation $(\rightarrow)$, i.e. $(\rightarrow_\Gamma) = \Gamma \circ (\rightarrow)$. As such, $(\rightarrow_\Gamma)$ applies GC after every step. This ensures that no garbage can be created by $(\rightarrow_\Gamma)$, rendering the abstract interpreter *garbage-free* [30,29].

Note that abstract GC synergises with abstract counting: abstract counts for collected addresses are reset to 0, increasing opportunities for future strong updates. Conversely, by applying strong updates, more garbage can be collected, as addresses reachable from the overwritten value no longer contribute to $\mathcal{R}(\varsigma)$.

Program Semantics. The collecting semantics of a program can now be defined using the function $\mathcal{S} : \mathsf{Exp} \to \mathcal{P}(\Sigma)$, which computes all the states reachable by the abstract interpreter, starting from the initial state of the program:[6]

$$\mathcal{S}(e) = \{\varsigma \in \Sigma \mid \langle \mathbf{ev}(e, []), \bot_\sigma, \bot_{\sigma_k}, a_{\mathsf{halt}} \rangle \overset{*}{\rightarrow}_\Gamma \varsigma\}$$

where $\bot_\sigma(a) = \langle \bot, 0 \rangle$, $\bot_{\sigma_k}(a) = \bot$ and a_{halt} is a special address in KAddr. As Σ is finite, for any program e it is guaranteed that $\mathcal{S}(e)$ is finite and therefore computable. We can reason over the behaviour of e by reasoning over $\mathcal{S}(e)$, yielding a sound and decidable program analysis. Note that the definition of $\mathcal{S}$ reveals the benefit of garbage irrelevance: as $(\rightarrow_\Gamma)$ is garbage-free, $\mathcal{S}$ only needs to explore the subset of Σ where $\varsigma = \Gamma(\varsigma)$.

We also define a function $\mathsf{eval} : \mathsf{Exp} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Store})$, returning a set of all possible return values (along with their corresponding store) for a program:

$$\mathsf{eval}(e) = \{(v, \sigma) \mid \langle \mathbf{ap}(v), \sigma, \sigma_k, a_{\mathsf{halt}} \rangle \in \mathcal{S}(e)\}$$

2.2 Big-Step Abstract Semantics of λ_{ANF}

We present the big-step abstract semantics of λ_{ANF} (the "definitional semantics") following the approach of Darais et al. [2] to abstract interpretation.

Evaluation Rules. For an abstract definitional interpreter, the semantics are formulated as recursive evaluation rules using a big-step relation $(\Downarrow) \subseteq \mathsf{Config} \times \mathsf{Result}$. We define input configurations to evaluate (Config) and the result of the evaluation for that configuration (Result) as follows:

$$\mathit{conf} \in \mathsf{Config} = \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Store} \qquad \mathit{res} \in \mathsf{Result} = \mathsf{Val} \times \mathsf{Store}$$

[6] We write $(\overset{*}{\rightarrow}_\Gamma)$ for the reflexive, transitive closure of $(\rightarrow_\Gamma)$.

That is, a configuration $\langle e, \rho, \sigma \rangle$ contains an expression e to evaluate using the environment ρ and store σ. In contrast to states in Σ, configurations crucially do not carry a continuation component, which is key to achieve context irrelevance (as configurations with the same expression, environment and store, but used in different evaluation contexts are still considered equal) and avoids a loss in stack precision (since there is no need to approximate this component). The result of an evaluation $\langle v, \sigma' \rangle$ holds the resulting value v along with the updated store σ'.

The big-step transition relation ($\Downarrow$) is defined below. Note again that — just as with the small-step transition relation ($\rightarrow$) — there is nondeterminism due to the over-approximating behaviour of the abstract interpreter.

$$\frac{v = \mathcal{A}(ae, \rho, \sigma)}{\langle ae, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle} \quad \text{(E-Atom)}$$

$$\frac{\langle \lambda x.e, \rho' \rangle \in \mathcal{A}(f, \rho, \sigma) \qquad a_x = \mathsf{alloc}(x) \qquad v_x = \mathcal{A}(ae, \rho, \sigma) \\ \langle e, \rho'[x \mapsto a_x], \mathsf{extend}(\sigma, a_x, v_x) \rangle \Downarrow \langle v, \sigma' \rangle}{\langle f(ae), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad \text{(E-App)}$$

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_x, \sigma' \rangle \qquad a_x = \mathsf{alloc}(x) \\ \langle e_2, \rho[x \mapsto a_x], \mathsf{extend}(\sigma', a_x, v_x) \rangle \Downarrow \langle v, \sigma'' \rangle}{\langle \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, \rho, \sigma \rangle \Downarrow \langle v, \sigma'' \rangle} \quad \text{(E-Let)}$$

$$\frac{a_x = \rho(x) \qquad v_x = \mathcal{A}(ae, \rho, \sigma) \\ \langle e, \rho, \mathsf{update}(\sigma, a_x, v_x) \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathbf{set}\ x := ae\ \mathbf{then}\ e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad \text{(E-Set)}$$

Note that strong updates are already supported in these semantics, since we are reusing the abstract-counting definitions of extend and update.

Abstract Garbage Collection. For the small-step abstract interpreter of Section 2.1, we computed the GC root set for the current state ς using the addresses directly reachable from the control component ($\mathcal{T}_{\mathsf{Control}}(c_\varsigma)$) and the current root of the continuation component (a_{k_ς}). For the abstract definitional interpreter presented here, we do not have access to the latter, as the continuation component is not explicitly reified. Nevertheless, addresses reachable from the continuation can not just be ignored: if they are not included in the GC root set, addresses that are used later on may accidentally be collected. The updated store σ' of an evaluation step $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$ is then no longer safe to continue evaluation with. For instance, in the E-Let rule, it is crucial that the updated store σ' (returned after evaluating e_1) still contains all addresses reachable from ρ, so that it can safely be used to evaluate e_2.

In order to address this, Darais et al. [2] show how abstract GC can still be integrated into an abstract definitional interpreter by explicitly passing along all addresses reachable from the (implicit) continuation (i.e., all addresses that need to be preserved, as they may be used after the current evaluation step). Concretely, this requires passing along a set of addresses ψ in each configuration:

$$\mathit{conf} \in \mathsf{Config} = \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Store} \times \mathcal{P}(\mathsf{Addr})$$

Incorporating GC into the evaluation rules then requires two modifications:

- Each recursive evaluation step passes along the set of stack addresses ψ that need to be preserved, along with additional addresses that are still needed to continue with the remaining evaluation steps in that rule. The latter corresponds to the addresses that would previously be directly reachable from the Frame (i.e., using $\mathcal{T}_{\mathsf{Frame}}$) allocated in the corresponding rule of the small-step semantics. Due to our simplification to ANF, the only rule with multiple recursive evaluation steps is the E-Let rule; addresses still required for the second evaluation step (in this case, $\mathcal{T}_{\mathsf{Env}}(\rho)$) are therefore added to ψ for the first evaluation step. As the second evaluation step is also the final one (i.e., e_2 is evaluated in *tail* position, so it does not grow the (implicit) continuation), no additional addresses need to be preserved.
- We add a new rule to collect garbage: for a configuration $\langle e, \rho, \sigma, \psi\rangle$, it uses $\psi \cup \mathcal{T}_{\mathsf{Env}}(\rho)$ as the GC root set, using the garbage-collected store to evaluate e, and subsequently also cleans up the updated store σ' of the result $\langle v, \sigma'\rangle$ using the GC root set $\psi \cup \mathcal{T}_{\mathsf{Val}}(v)$.[7] To interleave this rule with the existing evaluation rules, we introduce a new relation $(\Downarrow_\Gamma)$ with such a GC rule:

$$\frac{\begin{array}{c}\sigma_{\mathsf{gc}} = \sigma|_{\mathcal{R}_{(\rightsquigarrow_\sigma)}(\psi\,\cup\,\mathcal{T}_{\mathsf{Env}}(\rho))} \\ \langle e, \rho, \sigma_{\mathsf{gc}}, \psi\rangle \Downarrow \langle v, \sigma'\rangle \\ \sigma'_{\mathsf{gc}} = \sigma'|_{\mathcal{R}_{(\rightsquigarrow_{\sigma'})}(\psi\,\cup\,\mathcal{T}_{\mathsf{Val}}(v))}\end{array}}{\langle e, \rho, \sigma, \psi\rangle \Downarrow_\Gamma \langle v, \sigma'_{\mathsf{gc}}\rangle}\ \text{(E-GC)}$$

  Subsequently, we modify the other evaluation rules to use $(\Downarrow_\Gamma)$ instead of $(\Downarrow)$ for recursive evaluation steps. This effectively results in the big-step equivalent of the "GC at every step" policy used in $(\rightarrow_\Gamma)$.

  The updated evaluation rules are shown below:

$$\frac{v = \mathcal{A}(\mathrm{ae}, \rho, \sigma)}{\langle \mathrm{ae}, \rho, \sigma, \psi\rangle \Downarrow \langle v, \sigma\rangle}\ \text{(E-Atom)}$$

$$\frac{\begin{array}{c}\langle \lambda x.e, \rho'\rangle \in \mathcal{A}(f, \rho, \sigma) \qquad a_x = \mathsf{alloc}(x) \qquad v_x = \mathcal{A}(\mathrm{ae}, \rho, \sigma) \\ \langle e, \rho'[x \mapsto a_x], \mathsf{extend}(\sigma, a_x, v_x), \psi\rangle \Downarrow_\Gamma \langle v, \sigma'\rangle\end{array}}{\langle f(\mathrm{ae}), \rho, \sigma, \psi\rangle \Downarrow \langle v, \sigma'\rangle}\ \text{(E-App)}$$

[7] Note that we are able to collect more garbage here compared to the original work of Darais et al. [2], as we GC not only the result store, but also the configuration store.

$$\frac{\begin{array}{c}\langle e_1, \rho, \sigma, \boxed{\psi \cup \mathcal{T}_{\mathsf{Env}}(\rho)} \rangle \boxed{\Downarrow_{\Gamma}} \langle v_x, \sigma' \rangle \qquad a_x = \mathsf{alloc}(x) \\ \langle e_2, \rho[x \mapsto a_x], \mathsf{extend}(\sigma', a_x, v_x), \boxed{\psi} \rangle \boxed{\Downarrow_{\Gamma}} \langle v, \sigma'' \rangle\end{array}}{\langle \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, \rho, \sigma, \boxed{\psi} \rangle \Downarrow \langle v, \sigma'' \rangle} \quad \text{(E-Let)}$$

$$\frac{\begin{array}{c}a_x = \rho(x) \qquad v_x = \mathcal{A}(\mathit{ae}, \rho, \sigma) \\ \langle e, \rho, \mathsf{update}(\sigma, a_x, v_x), \boxed{\psi} \rangle \boxed{\Downarrow_{\Gamma}} \langle v, \sigma' \rangle\end{array}}{\langle \mathbf{set}\ x := \mathit{ae}\ \mathbf{then}\ e, \rho, \sigma, \boxed{\psi} \rangle \Downarrow \langle v, \sigma' \rangle} \quad \text{(E-Set)}$$

While this approach can safely collect garbage to improve precision, it still loses out on the full advantages of both abstract GC and pushdown control flow.

First, the abstract interpreter no longer enjoys context irrelevance. While the continuation itself is not directly part of a configuration, the addresses reachable from the continuation are. Configurations that only differ in the context in which they are used (i.e., they have different continuations), and that would otherwise be equal, may therefore be analysed multiple times by the analysis (i.e., when the set of reachable addresses ψ is also different, leading to multiple configurations).

Second, the abstract interpreter can not take advantage of the pushdown control flow to collect more garbage. That is, it can not reclaim any bindings reachable from the continuation (as these are kept alive by ψ), even though these are not actually used for the current evaluation step. An optimal approach would be able to more aggressively collect garbage using a smaller root set (i.e., without ψ), only adding back the bindings reachable by ψ *after* the evaluation step to construct the updated store to continue evaluation with.

Program Semantics. As with the small-step semantics, we can define a function $\mathsf{eval} : \mathsf{Exp} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Store})$ that returns all possible return values (and stores):

$$\mathsf{eval}(e) = \{(v, \sigma) \mid \langle e, [], \bot_\sigma \rangle \Downarrow_{\Gamma} \langle v, \sigma \rangle\}$$

We refer to the work of Darais et al. [2] for a cache-based fixpoint algorithm to compute the (finite) big-step relation ($\Downarrow_{\Gamma}$). The same paper also shows how the evaluation rules can be instrumented to compute the collecting semantics.

3 Delta Store Semantics

We now present the core foundation of delta store semantics (DSS). DSS closely resembles the standard big-step semantics presented in Section 2.2. The main difference is that evaluation now returns a *delta store* (representing changes to the original store) instead of the entire updated store.

We first introduce delta stores in Section 3.1, along with a set of operations on delta stores defining how they are *joined* and *composed*. We also show how a delta store can be applied to the original store, in order to reconstruct the updated store. In Section 3.2, we then modify the abstract big-step semantics for λ_{ANF}, resulting in an equivalent semantics formulated using delta stores.

3.1 Delta Stores

A delta store captures bindings that were changed with respect to a given store σ. Put differently, it is the subset of the updated store containing all bindings that σ has potentially[8] been extended or updated with.

Formally, we define delta stores similarly to regular stores:

$$\delta \in \mathsf{Delta} = \mathsf{Addr} \rightharpoonup (\mathsf{Val} \times \mathsf{Count})$$

The difference is that a delta store is represented as a *partial* function, since it only maps addresses that have been modified w.r.t. the original store.

In what follows, we show how delta stores can be applied, composed, and joined.

Application. A delta store δ, representing changes over an original store σ, can be applied to σ in order to construct the updated store in which these changes are directly integrated. Since both stores and delta stores are represented as functions, we can use the mathematical *override* operator ($\rhd$) for this purpose:

$$(f_1 \rhd f_2)(a) = \begin{cases} f_1(a) & \textbf{if } \; a \in dom(f_1) \\ f_2(a) & \textbf{if } \; a \notin dom(f_1) \land a \in dom(f_2) \end{cases}$$

That is, we can write $\delta \rhd \sigma$ to apply the changes of delta store δ to the original store σ. Note that the result of this operation is a Store, as $dom(\sigma) = \mathsf{Addr}$, so that $\delta \rhd \sigma$ is always a total function $\mathsf{Addr} \to (\mathsf{Val} \times \mathsf{Count}) = \mathsf{Store}$. A lookup in this store will first look for the address in the delta store δ, and for unchanged bindings (i.e., that are not in δ) fall back to the original store σ.

Composition. We also require an operator to combine multiple delta stores that represent a sequence of changes (e.g., resulting from the evaluation of multiple expressions in a sequence). That is, given a delta store δ_1 (representing changes over a store σ) followed by a delta store δ_2 (representing changes over $\delta_1 \rhd \sigma$), we want to be able to combine these changes into a single delta store.

To this end, we can use the same override operator ($\rhd$) as an operator for composition. A series of changes represented in order by delta stores $\delta_1, \delta_2, ..., \delta_n$ can then be composed into a single delta store $\delta_n \rhd ... \rhd \delta_2 \rhd \delta_1$, which (when applied to a store σ) will integrate all changes of these delta stores in that same order. Note that, when applied to two delta stores δ_1 and δ_2, the result of the composition is again a delta store (i.e., a partial function) with $dom(\delta_2 \rhd \delta_1) = dom(\delta_1) \cup dom(\delta_2)$.

Join. Delta stores are joinable, but only when they represent changes over the same store σ, so that the resulting delta store also represents changes over σ. Therefore, we define a join operator ($\sqcup_\sigma$) parameterized over some store σ:

$$(\delta_1 \sqcup_\sigma \delta_2)(a) = \begin{cases} \delta_1(a) \sqcup \delta_2(a) & \textbf{if } \; a \in dom(\delta_1) \land a \in dom(\delta_2) \\ \delta_1(a) \sqcup \sigma(a) & \textbf{if } \; a \in dom(\delta_1) \land a \notin dom(\delta_2) \\ \sigma(a) \sqcup \delta_2(a) & \textbf{if } \; a \notin dom(\delta_1) \land a \in dom(\delta_2) \end{cases}$$

[8] A delta store may also contain updated bindings that happen to have the same value as in the original store σ, although this makes its usage slightly less efficient.

Note that $dom(\delta_1 \sqcup_\sigma \delta_2) = dom(\delta_1) \cup dom(\delta_2)$. When joining delta stores, it is important to consider what happens when an address is in one delta store, but not the other. In this case, we join the binding of the delta store with the binding of the original store σ, since the lack of a binding in the other delta store implies no change w.r.t. the original store (hence, the original binding is preserved).

The join operator ($\sqcup_\sigma$) is useful to handle non-determinism in the abstract semantics, as the results of multiple program paths (which each include a delta store) following a non-deterministic choice can be joined together into a single result. We will make use of this operator later in Section 5 when we integrate flow-sensitive store widening into the abstract semantics using delta stores.

3.2 Big-Step Abstract Semantics with Delta Stores for λ_{ANF}

We now formulate the big-step abstract semantics for λ_{ANF} using delta stores. The main difference to the big-step semantics presented in Section 2.2 is that all modifications to the store are now expressed as delta stores. We first adapt the auxiliary functions that modify the store, extend and update, to return a delta store (with a single change) instead of an updated store. If $\langle v_a, c_a\rangle = \sigma(a)$, then:

$$\mathsf{extend}(\sigma, a, v) = [a \mapsto \langle v_a \sqcup v, \mathsf{inc}(c_a)\rangle]$$

$$\mathsf{update}(\sigma, a, v) = \begin{cases} [a \mapsto \langle v, 1\rangle] & \text{if } c_a = 1 \\ [a \mapsto \langle v_a \sqcup v, c_a\rangle] & \text{otherwise} \end{cases}$$

Note that these definitions have remained mostly unchanged. The only difference is that the change is now expressed as a "portable" delta store, as opposed to being directly integrated into the store that is extended or updated.

Next, we introduce a new big-step evaluation relation ($\Downarrow^{\Delta}$), which returns a delta store instead of an updated store as part of the result. That is, we define:

$$res \in \mathsf{Result} = \mathsf{Val} \times \boxed{\mathsf{Delta}}$$

The updated evaluation rules for ($\Downarrow^{\Delta}$) are given below:

$$\frac{v = \mathcal{A}(\mathrm{ae}, \rho, \sigma)}{\langle \mathrm{ae}, \rho, \sigma\rangle \Downarrow^{\Delta} \langle v, \boxed{[]}\rangle} \quad \text{(E-ATOM)}$$

$$\frac{\begin{array}{ccc}\langle \lambda x.e, \rho'\rangle \in \mathcal{A}(f, \rho, \sigma) & a_x = \mathsf{alloc}(x) & v_x = \mathcal{A}(\mathrm{ae}, \rho, \sigma) \\ \boxed{\delta_x = \mathsf{extend}(\sigma, a_x, v_x)} & \multicolumn{2}{c}{\langle e, \rho'[x \mapsto a_x], \boxed{\delta_x \rhd \sigma}\rangle \Downarrow^{\Delta} \langle v, \boxed{\delta}\rangle}\end{array}}{\langle f(\mathrm{ae}), \rho, \sigma\rangle \Downarrow^{\Delta} \langle v, \boxed{\delta \rhd \delta_x}\rangle} \quad \text{(E-APP)}$$

$$\frac{\langle e_1, \rho, \sigma\rangle \Downarrow^{\Delta} \langle v_x, \boxed{\delta_1}\rangle \qquad \sigma' = \boxed{\delta_1 \rhd \sigma} \qquad a_x = \mathsf{alloc}(x) \qquad \boxed{\delta_x = \mathsf{extend}(\sigma', a_x, v_x)} \qquad \langle e_2, \rho[x \mapsto a_x], \boxed{\delta_x \rhd \sigma'}\rangle \Downarrow^{\Delta} \langle v, \boxed{\delta_2}\rangle}{\langle \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, \rho, \sigma\rangle \Downarrow^{\Delta} \langle v, \boxed{\delta_2 \rhd \delta_x \rhd \delta_1}\rangle} \quad \text{(E-Let)}$$

$$\frac{a_x = \rho(x) \qquad v_x = \mathcal{A}(ae, \rho, \sigma) \qquad \boxed{\delta_x = \mathsf{update}(\sigma, a_x, v_x)} \qquad \langle e, \rho, \boxed{\delta_x \rhd \sigma}\rangle \Downarrow^{\Delta} \langle v, \boxed{\delta}\rangle}{\langle \mathbf{set}\ x := ae\ \mathbf{then}\ e, \rho, \sigma\rangle \Downarrow^{\Delta} \langle v, \boxed{\delta \rhd \delta_x}\rangle} \quad \text{(E-Set)}$$

When evaluating an atomic expression (E-Atom), the store is not modified, and hence the empty delta store [] is returned. When evaluating an application (E-App), the store is first extended with a binding for the argument (represented by δ_x), and afterwards also potentially modified during the evaluation of the function body (represented by δ). The resulting delta store is therefore $\delta \rhd \delta_x$. Similarly, when evaluating a let expression (E-Let), the resulting delta store composes the changes made by the evaluation of the right-hand side expression e_1, the binding of the variable x, and the evaluation of the body e_2. When evaluating a variable assignment (E-Set), we compose the change of the update operation with the changes returned by the subsequent evaluation.

One can show that the abstract big-step semantics defined with ($\Downarrow^{\Delta}$) is equivalent to the semantics defined with ($\Downarrow$), as stated by Theorem 1.

Theorem 1. $\forall e, \rho, \sigma, \sigma', v : \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle \iff \exists \delta : \langle e, \rho, \sigma\rangle \Downarrow^{\Delta} \langle v, \delta\rangle \wedge \sigma' = \delta \rhd \sigma$

4 Integrating Abstract GC

We now make use of these new big-step semantics using delta stores to integrate abstract GC. By using delta stores (instead of returning the updated store), it is now safe to evaluate an expression using an input store that has been garbage collected without taking references from the continuation into account (i.e., they may be removed by the GC). The key insight is that the computed delta store can later be *replayed* on the original store that has not yet been garbage collected, and in which the bindings necessary for the continuation are still present.

To enable this process, we first introduce such a replay operation for delta stores in Section 4.1. Using this new operation, we then integrate abstract GC into the delta store semantics in Section 4.2 In Section 4.3, we show that the resulting form of abstract GC is not just equivalent, but in fact superior to the traditional form of abstract GC found in small-step abstract interpreters.

4.1 Replaying Delta Stores

A key ingredient to allow the integration of abstract GC into the big-step semantics is the *replay* operation. It transforms a delta store δ (which was computed using a store where bindings for the continuation may have been collected) into a delta store δ' that "restores" the bindings necessary for the continuation (found in the original store σ). That is, it simulates ("replays") the changes captured by δ on σ, computing the changes that would have occured with respect to σ.

To illustrate the purpose of the replay operation, consider an abstract interpretation (with abstract GC) for the following Scheme program:

```
(define (make-adder n)
    (lambda (x) (+ x n)))
(let ((f1 (make-adder 1)))
    (let ((f2 (make-adder 2)))
        (f2 (f1 0))))
```

Assuming a context-insensitive allocation policy, the abstract interpreter will allocate the same address for the variable `n` (henceforth denoted as `@n`) in both calls to `make-adder`. The first call will return a delta store containing the binding `@n -> 1`. This binding can (temporarily) be garbage collected when evaluating the second call: while it is still necessary for the continuation of that call, its evaluation itself does not require this binding. The second call will therefore return a delta store containing the binding `@n -> 2`. However, after returning from the second call, we can not just continue evaluation using a store that binds `@n` to `2`. That is, we first need to use the replay operation to restore the old binding as well, resulting in a store where `@n` is bound to the join of `1` and `2`.

Formally, we define $\mathsf{replay} : \mathsf{Delta} \to \mathsf{Store} \to \mathcal{P}(\mathsf{Addr}) \to \mathsf{Delta}$ as follows:

$$\mathsf{replay}(\delta, \sigma, A)(a) = \begin{cases} \delta(a) & \textbf{if } a \in \mathit{dom}(\delta) \wedge a \notin A \\ \langle v_\sigma \sqcup v_\delta,\ \mathsf{inc}(c_\sigma)\rangle & \textbf{if } a \in \mathit{dom}(\delta) \wedge a \in A \wedge c_\delta = 1 \\ \langle v_\sigma \sqcup v_\delta,\ \infty\rangle & \textbf{if } a \in \mathit{dom}(\delta) \wedge a \in A \wedge c_\delta = \infty \end{cases}$$

where $\langle v_\sigma, c_\sigma\rangle = \sigma(a)$ and $\langle v_\delta, c_\delta\rangle = \delta(a)$. Note that replaying a delta store preserves the domain it is defined over (i.e., $\mathit{dom}(\mathsf{replay}(\delta, \sigma, A)) = \mathit{dom}(\delta)$).

Function replay takes three arguments: the delta store δ to be replayed, the original store σ (on which δ is replayed), and a set A of all addresses in δ that were *allocated* (i.e., representing (abstractions of) new addresses) during the computation of δ. The case where an address a is not in A, but is present in δ implies that the change represents an update to an existing binding (since no new binding for a was allocated). That is, address a was not collected, and therefore replay does not need to restore the original binding of a in σ (as we know that this binding was still present when the update happened). In fact, preserving $\delta(a)$ *as is* is important to maintain strong updates. In the case where address a in δ may come from a new allocation (i.e., $a \in A$), we join the old value in $\sigma(a)$ with the updated one in $\delta(a)$ (as extend would have done for a new allocation on σ). When $c_\delta = 1$, we know that a was only allocated once, and hence we increase the abstract count in σ by 1 (using inc). When $c_\delta = \infty$, we know that a may have been allocated multiple times, so the abstract count becomes ∞.

4.2 Delta Store Semantics with Abstract GC

We now modify the big-step evaluation relation ($\Downarrow^{\Delta}$) to integrate abstract GC. In order to do so, we need to keep track of which addresses in δ may have been *allocated* and which may have been *updated*. Keeping track of a set of allocated addresses A is necessary to support the replay operation (cf. sup.). Keeping track of a set of updated addresses U is necessary to correctly collect garbage in the resulting delta store, which needs to maintain all bindings reachable from the resulting value *and* all existing bindings that were updated during the evaluation (in order to ensure that side effects that happened during evaluation are not lost). We add these two sets of addresses directly to the result of the evaluation:

$$\mathit{res} \in \mathsf{Result} = \mathsf{Val} \times \mathsf{Delta} \times \boxed{\mathcal{P}(\mathsf{Addr})} \times \boxed{\mathcal{P}(\mathsf{Addr})}$$

Next, we adapt the evaluation rules to integrate abstract GC as follows:

– For each evaluation rule, we trivially construct both A and U as the set of addresses allocated (resp. updated) directly in that rule, combined with those allocated (resp. updated) in recursive evaluation steps.

– In the E-Let rule (the only evaluation rule for λ_{ANF} with a non-tail recursive evaluation step), we need to ensure that the delta store computed by the first recursive evaluation step is replayed, so that the bindings needed for the second recursive evaluation step are restored. All other recursive evaluation steps in the evaluation rules do not have an additional (implicit) continuation for which bindings need to be preserved, and therefore do not require their delta store to be replayed (doing so would unnecessarily add imprecision).

– Analogous to the integration of abstract GC into ($\Downarrow$) (cf. Section 2.2), we add an evaluation relation ($\Downarrow_{\Gamma}^{\Delta}$) with the following E-GC rule.

$$\frac{\begin{array}{c}\sigma_{\mathsf{gc}} = \sigma|_{\mathcal{R}_{(\rightsquigarrow_{\sigma})}(\mathcal{T}_{\mathsf{Env}}(\rho))} \\ \langle e, \rho, \sigma_{\mathsf{gc}}\rangle \Downarrow^{\Delta} \langle v, \delta, A, U\rangle \\ \delta_{\mathsf{gc}} = \delta|_{\mathcal{R}_{(\rightsquigarrow_{\delta \triangleright \sigma_{\mathsf{gc}}})}(\mathcal{T}_{\mathsf{Val}}(v)\, \cup\, U_{\mathsf{gc}})} \\ A_{\mathsf{gc}} = A \cap \mathit{dom}(\delta_{\mathsf{gc}}) \qquad U_{\mathsf{gc}} = \{a \in U \mid \sigma_{\mathsf{gc}}(a) \neq \bot\}\end{array}}{\langle e, \rho, \sigma\rangle \Downarrow_{\Gamma}^{\Delta} \langle v, \delta_{\mathsf{gc}}, A_{\mathsf{gc}}, U_{\mathsf{gc}}\rangle} \quad \text{(E-GC)}$$

That is, rule E-GC collects garbage in both the input store σ as well as the output delta store δ (computing reachable addresses over $\delta \triangleright \sigma_{\mathsf{gc}}$, as bindings in the delta store δ may reference those in σ_{gc} and vice versa). It also restricts the sets of addresses A and U: A can be restricted by only keeping addresses that are actually in δ (removing garbage-collected allocations), and U can be restricted by only keeping addresses that have an existing binding in the store σ_{gc}. The crucial difference with the E-GC rule of ($\Downarrow_{\Gamma}$) is that the stores are garbage collected using only the references from the environment (for σ) and the returned/updated values as the GC root set (for δ). References from the continuation, previously kept alive using ψ, now no longer need to be taken into account, leading to more garbage being collected and therefore amplifying the benefits of abstract GC. We subsequently replace ($\Downarrow^{\Delta}$) with ($\Downarrow_{\Gamma}^{\Delta}$) for all recursive evaluation steps, so that GC is applied at every step.

The resulting evaluation rules for $(\Downarrow^{\Delta})$ are given below:

$$\frac{v = \mathcal{A}(\mathrm{ae}, \rho, \sigma)}{\langle \mathrm{ae}, \rho, \sigma \rangle \Downarrow^{\Delta} \langle v, [], \emptyset, \emptyset \rangle} \quad (\text{E-Atom})$$

$$\frac{\begin{array}{c} \langle \lambda x.e, \rho' \rangle \in \mathcal{A}(f, \rho, \sigma) \qquad a_x = \mathsf{alloc}(x) \qquad v_x = \mathcal{A}(ae, \rho, \sigma) \\ \delta_x = \mathsf{extend}(\sigma, a_x, v_x) \qquad \langle e, \rho'[x \mapsto a_x], \delta_x \rhd \sigma \rangle \Downarrow^{\Delta}_{\Gamma} \langle v, \delta, A, U \rangle \end{array}}{\langle f(ae), \rho, \sigma \rangle \Downarrow^{\Delta} \langle v, \delta \rhd \delta_x, \{a_x\} \cup A, U \rangle} \quad (\text{E-App})$$

$$\frac{\begin{array}{c} \langle e_1, \rho, \sigma \rangle \Downarrow^{\Delta}_{\Gamma} \langle v_x, \delta_1, A_1, U_1 \rangle \qquad \delta_1' = \mathsf{replay}(\delta_1, \sigma, A_1) \\ \sigma' = \delta_1' \rhd \sigma \qquad a_x = \mathsf{alloc}(x) \qquad \delta_x = \mathsf{extend}(\sigma', a_x, v_x) \\ \langle e_2, \rho[x \mapsto a_x], \delta_x \rhd \sigma' \rangle \Downarrow^{\Delta}_{\Gamma} \langle v, \delta_2, A_2, U_2 \rangle \end{array}}{\langle \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, \rho, \sigma \rangle \Downarrow^{\Delta} \langle v, \delta_2 \rhd \delta_x \rhd \delta_1', A_1 \cup \{a_x\} \cup A_2, U_1 \cup U_2 \rangle} \quad (\text{E-Let})$$

$$\frac{\begin{array}{c} a_x = \rho(x) \qquad v_x = \mathcal{A}(ae, \rho, \sigma) \\ \delta_x = \mathsf{update}(\sigma, a_x, v_x) \qquad \langle e, \rho, \delta_x \rhd \sigma \rangle \Downarrow^{\Delta}_{\Gamma} \langle v, \delta, A, U \rangle \end{array}}{\langle \mathbf{set}\ x := ae\ \mathbf{then}\ e, \rho, \sigma \rangle \Downarrow^{\Delta} \langle v, \delta \rhd \delta_x, A, \{a_x\} \cup U \rangle} \quad (\text{E-Set})$$

4.3 Comparison to Traditional Abstract GC

It is interesting to compare the impact of our formulation of abstract GC for DSS with the impact of existing "traditional" abstract GC, as pioneered by Might et al. for small-step abstract interpreters [21]. Intuitively, we have already established that abstract GC in $(\Downarrow^{\Delta}_{\Gamma})$ (cf. Section 4.2) can be more efficient than the abstract GC integrated into $(\Downarrow_{\Gamma})$ (cf. Section 2.2): being able to omit the continuation references ψ results in a smaller GC root set, leading to more garbage being collected, therefore amplifying the beneficial effects of abstract GC. We now show that the same reasoning also holds for a comparison with abstract GC in the small-step semantics (as presented in Section 2.1). That is, because of the improved efficiency of *stackless* abstract GC, the result obtained through a big-step evaluation using $(\Downarrow^{\Delta}_{\Gamma})$ is guaranteed to be at least as precise as the corresponding result obtained by a small-step abstract interpreter using $(\rightarrow_{\Gamma})$.

Theorem 2 states this claim more formally:

Theorem 2. $\forall e, \rho, \sigma, v, \delta, A, U, \sigma_k, a_k : \langle e, \rho, \sigma \rangle \Downarrow^{\Delta}_{\Gamma} \langle v, \delta, A, U \rangle$
$\rightarrow \exists v', \sigma', {\sigma_k}' : \langle \boldsymbol{ev}(e, \rho), \sigma, \sigma_k, a_k \rangle \overset{*}{\rightarrow}_{\Gamma} \langle \boldsymbol{ap}(v'), \sigma', \sigma_k', a_k \rangle \wedge v \sqsubseteq v'$

That is, when an expression e evaluates to v (using an environment ρ and store σ), there exists at least one sequence of evaluation steps for the small-step abstract interpreter that evaluates the same expression e to a less precise (or equally precise) value v'. We can match the small-step state ς evaluating e with the state ς' holding its result v' by ensuring that $a_{k_\varsigma} = a_{k_{\varsigma'}}$ (i.e., when the abstract interpreter reaches the same continuation a_k again with a return value). A high-level sketch for the proof of Theorem 2 is given in the appendix, while the full proof (using the Rocq theorem prover) is part of the replication package.

5 Integrating Flow-Sensitive Store Widening

We now formulate a new form of store widening for the big-step semantics using delta stores. Without any form of store widening, a fixpoint computation over the evaluation relation of Section 4 might be expensive, or even in some cases not computable. The store widening we propose is similar to the traditional notion of flow sensitivity: it solves the scalability issues, while still allowing strong updates and a limited form of abstract GC. Moreover, it can be efficiently formulated for DSS, as we have previously shown that delta stores are also joinable (which is efficient because of their minimal representation).

Section 5.1 explains why store widening is necessary to ensure that program semantics remain computable. Next, Section 5.2 shows how the DSS evaluation relation can be modified to integrate such (flow-sensitive) store widening.

5.1 The Need for Store Widening

The program semantics for λ_{ANF} expressed in terms of the evaluation relation $(\Downarrow^{\Delta}_{\Gamma})$ (cf. Section 4) are always computable. The reason for this is that one can easily show the relation $(\Downarrow^{\Delta}_{\Gamma})$ to be finite, since both Config and Result are also finite. However, the relation can still grow very large, as the size of Store alone is in the order of $\mathcal{O}(|\mathsf{Val}|^{|\mathsf{Addr}|})$, potentially leading to many different input configurations that need to be evaluated. Even worse, in an extension of λ_{ANF} (for instance with numerical abstract domains), we may want to choose a set Val that is not necessarily finite. Clearly, in this case, the relation $(\Downarrow^{\Delta}_{\Gamma})$ would also no longer be finite, and therefore potentially not computable (preventing decidability).

For instance, a common abstraction for numerical domains is a *constant propagation lattice*, for which the Hasse diagram is shown in Figure 1. Such an abstract domain is no longer finite, but adheres to the *ascending chain condition* (ACC). The ACC states that every weakly ascending sequence of values (i.e., a sequence $(v_n)_{n\in\mathbb{N}}$ where $v_i \sqsubseteq v_{i+1}$) eventually converges to a stable value (i.e., $\exists n : \forall k \in \mathbb{N} : v_n = v_{n+k}$). Intuitively, such a lattice has a finite "height", so that we can only "go up" in the lattice a finite number of steps. It is reasonable to expect an analysis to be decidable when the abstract domain Val adheres to the ACC, even if it is infinite.

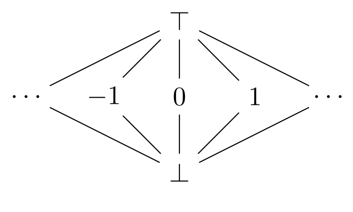

Fig. 1. Hasse diagram of a numerical constant propagation lattice.

Problem: Abstract GC Inhibits Convergence. Without abstract GC (and in the absence of strong updates), $\mathsf{eval}(e)$ would always be computable for any program e when Val adheres to the ACC (even when Val is otherwise infinite). The reason is that the store can only grow over a single execution trace (we say the sequence of stores is *monotonically increasing*), therefore eventually converging (as all values in the store converge, as guaranteed by the ACC). With abstract GC (and strong updates), this no longer holds. Bindings in the store can be removed (or strongly updated to a value that does not subsume the original one), and as such there is no guarantee of convergence in every trace. This requires us to make sure Val is finite in order to render $\mathsf{eval}(e)$ computable for any e.

We can easily illustrate this using the following Scheme program:

```
(letrec ((f (lambda (n) (f (+ n 1))))) (f 0))
```

We assume an abstract interpreter using a constant propagation lattice for the numerical domain (as depicted in Figure 1) and a context-insensitive allocation policy (again writing `@n` for the address of variable `n`). The first call to `f` binds `@n` to `0`. Without abstract GC, the next recursive call to `f` would extend that binding by joining the existing value with `1`, resulting in a configuration where `@n` is bound to $\top$. Subsequent recursive steps would end up in the same configuration, therefore allowing the fixpoint algorithm to terminate. With abstract GC, the old binding of `@n` could be garbage collected before the store is extended for the next recursive call. As such, the second call would end up in a configuration where `@n` is bound to `2`, the third one with `@n` bound to `3` and so on, resulting in an infinite number of configurations to evaluate.

Solution: Store Widening Recovers Convergence. The classical solution to ensure convergence for such domains is to introduce store widening [30]. Using store widening, stores of different configurations are joined together. Since a store is a finite mapping from addresses to values of the abstract domain Val, the ACC for Val implies the ACC for Store. Therefore, under the ACC, an infinite sequence of stores $\sigma_0, \sigma_1, ...$ from different congurations can be turned into a monotonically increasing sequence of stores $(\sigma'_n)_{n\in\mathbb{N}}$ where $\sigma'_0 = \sigma_0$ and $\sigma'_{i+1} = \sigma'_i \sqcup \sigma_{i+1}$, which is guaranteed to converge (to the stable value $\bigsqcup_{n\in\mathbb{N}} \sigma_i$) as $\sigma'_i \sqsubseteq \sigma'_{i+1}$ by design.

Traditionally, *global* store widening has been applied to both small-step (AAM-based) [30] and big-step (definitional) [2] abstract interpreters. Under global store widening, the stores of all configurations are joined in a single store. This results in a flow-insensitive analysis, as the abstract interpreter does not discern different stores at different program points. Unfortunately, global store widening renders both abstract GC and abstract counting completely useless: abstract GC is unable to collect any garbage (since it needs to keep everything that is reachable from any configuration/state), whereas strong updates are never applicable (since all abstract counts eventually become ∞ when the global store converges).

We can, however, choose a middle ground between purely local stores and global store widening. The store widening we propose for DSS in this section joins stores of configurations at the same program point. That is, the analysis keeps track of a single store per program point (being the join of all stores encountered at that program point). This corresponds to a flow-sensitive analysis. While less precise than local stores, it is known that flow sensitivity still allows for strong updates, and as we discuss in Section 5.2, a limited form of abstract GC.

5.2 Delta Store Semantics with Flow-Sensitive Store Widening

We now show how this form of flow-sensitive store widening can be integrated into the delta store semantics with abstract GC of Section 4. Doing so should ensure that the evaluation relation ($\Downarrow^{\Delta}_{\Gamma}$) becomes finite (and therefore computable) whenever Val adheres to the ACC. This requires the following two modifications:

- We no longer keep the store directly as part of each configuration. That is, instead of having multiple configurations $\langle e, \rho, \sigma_i \rangle$ (i.e., with the same expression e and environment ρ but different stores σ_i), we now only have a single configuration $\langle e, \rho \rangle$, which gets associated with a single store $\bigsqcup_i \sigma_i$. As such, the definition of Config is updated as follows:

  $$\mathit{conf} \in \mathsf{Config} = \mathsf{Exp} \times \mathsf{Env}$$

  For a context-insensitive analysis, such a configuration effectively corresponds to a single program point[9]. To associate a single store with each configuration, we make use of a map $\Xi_{e_0} : \mathsf{Config} \rightarrow \mathsf{Store}$. The store $\Xi_{e_0}(\mathit{conf})$ over-approximates the join of all stores σ_i that can occur at the configuration conf (i.e., that previously were part of that configuration), where e_0 is the initial expression of the program[10]. We first define a relation $(\rightsquigarrow)_{\mathit{conf}} \subseteq \mathsf{Store} \times \mathsf{Config}$, where intuitively $\sigma \rightsquigarrow_{\mathit{conf}} \mathit{conf}'$ implies that σ potentially "flows to" conf' during the evaluation of the configuration conf. We can then define $\Xi(\mathit{conf})$ as the join of all (garbage-collected[11]) stores that flow to conf:

  $$\Xi(\langle e, \rho \rangle) = \bigsqcup \{ \sigma|_{\mathcal{R}_{(\rightsquigarrow_\sigma)}(\mathcal{T}_{\mathsf{Env}}(\rho))} \mid \sigma \rightsquigarrow_{\langle e_0, [] \rangle} \langle e, \rho \rangle \}$$

- We ensure that the evaluation relation ($\Downarrow^{\Delta}$) (and ($\Downarrow^{\Delta}_{\Gamma}$)) become deterministic (i.e., they can be seen as functions $\mathsf{Config} \rightarrow \mathsf{Result}$). This is done by joining the results whenever there are multiple non-deterministic program paths.

Together, this suffices to show that ($\Downarrow^{\Delta}_{\Gamma}$) becomes a finite relation: it is clear that by factoring out the store, Config becomes finite, and when ($\Downarrow^{\Delta}_{\Gamma}$) is deterministic, only a single result $\mathit{res} \in \mathsf{Result}$ can be associated with each configuration. And although $(\rightsquigarrow)_{\mathit{conf}}$ is technically not finite (since an infinite number of stores can flow to a configuration conf when Val is infinite), $\Xi(\mathit{conf})$ is computed as the join of all these stores, which is guaranteed to converge due to the ACC.

[9] Note that for a context-sensitive analysis, the context would be part of Config, and so the store would be shared per program point and context.

[10] For brevity, from this point on we drop the subscript and instead write Ξ for Ξ_{e_0}.

[11] Note that we apply garbage collection before joining the stores, which results in a more precise store compared to doing it the other way around.

We now present the updated evaluation rules for $(\Downarrow^{\Delta})$ and $(\Downarrow^{\Delta}_{\Gamma})$, together with the derivation rules for the relation $(\rightsquigarrow)_{\mathit{conf}}$ (i.e., included in the same rules).

$$\frac{\sigma = \Xi(\mathit{conf}) \qquad v = \mathcal{A}(\mathrm{ae}, \rho, \sigma)}{\underbrace{\langle \mathrm{ae}, \rho\rangle}_{\mathit{conf}} \Downarrow^{\Delta} \langle v, [], \emptyset, \emptyset\rangle} \quad \text{(E-Atom)}$$

$$\frac{\begin{array}{c} \sigma = \Xi(\mathit{conf}) \qquad \langle \lambda x_i.e_i, \rho'_i\rangle \in \mathcal{A}(f, \rho, \sigma) \qquad v_x = \mathcal{A}(\mathrm{ae}, \rho, \sigma) \\ a_i = \mathsf{alloc}(x_i) \qquad \delta_i = \mathsf{extend}(\sigma, a_i, v_x) \qquad \sigma_i = \delta_i \rhd \sigma \\ \underbrace{\langle e_i, \rho'_i[x \mapsto a_i]\rangle}_{\mathit{conf'}} \Downarrow^{\Delta}_{\Gamma} \langle v_i, \delta'_i, A_i, U_i\rangle \qquad \sigma_r \rightsquigarrow_{\mathit{conf'}} \mathit{conf}_r \\ v = \bigsqcup_i v_i \qquad \delta = \bigsqcup_{i\,\sigma} \delta'_i \rhd \delta_i \qquad A = \bigcup_i \{a_i\} \cup A_i \qquad U = \bigcup_i U_i \end{array}}{\sigma' \rightsquigarrow_{\mathit{conf}} \mathit{conf'} \qquad \sigma_r \rightsquigarrow_{\mathit{conf}} \mathit{conf}_r \qquad \underbrace{\langle f(\mathrm{ae}), \rho\rangle}_{\mathit{conf}} \Downarrow^{\Delta} \langle v, \delta, A, U\rangle} \quad \text{(E-App)}$$

$$\frac{\begin{array}{c} \sigma = \Xi(\mathit{conf}) \qquad \overbrace{\langle e_1, \rho\rangle}^{\mathit{conf}_1} \Downarrow^{\Delta}_{\Gamma} \langle v_x, \delta_1, A_1, U_1\rangle \qquad \sigma_{r_1} \rightsquigarrow_{\mathit{conf}_1} \mathit{conf}_{r_1} \\ \delta'_1 = \mathsf{replay}(\delta_1, \sigma, A_1) \qquad \sigma' = \delta'_1 \rhd \sigma \qquad a_x = \mathsf{alloc}(x) \qquad \delta_x = \mathsf{extend}(\sigma', a_x, v_x) \\ \sigma'' = \delta_x \rhd \sigma' \qquad \underbrace{\langle e_2, \rho[x \mapsto a_x]\rangle}_{\mathit{conf}_2} \Downarrow^{\Delta}_{\Gamma} \langle v, \delta_2, A_2, U_2\rangle \qquad \sigma_{r_2} \rightsquigarrow_{\mathit{conf}_2} \mathit{conf}_{r_2} \end{array}}{\begin{array}{c} \sigma \rightsquigarrow_{\mathit{conf}} \mathit{conf}_1 \qquad \sigma'' \rightsquigarrow_{\mathit{conf}} \mathit{conf}_2 \qquad \sigma_{r_1} \rightsquigarrow_{\mathit{conf}} \mathit{conf}_{r_1} \qquad \sigma_{r_2} \rightsquigarrow_{\mathit{conf}} \mathit{conf}_{r_2} \\ \underbrace{\langle \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, \rho\rangle}_{\mathit{conf}} \Downarrow^{\Delta} \langle v, \delta_2 \rhd \delta_x \rhd \delta'_1, A_1 \cup \{a_x\} \cup A_2, U_1 \cup U_2\rangle \end{array}} \quad \text{(E-Let)}$$

$$\frac{\begin{array}{c} \sigma = \Xi(\mathit{conf}) \qquad a_x = \rho(x) \qquad v_x = \mathcal{A}(\mathrm{ae}, \rho, \sigma) \qquad \delta_x = \mathsf{update}(\sigma, a_x, v_x) \\ \sigma' = \delta_x \rhd \sigma \qquad \underbrace{\langle e, \rho\rangle}_{\mathit{conf'}} \Downarrow^{\Delta}_{\Gamma} \langle v, \delta, A, U\rangle \qquad \sigma_r \rightsquigarrow_{\mathit{conf'}} \mathit{conf}_r \end{array}}{\begin{array}{c} \sigma' \rightsquigarrow_{\mathit{conf}} \mathit{conf'} \qquad \sigma_r \rightsquigarrow_{\mathit{conf}} \mathit{conf}_r \\ \underbrace{\langle \mathbf{set}\ x := \mathit{ae}\ \mathbf{then}\ e, \rho\rangle}_{\mathit{conf}} \Downarrow^{\Delta} \langle v, \delta \rhd \delta_x, A, \{a_x\} \cup U\rangle \end{array}} \quad \text{(E-Set)}$$

$$\frac{\begin{array}{c} \sigma_{\mathsf{gc}} = \Xi(\mathit{conf}) \qquad \langle e, \rho\rangle \Downarrow^{\Delta} \langle v, \delta, A\rangle \\ \delta_{\mathsf{gc}} = \delta|_{\mathcal{R}_{(\rightsquigarrow_{\delta \rhd \sigma_{\mathsf{gc}}})}(\mathcal{T}_{\mathsf{Val}}(v)\, \cup\, U_{\mathsf{gc}})} \\ A_{\mathsf{gc}} = A \cap \mathit{dom}(\delta_{\mathsf{gc}}) \qquad U_{\mathsf{gc}} = \{a \in U \mid \sigma_{\mathsf{gc}}(a) \neq \bot\} \end{array}}{\underbrace{\langle e, \rho\rangle}_{\mathit{conf}} \Downarrow^{\Delta}_{\Gamma} \langle v, \delta_{\mathsf{gc}}, A_{\mathsf{gc}}, U_{\mathsf{gc}}\rangle} \quad \text{(E-GC)}$$

The first modification is that since the store is no longer part of a configuration *conf*, we instead retrieve it as $\Xi(\mathit{conf})$. Conversely, for every configuration *conf'* (reachable during the evaluation of *conf*) that normally would have included σ, we now instead derive that $\sigma \rightsquigarrow_{\mathit{conf}} \mathit{conf}'$. These derivations come both directly from the evaluation of the current configuration, as well as from recursive evaluations of other configurations. The second modification is that the E-App rule (the only evaluation rule for λ_{ANF} that faces non-determinism) now joins the results for each possible function that may be called. Doing so makes use of the join operator ($\sqcup_\sigma$) for delta stores to efficiently merge changes computed w.r.t. σ over multiple non-deterministic paths.

Note that computing ($\Downarrow_{\Gamma}^{\Delta}$) requires a fixpoint computation over both the relations ($\Downarrow_{\Gamma}^{\Delta}$) and $(\rightsquigarrow)_{\mathit{conf}}$. We again refer to the cache-based fixpoint algorithm presented by Darais et al. [2], which can be employed *mutatis mutandis* as one possible implementation for such a fixpoint computation.

Impact on Abstract GC. We recall that abstract GC offers three key advantages: (1) it avoids precision losses due to unnecessary joins (with garbage values), (2) it improves abstract counting by resetting abstract counts for collected addresses, and (3) it speeds up the fixpoint computation thanks to garbage irrelevance.

When using flow-sensitive ("per configuration") store widening, the precision improvements of (1) are largely negated. The reason is that addresses are usually identified by the program point they are allocated at, so that all values bound to an address end up joined in the same shared store associated with that program point (regardless of if it was collected before being allocated again[12]). In theory, some precision can still be gained from collecting garbage that arises from updates, since updates (unlike allocations) may occur at different program points.

The main precision benefit of abstract GC with flow-sensitive store widening comes from (2). With abstract GC, the abstract count of a collected address is reset to 0, so that it is increased to 1 when the address is allocated again. Store widening will only join (i.e., not necessarily increase) the abstract counts for all allocations of an address at its corresponding program point. Reclaiming garbage bindings before such allocations can therefore keep the count at 1 instead of ∞, increasing the potential for strong updates (and therefore precision).

Abstract GC is also useful in conjunction with flow-sensitive store widening because of (3). Specifically, garbage irrelevance makes the computation of Ξ more efficient, since the fixpoint iteration joining stores at the same program point together reaches a stable value more quickly when garbage bindings are removed. That is, the iteration does not have to continue until these garbage bindings converge to a stable value as well. In turn, this improves the convergence rate (and reduces memory consumption) for the fixpoint computation.

[12] Note that this assumes that allocations are differentiated using the same context sensitivity as configurations. For less precise allocators (e.g., a monovariant allocator in a context-sensitive analysis) – or when using trace partitioning [18] – the allocation of an address does not always end up widened in the same store. Therefore, it may be beneficial to reclaim previous occurences of that address first, and such analysis designs can gain more precision when combining flow sensitivity with abstract GC.

6 Evaluation

We have already formally shown that DSS with abstract GC always matches or improves upon the precision of traditional GC for a small-step abstract interpreter, using a mechanised proof implemented in the Rocq theorem prover (cf. Section 4.3). Likewise, we have discussed the impact of flow sensitivity for DSS, as well as its interaction with abstract GC (cf. Section 5.2). We now present an accompanying empirical evaluation, using our implementation in the MAF framework.[13] Both our implementation, as well as the mechanised Rocq proof for Theorem 2, are included as part of the replication package for this paper.

Specifically, we aim to answer the following research questions:

RQ1 How much precision is gained using a stackless form of abstract GC (as integrated for DSS in Section 4) compared to traditional abstract GC?
RQ2 What is the impact of flow-sensitive store widening (as discussed in Section 5) on the precision and scalability of DSS?
RQ3 What is the impact of abstract GC for flow-sensitive DSS?

Evaluation Setup. Our implementation in MAF extends the formalisations for λ_{ANF} to support a large subset of R5RS Scheme. As such, we are able to run our experiments using a benchmark suite of 15 Scheme programs: 9 from the Gabriel benchmarking suite [6][14] and 6 from the built-in benchmarking suite of MAF. Table 1 lists each Scheme program along with its size in LOC.

Table 1. Overview of the Scheme programs used as benchmarks, along with lines of code (LOC) for each benchmark. The Gabriel benchmarks are highlighted in **bold**.

Benchmark	LOC	Benchmark	LOC	Benchmark	LOC
boyer	593	**destruc**	65	matrix	648
browse	211	**diviter**	24	mceval	282
cpstak	24	**divrec**	19	regex	80
dderiv	83	**takl**	20	rsa	85
deriv	39	grid	35	tak	11

We compare different abstract interpreters in terms of precision and performance. For precision, we use the built-in precision measurement utilities of the MAF framework. These compare precision by first running the program using a concrete interpreter (multiple times to cover multiple program paths in

[13] Although we have not presented a formal proof for the soundness of DSS, we have validated this empirically using the automated soundness testing in MAF.

[14] We omitted the `ctak` benchmark and `triangl` benchmarks from the original Gabriel benchmarking suite. The `ctak` benchmark was removed due to its use of `call/cc` (which is not supported by the MAF framework, and also would not be trivial to integrate into DSS, since it breaks the context irrelevance of the semantics). The `triangl` benchmark was removed because it timed out for the concrete interpreter.

non-deterministic programs), and subsequently measuring how many values in each abstract interpreter strictly over-approximate (i.e., are less precise than) the corresponding results of the concrete interpreter. A more precise analysis should therefore result in fewer over-approximations compared to a less precise one.

Unlike the minimal formalisations presented in this paper for λ_{ANF} (which only allow for limited allocation policies), our implementations do support various configurations for context sensitivity. We run our experiments using m-CFA [23] context sensitivity (which uses the top m stack frames as context) for varying values of m (where higher values of m may increase precision). For the abstract domain, we employ a constant propagation lattice (as depicted in Figure 1, Section 5.1) as an abstraction for primitive domains in all experiments.

6.1 RQ1: The Precision Benefit of Stackless Abstract GC

Theorem 2 guarantees that the precision of DSS with abstract GC (i.e., using the evaluation relation ($\Downarrow^{\Delta}_{\Gamma}$)) is always the same or better than that of an equivalent small-step abstract interpreter with abstract GC (i.e., using the transition relation ($\rightarrow_{\Gamma}$)). However, it does not guarantee that the precision is *strictly* better, nor does it tell us how much precision improvement can be expected.

In order to evaluate this empirically, we therefore compare precision with an equivalent implementation of AAM with abstract GC and abstract counting (also known as ΓCFA [21]). However, recall that such a small-step abstract interpreter may exhibit decreased precision compared to DSS for two reasons: the lack of stackless abstract GC and the lack of full stack precision (which DSS inherits from its foundation in abstract definitional interpreters). Since we are only interested in measuring the former, we have modified ΓCFA to instead use a fully precise continuation address allocator (specifically, the one from AAC [13]), so that it exhibits the same stack precision as DSS.

Table 2 compares the results for ΓCFA and DSS with abstract GC. Note that for benchmarks that timed out, we can use the partial results of the analysis in order to compute a *lower bound* for the number of over-approximations (as the result would only grow more imprecise as the analysis continues).

Comparison of Precision. The results confirm the claim stated by Theorem 2: in all benchmarks where both ΓCFA and DSS terminate, we have an equal (`cpstak`, `regex`) or lower (`grid`, `rsa`, `tak`) number of over-approximations for DSS. The context-insensitive (i.e., $m = 0$) analyses of the `regex` benchmark also show significant precision improvements for DSS: the analysis with ΓCFA times out with *at least* 34 over-approximations, and requires increased context sensitivity (i.e., $m \geq 1$) in order to achieve the same precision as a context-insensitive analysis with DSS. The improvements also hold when context sensitivity is increased for both abstract interpreters: for the `dderiv` benchmark with $m = 2$, ΓCFA has at least 11 over-approximations compared to exactly 7 in DSS. As we have modified ΓCFA with a fully precise continuation allocator, we can attribute these improvements to the stackless nature of abstract GC in DSS, which can reclaim more garbage compared to the abstract GC of ΓCFA.

Table 2. Comparison of the number of strict over-approximations (lower is better) and time taken between small-step ΓCFA and big-step DSS with (stackless) abstract GC. A time of ∞ indicates that the benchmark exceeded the time limit of 10 minutes; in this case, we report a lower bound for the number of over-approximations.

	$m = 0$				$m = 1$				$m = 2$			
	ΓCFA		DSS		ΓCFA		DSS		ΓCFA		DSS	
	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time
boyer	≥0	∞	≥0	∞	≥0	∞	≥0	∞	≥0	∞	≥0	∞
browse	≥5	∞	≥7	∞	≥0	∞	≥4	∞	≥7	∞	≥4	∞
cpstak	3	50ms	3	26ms	3	11s	3	5s	≥3	∞	≥0	∞
dderiv	≥28	∞	38	20s	≥7	∞	7	1s	≥11	∞	7	1s
deriv	≥3	∞	3	1s	≥3	∞	3	1s	≥4	∞	3	1s
destruc	≥8	∞	≥10	∞	≥8	∞	≥10	∞	≥3	∞	≥3	∞
diviter	≥4	∞	4	3s	≥0	∞	4	3s	≥0	∞	2	4s
divrec	≥4	∞	4	3s	≥0	∞	3	3s	≥0	∞	3	3s
takl	≥6	∞	6	1s	≥3	∞	6	9s	≥3	∞	6	4m48s
grid	10	1m34s	7	200ms	≥2	∞	7	167ms	≥2	∞	7	390ms
matrix	≥10	∞	≥10	∞	≥9	∞	≥10	∞	≥9	∞	≥10	∞
mceval	≥3	∞	≥3	∞	≥3	∞	≥3	∞	≥12	∞	13	36s
regex	≥34	∞	0	92ms	0	293ms	0	95ms	0	275ms	0	100ms
rsa	14	923ms	7	75ms	14	58s	7	92ms	≥14	∞	7	78ms
tak	2	20s	0	2s	≥0	∞	0	1s	≥0	∞	0	1s

We have also compared precision to the big-step abstract interpreter with abstract GC of Section 2.2, which is more similar to DSS but keeps track of a set of continuation addresses ψ. Our experiments show identical[15] precision results for this interpreter as for ΓCFA (therefore omitted here for brevity). This confirms that the precision improvements for DSS in Table 2 indeed stem from being able to omit ψ from the GC root set (i.e., rendering the GC "stackless").

Comparison of Performance. Both ΓCFA and DSS time out for a significant number of benchmarks, showing poor scalability for both abstract interpreters. Timeouts are more frequent for ΓCFA. This can partially be explained by its use of the stack-precise AAC continuation allocator [13], which for the context-insensitive case (i.e., $m = 0$) is known to raise analysis complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^8)$. Further performance improvements can also in part be explained by the increased precision of DSS: as precision increases, the abstract interpreter spends less time having to explore *spurious program paths* (i.e., execution paths that only exist due to excessive over-approximation of the program's control flow behaviour). This effect can also be observed for the same abstract interpreter by increasing context sensitivity (e.g., for the `mceval` benchmark with DSS).

6.2 RQ2: The Impact of Flow-Sensitive Store Widening

We now evaluate the impact of flow-sensitive store widening on both precision and scalability. For this purpose, we run the same experiments again, this time

[15] expect for the `rsa` benchmark due to a known, unrelated implementation difference.

comparing a version of DSS without store widening (i.e., the $(\Downarrow^{A}_{\Gamma})$ evaluation relation of Section 4) and a version of DSS with flow-sensitive store widening (i.e., using $(\Downarrow^{A}_{\Gamma})$ as presented in Section 5). The results are shown in Table 3.

Table 3. Comparison of the number of strict over-approximations (lower is better) and time taken between DSS with and without flow-sensitive store widening (DSS-FS and DSS, resp.). A time of ∞ indicates that the benchmark exceeded the time limit of 10 minutes; in this case, we report a lower bound for the number of over-approximations.

	$m = 0$				$m = 1$				$m = 2$			
	DSS		DSS-FS		DSS		DSS-FS		DSS		DSS-FS	
	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time
boyer	$\geq$0	∞	1251	9m46s	$\geq$0	∞	$\geq$1236	∞	$\geq$0	∞	$\geq$48	∞
browse	$\geq$7	∞	91	8s	$\geq$4	∞	88	22s	$\geq$4	∞	$\geq$80	∞
cpstak	3	26ms	3	9ms	3	5s	3	72ms	$\geq$0	∞	3	222ms
dderiv	38	20s	48	905ms	7	1s	46	2s	7	1s	38	5s
deriv	3	1s	29	517ms	3	1s	9	451ms	3	1s	8	787ms
destruc	$\geq$10	∞	15	610ms	$\geq$10	∞	15	738ms	$\geq$3	∞	13	1s
diviter	4	3s	6	1s	4	3s	6	2s	2	4s	4	2s
divrec	4	3s	4	2s	3	3s	4	2s	3	3s	4	2s
takl	6	1s	6	60ms	6	9s	6	143ms	6	4m48s	6	710ms
grid	7	200ms	12	106ms	7	167ms	12	212ms	7	390ms	10	237ms
matrix	$\geq$10	∞	180	8s	$\geq$10	∞	129	18s	$\geq$10	∞	128	1m27s
mceval	$\geq$3	∞	159	19s	$\geq$3	∞	157	2m02s	13	36s	$\geq$157	∞
regex	0	92ms	48	547ms	0	95ms	48	3s	0	100ms	48	7s
rsa	7	75ms	18	54ms	7	92ms	17	74ms	7	78ms	17	96ms
tak	0	2s	2	5ms	0	1s	2	19ms	0	1s	2	120ms

Comparison of Precision. It is clear that store widening, even when applied per configuration, still greatly decreases precision compared to an equivalent abstract interpreter without any store widening. Across all benchmarks, DSS-FS has significantly more over-approximations compared to DSS. A key factor explaining these differences is that abstract GC no longer offers the same precision for DSS-FS as it did for DSS (we explore this further in RQ3). Instead, the analysis is now more reliant on increased context sensitivity in order to improve its precision.

Comparison of Performance. The key benefit of applying store widening is that it greatly improves the performance of the abstract interpreter (and therefore its scalability towards larger programs such as `boyer`, `browse` and `mceval`). Indeed, without widening, many programs fail to terminate for DSS, either because of an exponential explosion in the number of configurations to evaluate or because values in its (infinite) domain can never converge. In contrast, for DSS-FS, we only observe 4 timeouts in total, and in fact none for the context-insensitive variant (i.e, where $m = 0$). In other benchmarks (e.g, `takl` with $m = 2$), performance is improved by orders of magnitude. While the precision benefits of DSS without store widening are appealing, we therefore argue that *some* store widening (such as flow-sensitive store widening) is a must to analyze larger, real-world programs.

6.3 RQ3: The Impact of Abstract GC on Flow-Sensitive DSS

As discussed in Section 5.2, store widening limits the precision benefits of abstract GC. We can, however, expect it to have a positive impact on the performance of the analysis by reducing the number of iterations that are required for each flow-sensitive store to converge. We evaluate the impact of abstract GC in the setting of flow-sensitive store widening by comparing two versions of DSS-FS: one with and one without abstract GC. Table 4 shows the results.

Table 4. Comparison of the number of strict over-approximations (lower is better) and time taken between DSS-FS with and without abstract GC. A time of ∞ indicates that the benchmark exceeded the time limit of 10 minutes; in this case, we report a lower bound for the number of over-approximations.

	$m = 0$				$m = 1$				$m = 2$			
	with GC		without GC		with GC		without GC		with GC		without GC	
	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time
boyer	1251	9m46s	1251	3m12s	≥1236	∞	1249	6m13s	≥48	∞	48	1m36s
browse	91	8s	98	2s	88	22s	88	8s	≥80	∞	80	7m33s
cpstak	3	9ms	3	14ms	3	72ms	3	50ms	3	222ms	3	129ms
dderiv	48	905ms	48	379ms	46	2s	46	1s	38	5s	38	4s
deriv	29	517ms	29	223ms	9	451ms	9	288ms	8	787ms	8	453ms
destruc	15	610ms	15	449ms	15	738ms	15	743ms	13	1s	13	1s
diviter	6	1s	6	2s	6	2s	6	1s	4	2s	4	1s
divrec	4	2s	4	2s	4	2s	4	1s	4	2s	4	1s
takl	6	60ms	6	45ms	6	143ms	6	100ms	6	710ms	6	908ms
grid	12	106ms	12	185ms	12	212ms	12	301ms	10	237ms	10	374ms
matrix	180	8s	181	7s	129	18s	129	32s	128	1m27s	128	3m50s
mceval	159	19s	159	7s	157	2m02s	157	1m27s	≥157	∞	≥151	∞
regex	48	547ms	48	387ms	48	3s	48	3s	48	7s	48	12s
rsa	18	54ms	18	112ms	17	74ms	17	90ms	17	96ms	17	158ms
tak	2	5ms	2	6ms	2	19ms	2	28ms	2	120ms	2	323ms

Comparison of Precision. As expected, abstract GC has very little impact on precision when employing flow-sensitive store widening. The only benchmarks that show some minor precision improvement are `browse` and `matrix`. The reasons for these limited precision improvements are discussed in Section 5.2. In summary, abstract GC's main precision benefit comes from updates (i.e., updates to mutable variables and mutable data structures). However, these may not occur frequently in our benchmarking suite: although Scheme is technically an imperative language, in practice it often encourages a more functional style, avoiding such side-effecting mutations. The benefits on DSS-FS with abstract GC may therefore be more pronounced for languages with programming styles that make frequent use of (field) assignments (such as Python or JavaScript).

Comparison of Performance. Both configurations achieve comparable performance results, with some benchmarks (such as `boyer`) showing a clear edge for

DSS-FS without GC, and others (such as `matrix`) exhibiting better performance using DSS-FS with GC. The reason is that the integration of abstract GC impacts performance both positively and negatively.

The negative impact comes from the overhead that is associated with GC. In our implementation specifically, we use a tracing stop-and-copy GC applied at every evaluation step (in order to maximize the precision benefits); it is known that such a policy can potentially slow down an abstract interpreter by one or two orders of magnitude [29], and is therefore not recommended in practice.

Despite this significant overhead, the performance still holds up well compared to DSS-FS without abstract GC. This can be attributed to the positive impact of abstract GC on performance, as the stores at each program point converge faster without garbage. To measure this benefit without the technical overhead of abstract GC, we conducted a separate experiment measuring the number of iteration steps required for the fixpoint computation of the analysis. For instance, the `matrix` benchmark only requires 2448 iterations with GC compared to 3098 without, explaining its performance improvement. On average, for our experiments abstract GC decreases the number of iterations required by 27,8%.

7 Related Work

The benefits of abstract GC for the analysis of higher-order languages have long been recognised: early work by Jagannathan et al. [16] proposes a primitive (albeit inefficient) form of abstract GC in conjunction with abstract counting in order to improve the precision of must-alias analyses. Might et al. later pioneered the use of abstract GC (and abstract counting), as presented in this work, for small-step abstract interpreters constructed using the AAM approach [30] to abstract interpretation, referring to the resulting analysis as ΓCFA [21]. Their work shows that abstract GC offered order-of-magnitude improvements to both the precision and performance of abstract interpreters in this setting.

Later work made several efforts to integrate abstract GC into abstract interpreters that enjoy the benefits of pushdown control flow, such as the big-step abstract definitional interpreters of Darais et al. [2]. Abstract definitional interpreters are also the foundation of DSS. We have discussed the original integration of abstract GC into abstract definitional interpreters, and the shortcomings of this integration, in Section 2.2. Likewise, Earl et al. [4] show how abstract GC can be integrated into a pushdown flow analysis (aiming to reap the benefits of both). They formulate their approach as *stack introspection*: for every control point they approximate the set of continuation frames that could be on the stack, using their references as part of the root set to collect garbage. Similar to the suboptimal integration of abstract GC into abstract definitional interpreters, the resulting analysis no longer enjoys the context irrelevance of the original pushdown analysis, and does not achieve the same potential in precision improvements of abstract GC as the stackless form of abstract GC for DSS.

CFA2 [31] combines full stack precision with some form of abstract GC. It separates bindings on the stack from those on the heap. Stack bindings are automatically "garbage collected" as stack frames are popped; this is similar to how DSS can garbage collect (unreferenced) local variables (regardless of whether they are already allocated for the continuation) in a delta store upon returning. However, in CFA2, bindings may also escape to the heap, which is not GC'd.

The closest to our own work is the *heap fragment* semantics (HFAC) of Germane et al. [8,9]. Similar to how delta stores in DSS represent changes w.r.t. an original store, a heap fragment in HFAC only captures bindings relevant for the current evaluation. To our knowledge, HFAC is the only other existing formalism exhibiting all advantages of pushdown control flow and abstract GC, as listed in Section 1.1. We improve upon the work of [8] by adding support for abstract counting and collecting garbage in delta stores (whereas the technique presented in [8] only collected garbage for the input heap fragment). The authors of [9] point out the "formal weight" of HFAC, which significantly complicates the language semantics. Compared to their work, we present a novel and simpler formalism, capturing the essence of the mechanism that allows the integration of abstract GC. We also show how delta stores can be joined, and extend the formalism to incorporate a novel form of store widening that preserves flow sensitivity.

Monat et al. [24] also integrate abstract GC into a flow-sensitive analysis for Python (in the MOPSA framework). In contrast to our own work, their abstract garbage collector still includes continuation references as part of the GC root set, and therefore does not benefit from the precision improvements of "stackless" abstract GC. Similar to our own findings on RQ3 (cf. Section 6.3), they also report limited precision benefits when flow-sensitive widening is used, and argue that abstract GC is mainly useful to improve analysis performance and memory consumption. They also point out that abstract GC *may* improve precision due to its interaction with recency abstraction [1], similar to how we argued in Section 5 that it may improve precision due to its interaction with abstract counting.

We implemented abstract GC using a tracing stop-and-copy collector that is interleaved at every evaluation step. As observed by the results for RQ3 (cf. Section 6.3), this adds severe GC overhead (which for DSS-FS negates the performance benefits of abstract GC). Other frameworks [17,21,20] apply abstract GC less frequently to tame this overhead. Van Es et al. [29] propose replacing tracing abstract GC with abstract reference counting, which is automatically applied at every step without significant overhead. We leave the integration of such abstract reference counting into DSS open as future work.

The flow-sensitive store widening of Section 5 joins stores per control location (and context). Trace partitioning [18,25] can generalize this technique by using abstract traces to keep multiple stores at the same control location (but with different execution traces reaching that control location) separate. Keeping more stores separate improves precision, and may also increase the precision benefits of abstract GC (compared to what we observed in RQ3) as different values allocated at the same address may be joined in different stores (therefore making it useful to reclaim previous bindings at that address using abstract GC).

Germane et al. [7] distinguish between three different kinds of "control-flow sensitivity", so that an analysis is either *path-sensitive*, *flow-sensitive* or *flow-*

insensitive. The second indeed corresponds to our own interpretation of flow sensitivity, as formulated for DSS with store widening in Section 5 (whereas the semantics of Sections 2, 3 and 4 would be labelled as *path-sensitive*). Most existing abstract interpreters, based on AAM or abstract definitional interpreters, only consider using either entirely local stores (resulting in a path-sensitive analysis) or *globally* widened stores [30,12,2] (resulting in a flow-insensitive analysis) instead. Flow sensitivity has so far been more common in traditional data-flow analyses [14,26,15]. Interestingly, Oh et al. [27] suggest an *adaptive* approach to control-flow sensitivity, where different addresses are treated with different sensitivities (in their work, only handling them either flow-sensitively or flow-insensitively). For our own experiments, it is clear that the precision benefits of path sensitivity in combination with abstract GC are substantial, but impede analysis scalability when applied to all addresses. Applying their learning strategy for choosing between both control-flow sensitivities adaptively for each address could potentially bring together the benefits of both DSS and DSS-FS.

8 Conclusion

In this work, we have presented delta store semantics (DSS), a novel formulation for big-step abstract definitional interpreters where evaluation steps return a delta store to capture all changes made to the input store. Unlike regular stores, delta stores are more minimal (representing only changes), reusable (being able to be applied to or *replayed* for a (larger) store), composable and efficiently joinable. We have shown how these delta stores enable the integration of both abstract GC and a flow-sensitive variant of store widening into DSS.

When using DSS with abstract GC – without store widening – we not only achieve the full advantages of pushdown control flow and abstract GC, but also unlock an additional synergy between both that further increases the benefits of abstract GC. Specifically, we have shown both formally (using the Rocq theorem prover) and empirically (using our implementation in MAF) that this combination outperforms a small-step interpreter with abstract GC.

When using DSS with flow-sensitive store widening, our experiments confirm that DSS no longer faces the scalability issues that are inherent to the usage of local (path-sensitive) stores. Unlike global (flow-insensitive) store widening, we have shown that the resulting abstract interpreter still supports abstract counting (i.e., strong updates) and a limited form of abstract GC.

Acknowledgments. This work was partially supported by an Innoviris grant for the ECOPIPE project (Joint R&D 2022, grant number 2022-PS-JRDIC-8) and by the Research Foundation Flanders (FWO) (grant number 1187122N).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Appendix

We can prove Theorem 2 using mutual induction on ($\Downarrow^{\Delta}$) and ($\Downarrow^{\Delta}_{\Gamma}$). Doing so, however, first requires generalising this theorem in order to strengthen the induction hypothesis, resulting in Theorem 3.

Theorem 3. $\forall e, \rho, \sigma, v, \delta, A, U, \varsigma = \langle \boldsymbol{ev}(e, \rho), \sigma_{\varsigma}, \sigma_{k_{\varsigma}}, a_k \rangle,$
$\mathcal{R}_{\rho} = \mathcal{R}_{(\rightsquigarrow_{\sigma})}(\mathcal{T}_{\mathsf{Env}}(\rho)), \sigma_{\mathsf{gc}} = \sigma|_{\mathcal{R}_{\rho}}, \mathcal{R}_v = \mathcal{R}_{(\rightsquigarrow_{\delta \triangleright \sigma_{\mathsf{gc}}})}(\mathcal{T}_{\mathsf{Val}}(v)) :$
$\langle e, \rho, \sigma \rangle \Downarrow^{\Delta}_{\Gamma} \langle v, \delta, A, U \rangle \wedge (\sigma_{\mathsf{gc}} \sqsubseteq \sigma_{\varsigma}) \wedge (\Gamma(\varsigma) = \varsigma) \rightarrow \exists \varsigma' = \langle \boldsymbol{ap}(v'), \sigma_{\varsigma'}, \sigma_{k_{\varsigma'}}, a_k \rangle :$
$\varsigma \overset{*}{\rightarrow}_{\Gamma} \varsigma' \wedge (v \sqsubseteq v') \wedge (\sigma_{k_{\varsigma}} \sqsubseteq \sigma_{k_{\varsigma'}}) \wedge (\forall a \in dom(\delta) : \delta(a) \sqsubseteq \sigma_{\varsigma'}(a))$
$\wedge (\forall a \in \mathcal{R}_v \setminus dom(\delta) : \sigma_{\mathsf{gc}}(a) \sqsubseteq \sigma_{\varsigma'}(a)) \wedge (\mathcal{R}_v \subseteq \mathcal{R}_{(\rightsquigarrow_{\sigma_{\varsigma'}})}(\mathcal{T}_{\mathsf{Val}}(v')))$

The generalisation relaxes the precondition of the implication. That is, the big-step evaluation may use another store than the small-step one, as long as it is equal to or subsumed by the latter. This is necessary to ensure that the induction hypothesis can be applied in the E-Let case, since the second evaluation step may use a more precise store after continuing with a more precise result from the first evaluation step. Likewise, the generalisation strengthens the postconditions, which now for instance also ensure that the bindings in the resulting delta store are more precise than the corresponding bindings in the resulting store of the small-step evaluation. This is again necessary for the E-Let case, because the second recursive evaluation continues using the delta store produced by the first.

Theorem 2 can now be seen as a corollary of Theorem 3. We elide the proof of Theorem 3 here for brevity; instead, we have verified this property using a mechanised proof[16], implemented using the Rocq theorem prover.

References

1. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea,August 29-31,2006, Proceedings. pp. 221–239 (2006). https://doi.org/10.1007/11823230_15
2. Darais, D., Labich, N., Nguyen, P.C., Van Horn, D.: Abstracting definitional interpreters (functional pearl). Proc. ACM Program. Lang. **1**(ICFP), 12:1–12:25 (2017). https://doi.org/10.1145/3110256
3. Earl, C., Might, M., Van Horn, D.: Pushdown control-flow analysis of higher-order programs. In: The 2010 Workshop on Scheme and Functional Programming (2010)
4. Earl, C., Sergey, I., Might, M., Van Horn, D.: Introspective pushdown analysis of higher-order programs. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012. pp. 177–188. ACM (2012). https://doi.org/10.1145/2364527.2364576
5. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque,New Mexico,USA,June 23-25,1993. pp. 237–247 (1993). https://doi.org/10.1145/155090.155113
6. Gabriel, R.P.: Performance and evaluation of LISP systems, vol. 263. MIT press Cambridge, Mass. (1985). https://doi.org/10.7551/mitpress/5298.001.0001

[16] For convenience, on a more standard version of λ_{ANF} without mutable variables.

7. Germane, K.: Full control-flow sensitivity for definitional interpreters. In: Static Analysis - 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20-22, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14995, pp. 120–146. Springer (2024). https://doi.org/10.1007/978-3-031-74776-2_5
8. Germane, K., Adams, M.D.: Liberate abstract garbage collection from the stack by decomposing the heap. In: Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12075, pp. 197–223. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_8
9. Germane, K., McCarthy, J.: Newly-single and loving it: improving higher-order must-alias analysis with heap fragments. Proc. ACM Program. Lang. **5**(ICFP), 1–28 (2021). https://doi.org/10.1145/3473601
10. Gilray, T., Adams, M.D., Might, M.: Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In: Proc. of the 21st SIGPLAN International Conf. on Functional Programming,ICFP 2016, Nara,Japan, September 18-22, 2016. pp. 407–420 (2016). https://doi.org/10.1145/2951913.2951936
11. Gilray, T., Lyde, S., Adams, M.D., Might, M., Van Horn, D.: Pushdown control-flow analysis for free. In: Proc. of the 43rd Annual SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2016, St. Petersburg, FL,USA,January 20-22,2016. pp. 691–704. ACM (2016). https://doi.org/10.1145/2837614.2837631
12. Glaze, D.A., Labich, N., Might, M., Van Horn, D.: Optimizing abstract abstract machines. In: ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'13,Boston,MA,USA-September 25-27, 2013. pp. 443–454. ACM (2013). https://doi.org/10.1145/2500365.2500604
13. Glaze, D.A., Van Horn, D.: Abstracting abstract control. In: Black, A.P., Tratt, L. (eds.) DLS'14, Proc. of the 10th ACM Symposium on Dynamic Languages, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. pp. 11–22. ACM (2014). https://doi.org/10.1145/2661088.2661098
14. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011. pp. 289–298. IEEE Computer Society (2011). https://doi.org/10.1109/CGO.2011.5764696
15. Hind, M., Pioli, A.: Assessing the effects of flow-sensitivity on pointer alias analyses. In: Levi, G. (ed.) Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1503, pp. 57–81. Springer (1998). https://doi.org/10.1007/3-540-49727-7_4
16. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.K.: Single and loving it: Must-alias analysis for higher-order languages. In: MacQueen, D.B., Cardelli, L. (eds.) POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998. pp. 329–341. ACM (1998). https://doi.org/10.1145/268946.268973
17. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5673, pp. 238–255. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_17
18. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Part of ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3444, pp. 5–20. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_2

19. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. thesis, Georgia Inst. of Technology,Atlanta,GA,USA (2007), http://hdl.handle.net/1853/16289
20. Might, M.: Logic-flow analysis of higher-order programs. In: Proc. of the 34th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL2007,Nice,France,January 17-19, 2007. pp. 185–198 (2007). https://doi.org/10.1145/1190216.1190247
21. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: Proc. of the 11th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006. pp. 13–25 (2006). https://doi.org/10.1145/1159803.1159807
22. Might, M., Shivers, O.: Exploiting reachability and cardinality in higher-order flow analysis. J. Funct. Program. **18**(5-6), 821–864 (2008). https://doi.org/10.1017/S0956796808006941
23. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 305–315. ACM (2010). https://doi.org/10.1145/1806596.1806631
24. Monat, R., Ouadjaout, A., Miné, A.: Value and allocation sensitivity in static Python analyses. In: Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP@PLDI 2020, London, UK, June 15, 2020. pp. 8–13. ACM (2020). https://doi.org/10.1145/3394451.3397205
25. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C with trace partitioning and auto-suggestions (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 31st Int. Conf., TACAS 2025, Part of ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15698, pp. 229–235. Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_17
26. Muth, R., Debray, S.K.: On the complexity of flow-sensitive dataflow analyses. In: POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000. pp. 67–80. ACM (2000). https://doi.org/10.1145/325694.325704
27. Oh, H., Yang, H., Yi, K.: Learning a strategy for adapting a program analysis via bayesian optimisation. In: Proc. of the 2015 ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015. pp. 572–588. ACM (2015). https://doi.org/10.1145/2814270.2814309
28. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. High. Order Symb. Comput. **11**(4), 363–397 (1998). https://doi.org/10.1023/A:1010027404223
29. Van Es, N., Stiévenart, Q., De Roover, C.: Garbage-free abstract interpretation through abstract reference counting. In: 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom. LIPIcs, vol. 134, pp. 10:1–10:33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.10
30. Van Horn, D., Might, M.: Abstracting abstract machines. In: Proc. of the 15th ACM SIGPLAN int. conf. on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 51–62 (2010). https://doi.org/10.1145/1863543.1863553
31. Vardoulakis, D., Shivers, O.: CFA2: a context-free approach to control-flow analysis. Log. Methods Comput. Sci. **7**(2) (2011). https://doi.org/10.2168/LMCS-7(2:3)2011