Garbage-Free Abstract Interpretation Through Abstract Reference Counting

Noah Van Es
Software Languages Lab, Vrije Universiteit Brussel, Belgium
noah.van.es@vub.be

Quentin Stiévenart
Software Languages Lab, Vrije Universiteit Brussel, Belgium
quentin.stievenart@vub.be

Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
coen.de.roover@vub.be

Abstract

Abstract garbage collection is the application of garbage collection to an abstract interpreter. Existing work has shown that abstract garbage collection can improve both the interpreter's precision and performance. Current approaches rely on heuristics to decide when to apply abstract garbage collection. Garbage will build up and impact precision and performance when the collection is applied infrequently, while too frequent applications will bring about their own performance overhead. A balance between these tradeoffs is often difficult to strike.

We propose a new approach to cope with the buildup of garbage in the results of an abstract interpreter. Our approach is able to eliminate all garbage, therefore obtaining the maximum precision and performance benefits of abstract garbage collection. At the same time, our approach does not require frequent heap traversals, and therefore adds little to the interpreters's running time. The core of our approach uses reference counting to detect and eliminate garbage as soon as it arises. However, reference counting cannot deal with cycles, and we show that cycles are much more common in an abstract interpreter than in its concrete counterpart. To alleviate this problem, our approach detects cycles and employs reference counting at the level of strongly connected components. While this technique in general works for any system that uses reference counting, we argue that it works particularly well for an abstract interpreter. In fact, we show formally that for the continuation store, where most of the cycles occur, the cycle detection technique only requires $\mathcal{O}(1)$ amortized operations per continuation push.

We present our approach formally, and provide a proof-of-concept implementation in the Scala-AM framework. We empirically show our approach achieves both the optimal precision and significantly better performance compared to existing approaches to abstract garbage collection.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases abstract interpretation, abstract garbage collection, reference counting

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.10

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.7`

Funding *Noah Van Es*: Funded by a PhD Fellowship of the Research Foundation – Flanders (FWO).

1 Introduction

Garbage collection (GC) is a well-known approach to automatic memory management that reclaims resources by removing items from the heap that are no longer needed. In general, there are two main approaches to garbage collection [2, 31]: tracing GC, such as the mark-and-sweep and stop-and-copy algorithms, and reference counting. Implementers of language

runtimes often deem tracing GC superior. Regular reference counting cannot deal with cyclic structures in the heap, and may result in a greater overhead in terms of both memory usage and performance [50].

An abstract interpreter [12, 40] soundly over-approximates the behaviour of a concrete interpreter. The approximation has to be chosen such that the abstract interpreter terminates on all programs, but still accounts for every behaviour the program may exhibit when executed by a concrete interpreter. It is not straightforward to design an efficient abstract interpreter [11, 27, 29]. Over-approximating too much renders its results less precise, while over-approximating too little slows its convergence down. Next to their abstract value lattice, abstract interpreters feature several configuration parameters to strike this balance, such as context-sensitivity [51, 44], widening [13] and different control-flow abstractions [30, 21], the interplay of which impacts performance and precision in less than predictable manners.

1.1 Motivating Abstract Garbage Collection

Garbage collection appears to be an exception to this rule. Incorporating garbage collection in an abstract interpreter can greatly improve both its precision and performance [42, 43]. Garbage in the heap can disrupt the interpreter's behaviour. Several means of guaranteeing termination involve bounding the set of memory locations available to the interpreter. The interpreter may therefore bring garbage "back to life" whenever its allocator returns an already-occupied location. The resulting imprecision propagates to the abstract state space explored by the interpreter, causing it to explore spurious states. Garbage collection can keep the interpreter's memory, and hence the state space it explores, free of garbage and its negative effects. It has been shown to reduce the explored state space by orders of magnitude [43, 47], resulting in equally large improvements in precision and performance.

Precision Loss Illustrated. The Scheme program in Listing 1 illustrates how surviving garbage can cause an abstract interpreter to lose precision.

Listing 1 Motivating abstract garbage collection.

```
1 (let [(double (lambda (x) (+ x x)))
2       (square (lambda (y) (* y y)))
3       (apply-fn (lambda (f n) (f n)))]
4   (apply-fn double 3)
5   (apply-fn square 4))
```

Clearly, a concrete interpreter evaluates this program to `16`. As mentioned above, an abstract interpreter will sometimes reuse the same memory location for different allocations. Monovariant allocation schemes such as 0CFA [51], for instance, reuse the same memory location for all allocations of the same lexical variable [20]. Values that end up in the same memory location are joined together, for instance by computing the union of all these values.

We denote the memory location used for a variable `v` by `@v`, and use the notation `@v` $\mapsto$ `a` to indicate that the heap at memory location `@v` contains the abstract value `a`. After the first function call on line 4, the interpreter's memory will contain `@f` $\mapsto$ `{double}` and `@n` $\mapsto$ `{3}`. The contents of `@f` and `@n` can be considered garbage after this function call, but is kept in memory without abstract garbage collection. This means that the second call of `apply-fn` on line 5 will have to join its argument values with the ones of the previous call that still reside at locations `@f` and `@n`, resulting in `@f` $\mapsto$ `{double,square}` and `@n` $\mapsto$ `{3,4}`.

The garbage left at these locations is now "alive" again, and will cause further losses in precision. Due to the imprecision at location `@f`, the abstract interpreter will consider `double` as well as `square` as potential targets for the function call, causing *spurious paths* in

the control flow that affect both precision and performance. For instance, the contents of `@n` will propagate to both `@x` ↦ `{3,4}` and `@y` ↦ `{3,4}`. The abstract interpreter effectively considers every possible combination of operator and operands, and evaluates this program to `{6,7,8,9,12,16}`. This is a major, but sound, over-approximation of the actual result.

Now consider that abstract GC had been applied immediately after the call on line 4, but just before the call on line 5. This application would have removed all garbage values from memory locations `@f`, `@n` and `@x`, as these locations are out of scope at this program point. The second function call would then have been explored under the precise mappings `@f` ↦ `{square}` and `@n` ↦ `{4}`, causing the abstract interpreter to evaluate this program to `{16}`.

1.2 Problem: Application Policies for Abstract Garbage Collection

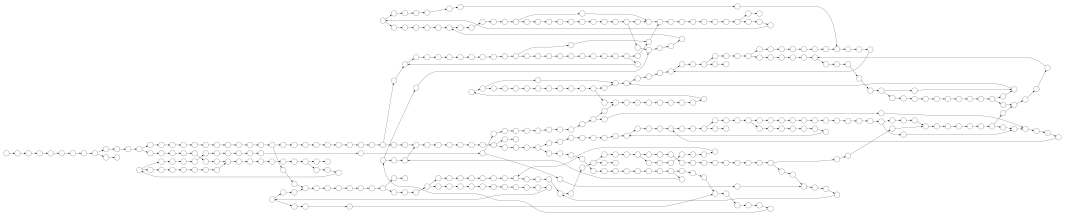
The objective of abstract GC is not to reclaim memory space, but to prevent potential precision loss due to the reuse of memory locations in the future that contain garbage values. Abstract GC therefore ought to be applied preemptively. The question is only *when*.

In the example of Listing 1, it was crucial to apply abstract GC just between the function calls on line 4 and 5 to avoid the precision loss. Of course, the abstract interpreter is not privy to this knowledge when it arrives to that program point. Nevertheless, the precision loss cannot be recovered at any later point through abstract GC.

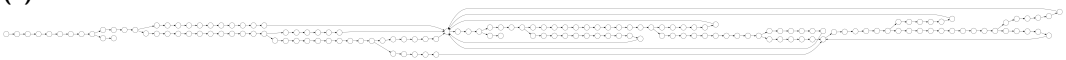
Survey of Existing Policies. A simple, but surprisingly effective policy that is often used in practice [43, 46, 49, 14, 16] is to apply abstract GC at every evaluation step of the abstract interpreter. Doing so ensures that all garbage is eliminated immediately, rendering the abstract interpreter effectively "garbage-free" [23]. A garbage-free interpreter only explores states that do not contain any garbage, which implies that no precision is lost to garbage values at all. However, realizing this property comes at a cost: frequent GC applications can become performance-detrimental, in particular for programs with large heaps [34, 16].

Other policies apply GC less frequently to avoid this overhead, which does result in some precision loss from non-collected garbage. For instance, the TAJS static analysis [26] applies abstract GC upon every function exit. This policy requires fewer GC applications, and function exits are good GC application candidates as garbage tends to be left behind when exiting a function's scope. The policy of ΓCFA [42] (and later, LFA [39]), which pioneered the use of abstract GC, is to apply abstract GC when the address allocator returns a memory location that is already in use. As such, it triggers abstract GC whenever values need to be joined in the store, so that by design a value can never be joined with garbage. While the ΓCFA policy avoids a common form of garbage-induced imprecision, it still applies GC frequently. The explored state space can moreover not be considered "garbage-free" as some of the states remain polluted and/or spurious. In fact, in a follow-up paper [43], the authors changed their policy to an application of GC at every step for the sake of garbage-freeness.

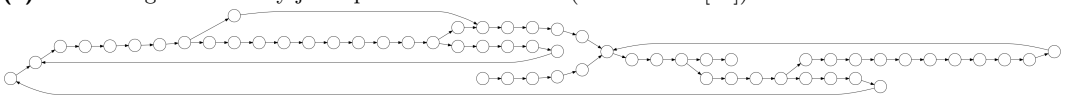
Precision Impact of Policies. Figure 1 illustrates the impact of common abstract GC policies on the state space explored by an abstract interpreter for a slightly more complicated Scheme program, included in Appendix C. For this example, the abstract interpreter uses a monovariant allocation policy and an abstract value lattice that approximates a value by the set of all its possible types. Policies that apply GC more frequently reduce the state space to be explored, which results in both higher precision and performance, but at the same time also causes a performance overhead. Our evaluation in Section 5 shows that this overhead significantly decreases the throughput of the abstract interpreter. In terms of performance,



(a) without GC: 431 states.



(b) when using GC at every join operation in the store (as in ΓCFA [42]): 159 states.



(c) when using GC at every step (as in [43, 16, 49, 46, 14]): 60 states.

Figure 1 State space explored by the abstract interpreter for different GC policies.

policies that apply GC less frequently (e.g., the ΓCFA one) often prove more efficient in terms of performance despite the larger state space to explore. In terms of precision, applying GC at every step yields the optimal result.

1.3 Approach: Abstract Reference Counting

We aim to realize "garbage-free" abstract interpretation more efficiently, attaining optimal precision without significant compromises to performance. We propose to collect garbage in an abstract interpreter using reference counting, and argue that its evaluation with respect to tracing GC needs to be reconsidered for abstract interpreters.

Reconsidering the Disadvantages of Reference Counting. The overhead of reference counting in the concrete is less outspoken in the abstract. The reason is twofold. First, abstract interpreters typically use fewer memory locations than their concrete counterparts. As such, fewer reference counts need to be maintained. Second, abstract interpreters can tolerate more memory management overhead than concrete interpreters. The fact that reference counting cannot reclaim cyclic garbage, however, remains an issue in the abstract. In fact, we have observed cyclic garbage to be more prevalent in the abstract. Section 4 addresses this problem separately as it directly impedes realizing the "garbage-free" property.

Reconsidering the Advantages of Reference Counting. The main advantage of reference counting stems from its continuous nature. Garbage is collected under reference counting as soon as it arises. As soon as the reference count of a memory location becomes zero, reference counting immediately reclaims that location. Tracing GC approaches hold on to garbage until the next GC application. Unless the application requires minimal GC pauses, this is generally not an issue for concrete interpreters. For abstract interpreters, however, garbage held on to impacts precision – requiring abstract tracing GCs to be applied frequently.

Simple bookkeeping suffices for abstract reference counting to immediately and efficiently eliminate all garbage, precluding the need for GC application policies that seek to balance the imprecision of long-lived garbage with the overhead of collection frequency. Abstract reference counting collects garbage continuously before it can cause a precision loss.

Finally, tracing GC needs to traverse the entire heap, which can get larger for more complex programs. Such expensive traversals increase overhead when applied frequently. The cost of reference counting is mostly insensitive to the size of the heap [2].

Contributions. The contributions of this work are as follows:

- We introduce abstract reference counting as a more efficient approach to abstract garbage collection that renders the explored state space garbage-free. We present this approach formally and prove that in terms of abstract GC, it is not only sound (i.e., it only removes garbage), but also complete (i.e., it removes all garbage).
- We discern sources of garbage cycles during abstract interpretation, and present a novel cycle detection algorithm for abstract reference counting based on these observations. We show that per insertion to the abstract interpreter's continuation store, from which most cyclic garbage stems, the algorithm only requires amortized $\mathcal{O}(1)$ additional operations.
- We provide a proof-of-concept implementation in the Scala-AM framework [54, 53] and an empirical evaluation of our approach using the Gabriel benchmark suite for Scheme programs [18]. The results show that our approach achieves optimal precision and is significantly more performant compared to existing approaches to abstract GC.

We present our work in a minimal setting using the AAM approach [23, 40] to abstract interpretation. Therefore, we elide common optimizations to AAM [27] and do not consider any of its variations (e.g., AAC [30]), although our approach remains applicable in such settings as well. An advantage of AAM is that it is a systematic method for the design of a wide variety of analyses, which is applicable to programs with highly-dynamic behavior (e.g., higher-order programs with mutable state). The performance and precision improvements brought by our work should therefore carry over to such analyses. Moreover, we argue that the concepts behind abstract reference counting are sufficiently general to support its incorporation in other existing analyses that already make use of abstract tracing GC.

Structure of the Paper. We introduce the necessary background in Section 2: an abstract interpreter for ANF-style λ-calculus, into which we have incorporated an abstract tracing GC. Section 3 replaces this abstract tracing GC by abstract reference counting, resulting in a more efficient collection of garbage during abstract interpretation. Section 4 explains why garbage cycles are more common in an abstract interpreter, and proposes a technique to detect these cycles so that reference counting can reclaim all cyclic garbage. We apply this technique to obtain an abstract interpreter that is completely garbage-free. Section 5 presents our experimental results. We compare to existing approaches to abstract GC in terms of precision and performance. We conclude with a discussion of related work in Section 6.

2 Background

Figure 2 defines the syntax of a higher-order language λ_{ANF}, based on the λ-calculus in A-Normal Form [17] (ANF). ANF is a syntactic form that restricts operators and operands to atomic expressions *ae* which can be evaluated immediately without impacting the program state. This simplification can be automated, is purely cosmetic, and without loss of generality.

We start with the formal definition of the small-step operational semantics for λ_{ANF} (Section 2.1). Following the Abstracting Abstract Machines (AAM) approach [23, 40], we derive an abstract interpreter from these semantics, and discuss the impact of abstract tracing GC on the state space explored by the interpreter (Section 2.2).

$$
\begin{array}{lll}
e \in \mathit{Exp} ::= \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 & [\textsc{Let}] & f, ae \in \mathit{Atom} ::= v \mid \mathit{lam} \\
\quad \mid\ f\ ae & [\textsc{Call}] & \mathit{lam} \in \mathit{Lam} ::= \lambda x.e \\
\quad \mid\ ae & [\textsc{Return}] & x, v \in \mathit{Var}\ (\text{a set of identifiers})
\end{array}
$$

Figure 2 Grammar of the minimalist higher-order language λ_{ANF}.

2.1 Concrete Interpretation of λ_{ANF}

The concrete interpreter for λ_{ANF} is formulated as an abstract machine using small-step operational semantics. We systematically describe the design of this abstract machine.

State Space. Figure 3 describes the state space Σ for the concrete interpreter. A state ς consists of an expression e under evaluation, an environment ρ mapping variables to addresses, a store σ mapping addresses to values (in case of λ_{ANF}, the only values are closures), a continuation store σ_k to model the "stack", which maps continuation addresses to continuations, and the current continuation address a_k (pointing to "the top of the stack"). A continuation κ consists of the variable address to which the resulting value should be bound and the next expression, environment, and continuation address to continue the evaluation. For a concrete interpreter, addresses come from an infinite set (e.g., $\mathsf{Addr} = \mathsf{KAddr} = \mathbb{N}$).

$$
\begin{array}{rlrl}
\varsigma \in \Sigma &= \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Store} & \mathit{clo} \in \mathsf{Clo} &= \mathit{Lam} \times \mathsf{Env} \\
&\quad \times \mathsf{KStore} \times \mathsf{KAddr} & \kappa \in \mathsf{Kont} &= \mathsf{Addr} \times \mathit{Exp} \times \mathsf{Env} \times \mathsf{KAddr} \\
\rho \in \mathsf{Env} &= \mathit{Var} \rightharpoonup \mathsf{Addr} & l \in \mathsf{Loc} &= \mathsf{Addr} \cup \mathsf{KAddr} \\
\sigma \in \mathsf{Store} &= \mathsf{Addr} \rightharpoonup \mathsf{Clo} & a \in \mathsf{Addr} &\ (\text{an infinite set}) \\
\sigma_k \in \mathsf{KStore} &= \mathsf{KAddr} \rightharpoonup \mathsf{Kont} & a_k \in \mathsf{KAddr} &\ (\text{an infinite set})
\end{array}
$$

Figure 3 State space of the concrete interpreter for λ_{ANF}.

For a state ς, we implicitly assume subscripted notations for its components so that $\varsigma = \langle e_\varsigma, \rho_\varsigma, \sigma_\varsigma, \sigma_{k_\varsigma}, a_{k_\varsigma} \rangle$. For (partial) maps, [] denotes the empty map, while 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 $x \neq a$.

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

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

Figure 4 shows the small-step transition relation $(\to)$ for λ_{ANF}. We assume that the value allocation function alloc and the continuation allocation function alloc_k always return a *fresh* address so that $\mathsf{alloc}(x, \varsigma) \notin \mathit{dom}(\sigma_\varsigma)$ and $\mathsf{alloc}_k(e, \varsigma) \notin \mathit{dom}(\sigma_{k_\varsigma})$.

We write $\varsigma_a \xrightarrow{n} \varsigma_b$ when $\varsigma_a = \varsigma_0 \to \varsigma_1 \to \ldots \to \varsigma_n = \varsigma_b$. Similarly, we define $(\xrightarrow{*})$ as the symmetric, transitive closure of $(\to)$. Note that for a concrete interpreter, this transition relation is deterministic, i.e. if $\varsigma_0 \xrightarrow{n} \varsigma_1$ and $\varsigma_0 \xrightarrow{n} \varsigma_2$ then $\varsigma_1 = \varsigma_2$.

Garbage Collection. We now add a tracing garbage collector to this concrete interpreter. We use notation and definitions similar to those used by Might and Shivers [42]. 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 state, environment, closure or continuation:

$$\frac{a = \mathsf{alloc}(x, \varsigma) \qquad a_k{}' = \mathsf{alloc}_k(e_1, \varsigma) \qquad \rho' = \rho[x \mapsto a]}{\underbrace{\langle \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2, \rho, \sigma, \sigma_k, a_k\rangle}_{\varsigma} \to \langle e_1, \rho, \sigma, \sigma_k[a_k{}' \mapsto \langle a, e_2, \rho', a_k\rangle], a_k{}'\rangle} \quad \text{(E-Let)}$$

$$\frac{\mathcal{A}(f, \rho, \sigma) = \langle \lambda x.e', \rho'\rangle \qquad \mathcal{A}(ae, \rho, \sigma) = v \qquad a = \mathsf{alloc}(x, \varsigma)}{\underbrace{\langle f\ ae, \rho, \sigma, \sigma_k, a_k\rangle}_{\varsigma} \to \langle e', \rho'[x \mapsto a], \sigma[a \mapsto v], \sigma_k, a_k\rangle} \quad \text{(E-Call)}$$

$$\frac{\mathcal{A}(ae, \rho, \sigma) = v \qquad \sigma_k(a_k) = \langle a', e', \rho', a_k{}'\rangle}{\langle ae, \rho, \sigma, \sigma_k, a_k\rangle \to \langle e', \rho', \sigma[a' \mapsto v], \sigma_k, a_k{}'\rangle} \quad \text{(E-Return)}$$

Figure 4 Transition rules of the concrete interpreter for λ_{ANF}.

$$\mathcal{T}_\Sigma(\langle e, \rho, \sigma, \sigma_k, a_k\rangle) = \mathcal{T}_{\mathsf{Env}}(\rho) \cup \{a_k\} \qquad \mathcal{T}_{\mathsf{Env}}(\rho) = \mathsf{range}(\rho)$$
$$\mathcal{T}_{\mathsf{Clo}}(\langle \lambda x.e, \rho\rangle) = \mathcal{T}_{\mathsf{Env}}(\rho) \qquad \mathcal{T}_{\mathsf{Kont}}(\langle a, e, \rho, a_k\rangle) = \mathcal{T}_{\mathsf{Env}}(\rho) \cup \{a, a_k\}$$

Next, we introduce the adjacency relation between addresses, $(\rightsquigarrow_\varsigma) : \mathsf{Loc} \times \mathsf{Loc}$, where intuitively $\widehat{l} \rightsquigarrow_\varsigma \widehat{l'}$ means that there is a reference from address $\widehat{l}$ to address $\widehat{l'}$:

$$\frac{l \in \mathcal{T}_{\mathsf{Clo}}(\sigma_\varsigma(a))}{a \rightsquigarrow_\varsigma l} \qquad \frac{l \in \mathcal{T}_{\mathsf{Kont}}(\sigma_{k\varsigma}(a_k))}{a_k \rightsquigarrow_\varsigma l}$$

and use the notation $(\overset{*}{\rightsquigarrow}_\varsigma)$ for its reflexive transitive closure. The function $\mathcal{R} : \Sigma \to \mathcal{P}(\mathsf{Loc})$, which computes all reachable addresses of a state, is then defined as follows:

$$\mathcal{R}(\varsigma) = \{l' \in \mathsf{Loc} \mid l \in \mathcal{T}_\Sigma(\varsigma) \wedge l \overset{*}{\rightsquigarrow}_\varsigma l'\}.$$

Finally, the function $\Gamma : \Sigma \to \Sigma$ applies garbage collection to a state by restricting the domain of the store and the continuation store to the addresses that are still reachable. The function Γ can be seen as a tracing garbage collector, since it defines garbage directly in terms of what is no longer reachable [2, 31]. Domain restriction is denoted by a vertical bar, so that $f|_X(x) = f(x)$ for $x \in X$ and $f|_X(x)$ is undefined for $x \notin X$.

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

To keep the abstract machine's transition relation simple and deterministic, we incorporate GC into the semantics by defining a new relation $(\to_\Gamma)$ as the composition of $(\to)$ and Γ:

$$(\to_\Gamma) = \Gamma \circ (\to)$$

This means that $(\to_\Gamma)$ first takes an evaluation step, followed by a garbage collection. This "GC at every step" strategy makes the interpreter *garbage-free*: during evaluation, for every state ς, we have that $\mathcal{R}(\varsigma) = dom(\sigma_\varsigma) \cup dom(\sigma_{k\varsigma})$. However, for a concrete interpreter, this property does not really have an impact on the result of the evaluation. Theorem 1 states this formally: for a given state ς, taking n transition steps using $(\to_\Gamma)$ leads to the same state, *modulo GC*, as taking n transition steps using $(\to)$.

▶ **Theorem 1.** *If* $\varsigma_0 \overset{n}{\to}_\Gamma \varsigma_1$ *and* $\varsigma_0 \overset{n}{\to} \varsigma_2$*, then* $\varsigma_1 = \Gamma(\varsigma_2)$.

Proof. This follows directly from Might and Shivers [42] and is detailed in Appendix A. ◀

This confirms that concrete garbage collection does not affect the evaluation of a program. For a practical implementation, however, concrete garbage collection can still be useful when the size of the σ and σ_k components become too large to fit in computer memory.

Program Semantics. Using the transition relation ($\rightarrow_\Gamma$), we can define a function $\mathsf{eval} : Exp \rightarrow \mathcal{P}(\Sigma)$ which computes all the states reachable by the concrete interpreter, starting from the initial state of the program:

$$\mathsf{eval}(e) = \{\varsigma \in \Sigma \mid \langle e, [], [], [], a_{\mathsf{halt}}\rangle \xrightarrow{*}_\Gamma \varsigma\}$$

where a_{halt} is a special address in the set KAddr. By computing $\mathsf{eval}(e)$, we obtain the *collecting semantics*[1] of the program e. Unfortunately, when e does not terminate, the concrete interpreter may explore an infinite amount of states in Σ. Hence, for a concrete interpreter, the set $\mathsf{eval}(e)$ is potentially infinite, and therefore not always computable.

2.2 Abstract Interpretation of λ_{ANF}

We now systematically turn this concrete interpreter for λ_{ANF} into an abstract interpreter, highlighting the important changes that we need to make in *gray*.

State Space. The main issue with the concrete interpreter in Section 2.1 is that the state space Σ is infinite, and hence the set $\mathsf{eval}(e)$ is not always computable for any program e. To solve this issue, the AAM approach [23, 40] replaces the infinite sets Addr and KAddr with finite sets $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$, respectively. One can easily verify that this suffices to keep the state space finite. Figure 5 shows the *abstract* state space $\widehat{\Sigma}$.

$$\begin{aligned}
\widehat{\varsigma} \in \widehat{\Sigma} &= \mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \\
&\quad \times \widehat{\mathsf{KStore}} \times \widehat{\mathsf{KAddr}} \\
\widehat{\rho} \in \widehat{\mathsf{Env}} &= \mathit{Var} \rightharpoonup \widehat{\mathsf{Addr}} \\
\widehat{\sigma} \in \widehat{\mathsf{Store}} &= \widehat{\mathsf{Addr}} \rightarrow \mathcal{P}(\widehat{\mathsf{Clo}}) \\
\widehat{\sigma}_k \in \widehat{\mathsf{KStore}} &= \widehat{\mathsf{KAddr}} \rightarrow \mathcal{P}(\widehat{\mathsf{Kont}})
\end{aligned}
\qquad
\begin{aligned}
\widehat{clo} \in \widehat{\mathsf{Clo}} &= \mathit{Lam} \times \widehat{\mathsf{Env}} \\
\widehat{\kappa} \in \widehat{\mathsf{Kont}} &= \widehat{\mathsf{Addr}} \times \mathit{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{KAddr}} \\
\widehat{l} \in \widehat{\mathsf{Loc}} &= \widehat{\mathsf{Addr}} \cup \widehat{\mathsf{KAddr}} \\
\widehat{a} \in \widehat{\mathsf{Addr}} &\text{ (a finite set)} \\
\widehat{a}_k \in \widehat{\mathsf{KAddr}} &\text{ (a finite set)}
\end{aligned}$$

Figure 5 State space of the abstract interpreter for λ_{ANF}.

As the abstract interpreter can only use a finite number of addresses, it may need to reuse the same address for multiple allocations. Values that end up at the same address need to be *joined* together to obtain a sound, but finite approximation. Hence, both the store and continuation store now map addresses to a *set* of closures and a *set* of continuations, respectively. We introduce the *join operator* $\sqcup$ defined as follows:

$$(\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(\widehat{a}) = \widehat{\sigma}_1(\widehat{a}) \cup \widehat{\sigma}_2(\widehat{a}) \qquad (\widehat{\sigma_{k1}} \sqcup \widehat{\sigma_{k2}})(\widehat{a}_k) = \widehat{\sigma_{k1}}(\widehat{a}_k) \cup \widehat{\sigma_{k2}}(\widehat{a}_k) \qquad \bot_{\widehat{\sigma}} = \bot_{\widehat{\sigma}_k} = \lambda\widehat{l}.\,\emptyset$$

Evaluation Rules. The atomic evaluation function $\widehat{\mathcal{A}} : \mathit{Atom} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \rightarrow \mathcal{P}(\widehat{\mathsf{Clo}})$ for the abstract interpreter evaluates atomic expressions to a *set* of closures:

$$\widehat{\mathcal{A}}(v, \widehat{\rho}, \widehat{\sigma}) = \widehat{\sigma}(\widehat{\rho}(v)) \qquad \widehat{\mathcal{A}}(lam, \widehat{\rho}, \widehat{\sigma}) = \{\, \langle lam, \widehat{\rho}\rangle \,\}$$

[1] not to be confused with *garbage collection*

We leave the choice of sets $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$, as well as the allocation functions $\widehat{\mathsf{alloc}}$ and $\widehat{\mathsf{alloc}}_k$ open as configuration parameters of the abstract interpreter (resulting in a particular *allocation policy*). Any allocation policy yields a sound and decidable analysis [23] (as long as the sets $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$ are chosen to be finite). However, the choice is not arbitrary, as the allocation policy decides how often (determined by the size of $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$) and when (determined by $\widehat{\mathsf{alloc}}$ and $\widehat{\mathsf{alloc}}_k$) addresses need to be reused. This choice therefore affects the precision and *polyvariance* [20] of the abstract interpreter[2]. For example, the following allocation policy results in monovariant analysis [51] which reuses the same address for all allocations of the same variable:

$$\widehat{\mathsf{Addr}} = Var \qquad \widehat{\mathsf{KAddr}} = Exp \qquad \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) = x \qquad \widehat{\mathsf{alloc}}_k(e, \widehat{\varsigma}) = e$$

Figure 6 shows the abstract transition relation ($\widehat{\rightarrow}$) for λ_{ANF}. Note that, unlike in the concrete interpreter for λ_{ANF}, the transition relation is no longer deterministic.

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{a}'_k = \widehat{\mathsf{alloc}}_k(e_1, \widehat{\varsigma}) \qquad \widehat{\rho}' = \widehat{\rho}[x \mapsto \widehat{a}]}{\underbrace{\langle \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle}_{\widehat{\varsigma}} \widehat{\rightarrow} \langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k \sqcup [\widehat{a}'_k \mapsto \{\, \langle \widehat{a}, e_2, \widehat{\rho}', \widehat{a}_k \rangle \,\}], \widehat{a}'_k \rangle} \quad \text{(E-Let)}$$

$$\frac{\widehat{\mathcal{A}}(f, \widehat{\rho}, \widehat{\sigma}) \ni \langle \lambda x.e', \widehat{\rho}' \rangle \qquad \widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma})}{\underbrace{\langle f\, ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle}_{\widehat{\varsigma}} \widehat{\rightarrow} \langle e', \widehat{\rho}'[x \mapsto \widehat{a}], \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}_k \rangle} \quad \text{(E-Call)}$$

$$\frac{\widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\sigma}_k(\widehat{a}_k) \ni \langle \widehat{a}', e', \widehat{\rho}', \widehat{a}'_k \rangle}{\langle ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle \widehat{\rightarrow} \langle e', \widehat{\rho}', \widehat{\sigma} \sqcup [\widehat{a}' \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}'_k \rangle} \quad \text{(E-Return)}$$

Figure 6 Transition rules of the abstract interpreter for λ_{ANF}.

Garbage Collection. Just as in the concrete interpreter, we can perform garbage collection in the abstract interpreter (known as *abstract garbage collection*) by removing all addresses from $\widehat{\sigma}_{\widehat{\varsigma}}$ and $\widehat{\sigma_k}_{\widehat{\varsigma}}$ that are no longer reachable from $\widehat{\varsigma}$. The definitions remain mostly unchanged:

$$\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle) = \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) \cup \{\widehat{a}_k\} \qquad \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) = \mathsf{range}(\widehat{\rho})$$

$$\widehat{\mathcal{T}}_{\widehat{\mathsf{Clo}}}(\langle \lambda x.e, \widehat{\rho} \rangle) = \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) \qquad \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\langle a, e, \widehat{\rho}, \widehat{a}_k \rangle) = \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) \cup \{\widehat{a}, \widehat{a}_k\}$$

$$\widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{\mathsf{V}}) = \bigcup_{\widehat{clo} \in \widehat{\mathsf{V}}} \widehat{\mathcal{T}}_{\widehat{\mathsf{Clo}}}(\widehat{clo}) \qquad \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Kont}})}(\hat{\mathsf{K}}) = \bigcup_{\widehat{\kappa} \in \widehat{\mathsf{K}}} \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa})$$

Similarly, the adjacency relation $(\widehat{\rightsquigarrow}_{\widehat{\varsigma}}) : \widehat{\mathsf{Loc}} \times \widehat{\mathsf{Loc}}$ is adapted for abstract addresses:

$$\frac{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{\sigma}_{\widehat{\varsigma}}(\widehat{a}))}{\widehat{a} \mathrel{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l}} \qquad \frac{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Kont}})}(\widehat{\sigma_k}_{\widehat{\varsigma}}(\widehat{a}_k))}{\widehat{a}_k \mathrel{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l}}$$

[2] For the sake of simplicity, our abstract interpreter does not include a timestamp component (as in [23]), which could be used to express more complex allocation policies such as k-CFA with $k \geq 1$.

All reachable addresses can then be computed using $\widehat{\mathcal{R}} : \widehat{\Sigma} \to \mathcal{P}(\widehat{\mathsf{Loc}})$:

$$\widehat{\mathcal{R}}(\widehat{\varsigma}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}) \wedge \widehat{l} \overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l'}\}.$$

and the abstract garbage collection function $\widehat{\Gamma} : \widehat{\Sigma} \to \widehat{\Sigma}$ is defined as follows:

$$\widehat{\Gamma}(\widehat{\varsigma}) = \langle e_{\hat{\varsigma}}, \widehat{\rho}_{\hat{\varsigma}}, \widehat{\sigma}_{\hat{\varsigma}}|_{\widehat{\mathcal{R}}(\hat{\varsigma})}, \widehat{\sigma_k}_{\hat{\varsigma}}|_{\widehat{\mathcal{R}}(\hat{\varsigma})}, \widehat{a_k}_{\hat{\varsigma}} \rangle$$

where $f|_X(x) = f(x)$ for $x \in X$ and $f|_X(x) = \emptyset$ for $x \notin X$. The new abstract transition relation $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ can again be defined as a composition of $\widehat{\Gamma}$ and $(\widehat{\rightarrow})$:

$$(\widehat{\rightarrow}_{\widehat{\Gamma}}) = \widehat{\Gamma} \circ (\widehat{\rightarrow})$$

Applying garbage collection at every evaluation step ensures that we end up with a *garbage-free* abstract interpreter. More precisely, if we define $\widehat{\mathcal{S}} : \widehat{\Sigma} \to \mathcal{P}(\widehat{\mathsf{Loc}})$, so that $\widehat{\mathcal{S}}(\widehat{\varsigma})$ is the set of addresses bound in the stores of $\widehat{\varsigma}$, as:

$$\widehat{\mathcal{S}}(\widehat{\varsigma}) = \{\widehat{a} \in \widehat{\mathsf{Addr}} \mid \widehat{\sigma}_{\hat{\varsigma}}(\widehat{a}) \neq \emptyset\} \cup \{\widehat{a}_k \in \widehat{\mathsf{KAddr}} \mid \widehat{\sigma_k}_{\hat{\varsigma}}(\widehat{a}_k) \neq \emptyset\}$$

Then we define garbage-free as follows:

▶ **Definition 2** (Garbage-free). *A state $\widehat{\varsigma}$ is* garbage-free *iff $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \widehat{\mathcal{S}}(\widehat{\varsigma})$, or equivalently: $\widehat{\Gamma}(\widehat{\varsigma}) = \widehat{\varsigma}$. A transition relation $(\widehat{\rightarrow})$ is* garbage-free *iff it preserves garbage-freeness. That is, $(\widehat{\rightarrow})$ is garbage-free iff for every garbage-free state $\widehat{\varsigma}$ where $\widehat{\varsigma} \widehat{\rightarrow} \widehat{\varsigma}'$, $\widehat{\varsigma}'$ is garbage-free.*

▶ **Theorem 3.** *$(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free*

Proof. By definition, $\widehat{\Gamma}(\widehat{\varsigma})$ is always a garbage-free state, and $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ applies $\widehat{\Gamma}$ to the resulting state. Therefore, every resulting state of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free, hence $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free. ◀

By design, every state produced by $\widehat{\varsigma}$ produced by $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free (Theorem 3). Note however, that applying garbage collection at every step, as in $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, may not be practical, since computing the set $\widehat{\mathcal{R}}(\widehat{\varsigma})$ at every evaluation step causes a significant performance overhead. In a practical implementation, one may choose to apply abstract garbage collection at less regular intervals; however, in doing so the abstract interpreter is no longer guaranteed to be garbage-free (i.e., in general, $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$).

Theorem 1 previously showed that this garbage-free property does not really matter for a concrete interpreter. This is no longer the case for the abstract interpreter: as a counter-example, consider a monovariant analysis of the program presented in Listing 1. If the abstract interpreter uses the transition relation $(\widehat{\rightarrow})$, then after the function call on line 4 we have that $\widehat{\sigma}_{\hat{\varsigma}}(f) = \{\widehat{clo}_{\texttt{double}}\}$, where $\widehat{clo}_{\texttt{double}}$ is the closure of the `double` function that f was bound to. Note that at that program point, $f \notin \widehat{\mathcal{R}}(\hat{\varsigma})$, but since $(\widehat{\rightarrow})$ does not perform abstract GC, the binding is not removed from $\widehat{\sigma}$. Therefore, at the function call of line 5, when f needs to be bound to the closure $\widehat{clo}_{\texttt{square}}$ of the `square` function, the values are joined together using $\sqcup$ so that $\widehat{\sigma}_{\hat{\varsigma}'}(f) = \{\widehat{clo}_{\texttt{double}}, \widehat{clo}_{\texttt{square}}\}$. Intuitively, this means that the abstract interpreter does not know exactly which of these two closures f is bound to, and to soundly over-approximate any possible execution behaviour, it needs to consider both options. This imprecision therefore introduces spurious control flow into the abstract interpreter, which implies that it will explore more states than necessary. Also note that at that point, $f \in \widehat{\mathcal{R}}(\hat{\varsigma}')$, so applying $\widehat{\Gamma}$ to collect garbage is no longer able to recover this loss of precision in $\widehat{\varsigma}'$. In contrast, if we keep the abstract interpreter garbage-free by using the $(\widehat{\rightarrow}_{\widehat{\Gamma}})$

transition relation, we can avoid this precision loss. After the function call at line 4, since $f \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$, and since all states produced by $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ are garbage-free, we have that $f \notin \widehat{\mathcal{S}}(\widehat{\varsigma})$, which implies that $\widehat{\sigma}_{\widehat{\varsigma}}(f) = \emptyset$. Therefore, for the next call to `apply-fn`, $\widehat{\sigma}_{\widehat{\varsigma}'}(f) = \{\widehat{clo}_{\texttt{square}}\}$, so that the abstract interpreter knows precisely which closure is bound to f.

In general, an abstract interpreter loses precision whenever two non-empty sets are joined in the store (or continuation store) using $\sqcup$ (which happens when the abstract interpreter reuses an address that is already allocated). As common wisdom puts it: *"merging [i.e., join] is the enemy of precision"* [29], and as such it should only be used sparingly. This is exactly what abstract garbage collection achieves by emptying all sets at addresses that are no longer reachable, so that these sets can no longer be merged with in the future.

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

$$\widehat{\mathsf{eval}}(e) = \{\widehat{\varsigma} \in \widehat{\Sigma} \mid \langle e, [], \bot_{\widehat{\sigma}}, \bot_{\widehat{\sigma}_k}, \widehat{a}_{\mathsf{halt}}\rangle \overset{*}{\widehat{\rightarrow}}_{\widehat{\Gamma}} \widehat{\varsigma}\}$$

where $\widehat{a}_{\mathsf{halt}}$ is a special address in the set $\widehat{\mathsf{KAddr}}$. As $\widehat{\Sigma}$ is finite, for any program e it is guaranteed that $\widehat{\mathsf{eval}}(e)$ is finite and therefore computable. We can reason over the behaviour of e by reasoning over $\widehat{\mathsf{eval}}(e)$ to obtain a sound and decidable program analysis.

The definition of $\widehat{\mathsf{eval}}$ reveals another benefit of abstract garbage collection. Garbage-free abstract interpretation reduces the state space that needs to be explored (i.e., the size of $\widehat{\mathsf{eval}}(e)$), which improves both its precision and performance while still producing a sound over-approximation of the program's concrete semantics. That is, a garbage-free abstract interpreter only explores a smaller subset of $\widehat{\Sigma}$ where $\widehat{\Gamma}(\widehat{\varsigma}) = \widehat{\varsigma}$. In contrast, an abstract interpreter that is not garbage-free may explore equivalent states multiple times, i.e., it may explore $\widehat{\varsigma}_1, \widehat{\varsigma}_2, ..., \widehat{\varsigma}_k$ where $\widehat{\varsigma}_1 \neq \widehat{\varsigma}_2 \neq ... \neq \widehat{\varsigma}_k$, but $\widehat{\Gamma}(\widehat{\varsigma}_1) = \widehat{\Gamma}(\widehat{\varsigma}_2) = ... = \widehat{\Gamma}(\widehat{\varsigma}_k)$. This can be seen clearly in Figure 1, where performing a GC before every join as in ΓCFA [43] results in 159 states, while performing a GC for every state results in 60 states.

3 Abstract Reference Counting

Section 2.2 detailed the prototypical design of an abstract interpreter that is garbage-free thanks to abstract GC [42]. The benefits of being garbage-free include substantial improvements to both the precision and performance of the abstract interpreter. The cost of realizing this property through abstract tracing GC is the need for an application policy that collects garbage at every evaluation step, which results in its own performance overhead.

In this section, we develop a more efficient design for garbage-free abstract interpreters. Instead of computing $\widehat{\mathcal{R}}(\widehat{\varsigma})$ at every step (as in tracing garbage collection), the main idea is to keep track of all references to every address $\widehat{l}$ in $\widehat{\varsigma}$ (as in reference counting), from which it is possible to determine whether $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$. When there are no more references to $\widehat{l}$, we have that $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$, and the memory location can be reclaimed. If the abstract interpreter consistently collects all addresses without references, and there are no cyclic references in memory[3], then for every address $\widehat{l}$ that still has references, we have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$.

[3] which means that $\forall\, \widehat{l}_1, \widehat{l}_2 \in \widehat{\mathsf{Loc}}, \neg(\widehat{l}_1 \overset{*}{\rightsquigarrow} \widehat{l}_2 \wedge \widehat{l}_2 \overset{*}{\rightsquigarrow} \widehat{l}_1)$.

As the abstract interpreter performs abstract garbage collection through reference counting, we refer to it as *abstract reference counting*. We augment the abstract interpreter of Section 2.2 with abstract reference counting and show that, in the absence of cycles, it is equivalent to a garbage-free abstract interpreter using abstract tracing GC with the transition relation ($\widehat{\rightarrow}_{\widehat{\Gamma}}$). Section 4 discusses our solution to the problem of cyclic garbage.

3.1 Abstract Interpretation with Reference Counting using ($\widehat{\rightarrow}_{\mathsf{arc}}$)

$$\widehat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc}} = \mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \times \widehat{\mathsf{KStore}} \times \widehat{\mathsf{KAddr}} \times \widehat{\mathsf{Refs}} \qquad \widehat{\phi} \in \widehat{\mathsf{Refs}} = \widehat{\mathsf{Loc}} \to \mathcal{P}(\widehat{\mathsf{Loc}})$$

Figure 7 State space of the abstract interpreter with reference counting for λ_{ANF}. Other components preserve their definition given in Figure 5.

Figure 7 shows the updated abstract state space. Every state $\widehat{\varsigma}$ has been augmented with an additional component $\widehat{\phi}_{\widehat{\varsigma}}$ that keeps track of the references between addresses. Defined deterministically in terms of $\widehat{\varsigma}$'s original components, this addition does not increase the complexity of the state space explored by the abstract interpreter:

$$\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l'} \mathrel{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l}\}$$

That is, $\widehat{\phi}(\widehat{l})$ contains all addresses that refer directly to $\widehat{l}$. The actual *reference count* of $\widehat{l}$ is obtained as $|\widehat{\phi}(\widehat{l})|$. Mapping the addresses in $\widehat{\phi}$ to a set of addresses ($\mathcal{P}(\widehat{\mathsf{Loc}})$) rather than an actual reference count ($\mathbb{N}$)[4] renders our formalization more concise as well as amenable to the incorporation of cycle detection (cf. Section 4). We introduce the following definitions:

$$(\widehat{\phi}_1 \sqcup \widehat{\phi}_2)(\widehat{l}) = \widehat{\phi}_1(\widehat{l}) \cup \widehat{\phi}_2(\widehat{l}) \qquad \bot_{\widehat{\phi}} = \lambda \widehat{l}.\, \emptyset$$

We introduce the new transition relation ($\widehat{\rightarrow}_{\mathsf{arc}}$) as a composition of an auxiliary transition relation ($\widehat{\rightarrow}_0$) and a function $\widehat{\mathsf{collect}}$. The auxiliary relation ($\widehat{\rightarrow}_0$) extends the transition relation ($\widehat{\rightarrow}$) of Figure 6 by updating the references of addresses (in $\widehat{\phi}$) for every update to the store (or continuation store). The function $\widehat{\mathsf{collect}}$ is responsible for collecting garbage as reference counts become zero (i.e. $|\widehat{\phi}(\widehat{l})| = 0$ for some $\widehat{l}$) after every transition step of ($\widehat{\rightarrow}_0$).

$$\frac{\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_0} \widehat{\varsigma}' \qquad \widehat{\varsigma}'' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}')}{\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\varsigma}''}$$

Figure 8 shows the definition of the auxiliary transition ($\widehat{\rightarrow}_0$). Whenever the updated transition rules need to insert a value $\widehat{v}$ (or continuation $\widehat{\kappa}$) at address $\widehat{l}$, they now add $\widehat{l}$ to the set of references $\widehat{\phi}(\widehat{l'})$ for every address $\widehat{l'}$ that is directly reachable from the new value $\widehat{v}$ (or continuation $\widehat{\kappa}$) that $\widehat{l}$ is bound to. These addresses can be computed using $\widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v})$ (or $\widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa})$), where $\widehat{\mathcal{T}}$ was defined in Section 2.2. While ($\widehat{\rightarrow}_0$) maintains $\widehat{\phi}$ as intended, it does not yet collect any garbage. Indeed, none of the transition rules in ($\widehat{\rightarrow}_0$) remove any references to or from addresses, and the components $\widehat{\sigma}$, $\widehat{\sigma}_k$ and $\widehat{\phi}$ only grow monotonically.

[4] Note that it is not necessary to maintain a finite approximation such as $\widehat{\mathbb{N}} = \{0, 1, \infty\}$ for this reference count. The key insight is to count the "abstract" references to an abstract address, not the concrete references to the corresponding concrete address(es) that are approximated by that abstract address. That is, our approach performs abstract interpretation *with* reference counting, not an abstract interpretation *of* reference counting.

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{a}'_k = \widehat{\mathsf{alloc}}_k(e_1, \widehat{\varsigma}) \qquad \widehat{\rho}' = \widehat{\rho}[x \mapsto \widehat{a}] \qquad \widehat{\kappa} = \langle \widehat{a}, e_2, \widehat{\rho}', \widehat{a}_k \rangle \qquad \widehat{\phi}' = \widehat{\phi} \sqcup \bigsqcup_{\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa})} [\widehat{l} \mapsto \{\widehat{a}'_k\}]}{\underbrace{\langle \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi} \rangle}_{\widehat{\varsigma}} \;\widehat{\rightarrow}_0\; \langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k \sqcup [\widehat{a}'_k \mapsto \widehat{\kappa}], \widehat{a}'_k, \widehat{\phi}' \rangle} \quad \text{(E-Let)}$$

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{\mathcal{A}}(f, \widehat{\rho}, \widehat{\sigma}) \ni \langle \lambda x.e', \widehat{\rho}' \rangle \qquad \widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\phi}' = \widehat{\phi} \sqcup \bigsqcup_{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v})} [\widehat{l} \mapsto \{\widehat{a}\}]}{\underbrace{\langle f\, ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi} \rangle}_{\widehat{\varsigma}} \;\widehat{\rightarrow}_0\; \langle e', \widehat{\rho}'[x \mapsto \widehat{a}], \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}' \rangle} \quad \text{(E-Call)}$$

$$\frac{\widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\sigma}_k(\widehat{a}_k) \ni \langle \widehat{a}, e', \widehat{\rho}', \widehat{a}'_k \rangle \qquad \widehat{\phi}' = \widehat{\phi} \sqcup \bigsqcup_{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v})} [\widehat{l} \mapsto \{\widehat{a}\}]}{\langle ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi} \rangle \;\widehat{\rightarrow}_0\; \langle e', \widehat{\rho}', \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}'_k, \widehat{\phi}' \rangle} \quad \text{(E-Return)}$$

Figure 8 Auxiliary transition relation ($\widehat{\rightarrow}_0$) for abstract reference counting in λ_{ANF}.

Recall that garbage is defined as all addresses that are not reachable, i.e., all addresses that are not in $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \{\widehat{l}' \in \widehat{\mathsf{Loc}} \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}) \wedge \widehat{l} \overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l}'\}$. We refer to $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ as the *root set* of addresses for a given state $\widehat{\varsigma}$. Whenever $\widehat{\varsigma}$ transitions to $\widehat{\varsigma}'$, the root set changes from $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ to $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$, which creates garbage when an address $\widehat{l}$ that was in $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ is no longer in $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$ *and* $\widehat{l}$ is not referenced from any other address (i.e., $|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| = 0$). When this happens, the abstract interpreter needs to garbage collect $\widehat{l}$ by removing it from the store. Garbage collecting an address $\widehat{l}$ also removes $\widehat{l}$ from $\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l}')$ for every $\widehat{l}' \in \widehat{\mathsf{Loc}}$ where $\widehat{l} \;\widehat{\rightsquigarrow}_{\widehat{\varsigma}'}\; \widehat{l}'$. This is akin to decrementing the reference count of all addresses $\widehat{l}'$ referenced by $\widehat{l}$. The update can cause other addresses to be garbage collected because they no longer have any references. Note that $\widehat{\phi}$ only takes into account references coming from other addresses; we take care of root references using the function $\widehat{\mathsf{collect}} : \widehat{\Sigma}_{\mathsf{arc}} \times \widehat{\Sigma}_{\mathsf{arc}} \to \widehat{\Sigma}_{\mathsf{arc}}$, which removes all garbage from $\widehat{\varsigma}'$ that is created due to a change in the root set (from $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ to $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$) as $\widehat{\varsigma} \;\widehat{\rightarrow}_0\; \widehat{\varsigma}'$.

$$\widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}') = \langle e_{\widehat{\varsigma}'}, \widehat{\rho}_{\widehat{\varsigma}'}, \widehat{\sigma}_{\widehat{\varsigma}'} \setminus \widehat{G}, \widehat{\sigma_k}_{\widehat{\varsigma}'} \setminus \widehat{G}, \widehat{a_k}_{\widehat{\varsigma}'}, \widehat{\phi}' \rangle \text{where } \langle \widehat{\phi}', \widehat{G} \rangle = \widehat{\mathsf{check*}}(\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}), \emptyset, \widehat{\phi}_{\widehat{\varsigma}'})$$

$$\widehat{\mathsf{check*}}(\widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \langle \widehat{\phi}, \widehat{G} \rangle & \text{if } \widehat{C} = \emptyset \\ \widehat{\mathsf{check}}(\widehat{l}, \widehat{C} \setminus \{\widehat{l}\}, \widehat{G}, \widehat{\phi}) & \text{otherwise, for any } \widehat{l} \in \widehat{C} \end{cases}$$

$$\widehat{\mathsf{check}}(\widehat{l}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \widehat{\mathsf{dealloc}}(\widehat{l}, \widehat{C}, \widehat{G}, \widehat{\phi}) & \text{if } |\widehat{\phi}(\widehat{l})| = 0 \wedge \widehat{l} \notin \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}') \\ \widehat{\mathsf{check*}}(\widehat{C}, \widehat{G}, \widehat{\phi}) & \text{otherwise} \end{cases}$$

$$\widehat{\mathsf{dealloc}}(\widehat{l}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \widehat{\mathsf{check*}}(\widehat{C} \cup \widehat{S}, \widehat{G} \cup \{\widehat{l}\}, \widehat{\phi} \ominus \bigsqcup_{\widehat{l}' \in \widehat{S}} [\widehat{l}' \mapsto \{\widehat{l}\}]) \quad \text{where } \widehat{S} = \{\widehat{l}' \mid \widehat{l} \;\widehat{\rightsquigarrow}_{\widehat{\varsigma}'}\; \widehat{l}'\}$$

We use the notation for set removal to remove elements from a map m, so that $(m \setminus S)(x) = \emptyset$ if $x \in S$ and $(m \setminus S)(x) = m(x)$ if $x \notin S$. We define $(\widehat{\phi}_1 \ominus \widehat{\phi}_2)(\widehat{l}) = \widehat{\phi}_1(\widehat{l}) \setminus \widehat{\phi}_2(\widehat{l})$ to conveniently update $\widehat{\phi}$ when references are removed (due to addresses being garbage collected).

The function $\widehat{\mathsf{collect}}$ checks for every address $\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ whether it is still referenced by at least one other address ($|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| > 0$) or whether it is still part of the root set ($\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$). If this is not the case, then it should garbage collect $\widehat{l}$ and update $\widehat{\phi}$ to remove $\widehat{l}$ from $\widehat{\phi}(\widehat{l}')$ for every $\widehat{l}'$ where $\widehat{l} \mathrel{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}'} \widehat{l}'$, and recursively apply the same check for every such $\widehat{l}'$. Finally, $\widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}')$ returns a new state $\widehat{\varsigma}''$ by removing all garbage from $\widehat{\sigma}_{\widehat{\varsigma}'}$ and $\widehat{\sigma_k}_{\widehat{\varsigma}'}$ and by updating $\widehat{\phi}_{\widehat{\varsigma}'}$ to take into account the removed references.

One can verify from the definitions of $(\widehat{\rightarrow}_0)$ and $\widehat{\mathsf{collect}}$ that the $\widehat{\phi}$-component is always updated correctly. That is, for every state $\widehat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc}}$, $(\widehat{\rightarrow}_{\mathsf{arc}})$, they maintain the invariant $\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l}) = \{\widehat{l}' \mid \widehat{l}' \mathrel{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l}\}$, so that we can reason about $\widehat{\phi}(\widehat{l})$ as the set of addresses that refer to $\widehat{l}$.

3.2 Properties of $(\widehat{\rightarrow}_{\mathsf{arc}})$

We now show that – in the absence of cycles – $(\widehat{\rightarrow}_{\mathsf{arc}})$ is garbage-free, which means that abstract reference counting results in a garbage-free abstract interpreter.

First, we show that the garbage collection is *sound*. In terms of abstract garbage collection, soundness means that addresses that are reachable are not removed from the store [42]. Using the definitions of Section 2.2, this means that for every state $\widehat{\varsigma}$ it collects *only* unreachable addresses $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$, so that $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$. Intuitively, this implies that the garbage collection is safe, and never removes a binding that is still needed.

▶ **Definition 4** (GC Soundness). *A state $\widehat{\varsigma}$ is sound iff $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$. A transition relation $(\widehat{\rightarrow})$ is sound iff it preserves soundness, i.e., if $\widehat{\varsigma}$ is sound and $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}} \widehat{\varsigma}'$, then $\widehat{\varsigma}'$ is sound.*

▶ **Lemma 5.** *$(\widehat{\rightarrow}_{arc})$ is sound.*

Proof. The proof is detailed in Appendix A. ◀

We refer to the dual of sound garbage collection as *complete* garbage collection. In the context of abstract GC, completeness means that all addresses that are in the store are still reachable. For every state $\widehat{\varsigma}$, a complete garbage collector collects *all* addresses $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$, so that $\widehat{\mathcal{S}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma})$. Intuitively, this implies that the garbage collection never misses any garbage, only keeping non-garbage values in the store at all times. We show that $(\widehat{\rightarrow}_{\mathsf{arc}})$ is GC complete in the absence of cycles.

▶ **Definition 6** (GC Completeness). *A state $\widehat{\varsigma}$ is complete iff $\widehat{\mathcal{S}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma})$. A relation $(\widehat{\rightarrow})$ is complete iff it preserves completeness, i.e. if $\widehat{\varsigma}$ is complete and $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}} \widehat{\varsigma}'$, then $\widehat{\varsigma}'$ is complete.*

▶ **Lemma 7.** *In the absence of cycles, $(\widehat{\rightarrow}_{arc})$ is complete.*

Proof. The proof is detailed in Appendix A. ◀

Note that combining GC soundness with GC completeness yields the garbage-free property. That is, if a state $\widehat{\varsigma}$ is both sound and complete, we have that $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$ and $\widehat{\mathcal{S}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma})$, hence $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \widehat{\mathcal{S}}(\widehat{\varsigma})$ so that $\widehat{\varsigma}$ by definition is garbage-free. An abstract interpreter that applies abstract tracing GC (as in Section 2.2), but not at every step, is GC sound, but not GC complete, and therefore not garbage-free. Similarly, abstract reference counting in the presence of cycles is still GC sound (Lemma 5), but not GC complete, nor garbage-free.

▶ **Theorem 8.** *In the absence of cycles, $(\widehat{\rightarrow}_{arc})$ is garbage-free.*

Proof. Follows immediately from Lemma 5 and Lemma 7. ◀

We have now shown that – in the absence of cycles – using $(\widehat{\rightarrow}_{\mathsf{arc}})$ as a transition relation results in a garbage-free abstract interpreter without the need to trigger a full tracing GC at every step. Normally, we would still have to prove other properties of $(\widehat{\rightarrow}_{\mathsf{arc}})$ (such as the soundness of the abstract interpretation); instead, we prove an equivalence with the existing transition relation $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, for which such properties have already been established in related work [23, 42]. First, we introduce a function $\widehat{\mathsf{rc}} : \widehat{\Sigma} \to \widehat{\Sigma}_{\mathsf{arc}}$ to map states from the abstract interpreter in Section 2.2 to equivalent states in $\widehat{\Sigma}_{\mathsf{arc}}$ with the following definition:

$$\widehat{\mathsf{rc}}(\widehat{\varsigma}) = \langle e_{\hat{\varsigma}}, \widehat{\rho}_{\hat{\varsigma}}, \widehat{\sigma}_{\hat{\varsigma}}, \widehat{\sigma_k}_{\hat{\varsigma}}, \widehat{a}_k, \widehat{\phi} \rangle \qquad \text{where } \widehat{\phi}(\widehat{l}) = \{\widehat{l}' \in \widehat{\mathsf{Loc}} \mid \widehat{l}' \mathrel{\widehat{\rightsquigarrow}}_{\hat{\varsigma}} \widehat{l}\}$$

▶ **Theorem 9.** *In the absence of cycles, the transition relation* $(\widehat{\rightarrow}_{\mathsf{arc}})$ *is equivalent to* $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ *in the sense that:* $\forall \widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\Sigma},\ \widehat{\varsigma} \mathrel{\widehat{\rightarrow}}_{\widehat{\Gamma}} \widehat{\varsigma}' \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}}_{arc} \widehat{\mathsf{rc}}(\widehat{\varsigma}')$.

Proof. The proof is detailed in Appendix A. ◀

As a consequence, we can design a garbage-free abstract interpreter using $(\widehat{\rightarrow}_{\mathsf{arc}})$ by defining

$$\widehat{\mathsf{eval}}_{\mathsf{arc}}(e) = \{\widehat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc}} \mid \langle e, [], \bot_{\widehat{\sigma}}, \bot_{\widehat{\sigma_k}}, \widehat{a}_{\mathsf{halt}}, \bot_{\widehat{\phi}} \rangle \mathrel{\overset{*}{\widehat{\rightarrow}}}_{\mathsf{arc}} \widehat{\varsigma}\}$$

so that for every program e that, in the absence of cycles, $\widehat{\mathsf{eval}}_{\mathsf{arc}}(e) = \left\{\widehat{\mathsf{rc}}(\widehat{\varsigma}) \mid \widehat{\varsigma} \in \widehat{\mathsf{eval}}(e)\right\}$.

4 Reclaiming Garbage Cycles

In the previous section, we showed that the $(\widehat{\rightarrow}_{\mathsf{arc}})$ transition relation is garbage-free under the premise that no states are encountered with cycles in their store or continuation store. The underlying reason for this limitation is that pure reference counting cannot reclaim cycles [50]. However, it is clear that this premise does not hold in general. A concrete interpreter can end up with cycles in the store σ when a program creates cyclic data structures. In this case, abstract interpretation of this program will also result in a cyclic structure in $\widehat{\sigma}$, which will never be reclaimed using $(\widehat{\rightarrow}_{\mathsf{arc}})$. Hence, in general the transition relation $(\widehat{\rightarrow}_{\mathsf{arc}})$ is only GC sound, not GC complete and therefore also not garbage-free.

4.1 Artificial Cycles in $(\widehat{\rightarrow}_{\mathsf{arc}})$

To make matters worse, we can show that cycles are much more common in the abstract. That is, for a given program e, an abstract interpreter may encounter states with cycles in either $\widehat{\sigma}$ (or $\widehat{\sigma}_k$) that would not occur in σ (or σ_k) in states explored by a concrete interpreter. The underlying reason is that an abstract interpreter can only use a finite number of addresses, and therefore is sometimes forced to reuse an address for different allocations.

From a theoretical perspective, this implies that an abstract address ($\in \widehat{\mathsf{Loc}}$) is used as the abstraction for multiple concrete addresses ($\in \mathsf{Loc}$). We assume that abstraction function $\alpha : \mathsf{Loc} \to \widehat{\mathsf{Loc}}$ maps a concrete address l to the abstract address $\widehat{l}$, while concretization function $\gamma : \widehat{\mathsf{Loc}} \to \mathcal{P}(\mathsf{Loc})$ returns all concrete addresses that are abstracted to an abstract address. It can be shown from the abstract semantics [23] that if l_1 references l_2 ($l_1 \rightsquigarrow l_2$) under concrete interpretation, then $\alpha(l_1)$ references $\alpha(l_2)$ under abstract interpretation too ($\alpha(l_1) \mathrel{\widehat{\rightsquigarrow}} \alpha(l_2)$). Now consider the linear sequence of concrete references $l_0 \rightsquigarrow l_1 \rightsquigarrow \ldots \rightsquigarrow l_k$ with $l_0 \neq l_1 \neq \ldots \neq l_k$. Its abstraction $\alpha(l_0) \mathrel{\widehat{\rightsquigarrow}} \alpha(l_1) \mathrel{\widehat{\rightsquigarrow}} \ldots \mathrel{\widehat{\rightsquigarrow}} \alpha(l_k)$ becomes cyclic as soon as for some $i \neq j$, $\alpha(l_i) = \alpha(l_j)$. We refer to the cycles that lack a concrete counterpart and stem from the interpreter's abstraction as *artificial cycles*.

In practice, it turns out that artificial cycles are mostly a problem in the continuation store. Consider the E-Let transition rule. This transition rule "pushes" a continuation on the continuation store, so that if $\varsigma \to \varsigma'$, we have that $a_{k\varsigma'} \rightsquigarrow a_{k\varsigma}$. As kalloc always returns a fresh address under concrete interpretation, a sequence of transitions of this rule will give rise to $a_{kn} \rightsquigarrow a_{kn-1} \rightsquigarrow \ldots \rightsquigarrow a_{k0}$ where $a_{kn} \neq a_{kn-1} \neq \ldots \neq a_{k0}$. Under abstract interpretation, in contrast, the same sequence will result in $\alpha(a_{kn}) \mathrel{\widehat{\rightsquigarrow}} \alpha(a_{kn-1}) \mathrel{\widehat{\rightsquigarrow}} \ldots \mathrel{\widehat{\rightsquigarrow}} \alpha(a_{k0})$. A cycle will be created as soon as $\widehat{\mathsf{kalloc}}$ allocates the same address more than once in such a sequence (i.e., once $\alpha(a_{ki}) = \alpha(a_{kj})$ for some $0 \leq i < j \leq n$). This is usually the case when the abstract interpreter visits the same program location multiple times due to, for instance, looping behavior. When the resulting cycle becomes unreachable, none of its constituent addresses will be reclaimed under $(\widehat{\rightarrow}_{\mathsf{arc}})$ from the continuation store where they cause imprecisions. While imprecision in the value store $\widehat{\sigma}$ may be tolerable, imprecision in the continuation store $\widehat{\sigma}_k$ has a detrimental impact. The E-Return transition rule, for instance, will generate *spurious* successor states if set $\widehat{\sigma}_k(\widehat{a}_k)$ contains "garbage continuations" due to such imprecision resulting in the exploration of infeasible control flow with an impact on both precision and performance.

4.2 Abstract Reference Counting with Cycle Detection: $(\widehat{\rightarrow}_{\mathsf{arc++}})$

We extend $(\widehat{\rightarrow}_{\mathsf{arc}})$ so that it remains GC complete in the presence of cycles. Existing techniques for reference counting to reclaim cycles do not immediately help in this setting (cf. Section 6). We therefore propose a domain-specific solution that is tailored towards the artificial cycles that are artefacts of abstraction. First, we make a few observations:

- As discussed above, cycles are the most prevalent in the continuation store. Not reclaiming such cycles impacts precision and performance negatively. Our main priority is therefore the continuation store. While realizing the garbage-free property requires cycle detection in both stores, we show that its application to the continuation store does not have a significant performance overhead (cf. Theorem 10).
- An artificial cycle can only be created when an address is reused, and address reuse often results in a cycle in practice. In a garbage-free abstract interpreter, reusing an address in the continuation store is guaranteed to create a cycle.
- The transition rules in $(\widehat{\rightarrow}_0)$ only make the stores grow monotonically; elements are only removed from the store when addresses are garbage collected. As a consequence, cycles never "break up", because if some address that is part of a cycle is collected as garbage, then by the definition of garbage the entire cycle should be collected.

Hence, while it is not obvious how to detect when a cycle becomes a garbage cycle, it is possible to predict the creation of a cycle efficiently. Therefore, our approach explicitly tracks all cycles – or rather *strongly connected components* (SCC) – in $\widehat{\mathcal{S}}(\widehat{\varsigma})$. Such cycles can only grow when two SCCs are merged together, or be removed in their entirety when collected as garbage, enabling maintaining a disjoint-set of the SCCs in $\widehat{\mathcal{S}}(\widehat{\varsigma})$ efficiently. The idea is then to maintain the reference count for every SCC, only counting "external" references coming from other SCCs. When a SCC loses all of its external references, all addresses that are part of that SCC can be garbage collected. As a cycle can by definition never occur between two SCCs, applying reference counting at this level overcomes its main limitation.

Figure 9 depicts the updates to the state space $\widehat{\Sigma}_{\mathsf{arc}}$ that incorporate cycle detection. Newly-added component $\widehat{\pi}$ tracks the partitioning of $\widehat{\mathcal{S}}(\widehat{\varsigma})$ into its SCCs.

$$\widehat{\varsigma} \in \widehat{\Sigma}_{\text{arc+}} = \mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \times \widehat{\mathsf{KStore}} \times \widehat{\mathsf{KAddr}} \times \widehat{\mathsf{Refs}} \times \widehat{\mathsf{DS}}$$

$$\widehat{s}, \widehat{scc} \in \widehat{\mathsf{SCC}} = \mathcal{P}(\widehat{\mathsf{Loc}})$$

$$\widehat{\phi} \in \widehat{\mathsf{Refs}} = \widehat{\mathsf{SCC}} \rightharpoonup \mathcal{P}(\widehat{\mathsf{Loc}})$$

$$\widehat{\pi} \in \widehat{\mathsf{DS}} = \mathcal{P}(\widehat{\mathsf{SCC}})$$

Figure 9 Updated state space of the abstract interpreter with reference counting to detect cycles. Other components preserve the original definition from Figure 7.

The abstract interpreter needs to maintain disjoint-set $\widehat{\pi}$ so that $\bigcup_{\widehat{scc} \in \widehat{\pi}} \widehat{scc} = \widehat{\mathsf{Loc}}$ and for any $\widehat{s}_1, \widehat{s}_2 \in \widehat{\pi}$ where $\widehat{s}_1 \neq \widehat{s}_2$, $\widehat{s}_1 \cap \widehat{s}_2 = \emptyset$. In what follows, we assume that functions union and find (for which efficient implementations have been proposed [19]) exist, so that:

$$\mathsf{union}(\widehat{\pi}, \widehat{s}_1, \widehat{s}_2) = (\widehat{\pi} \setminus \{\widehat{s}_1, \widehat{s}_2\}) \cup \{\widehat{s}_1 \cup \widehat{s}_2\} \qquad \mathsf{find}(\widehat{\pi}, \widehat{l}) = \widehat{scc} \text{ where } \widehat{scc} \in \widehat{\pi} \wedge \widehat{l} \in \widehat{scc}$$

The original definition of $\widehat{\phi}$ from Section 3.1 also changes in that it now tracks all incoming references *for every strongly connected component* from the other SCCs. Note again that the state space does not increase in complexity, since both $\widehat{\phi}_{\widehat{\varsigma}}$ and $\widehat{\pi}_{\widehat{\varsigma}}$ are defined deterministically in terms of $\widehat{\varsigma}$'s other components:

$$\widehat{\phi}_{\widehat{\varsigma}}(\widehat{scc}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l'} \notin \widehat{scc} \wedge \exists \widehat{l} \in \widehat{scc}, \widehat{l'} \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}}} \widehat{l}\} \text{ for } \widehat{scc} \in \widehat{\pi}_{\widehat{\varsigma}} \qquad \bot_{\widehat{\phi}} = \lambda \widehat{scc}.\, \emptyset$$

$$\forall \widehat{l}_1, \widehat{l}_2 \in \widehat{\mathsf{Loc}},\ \mathsf{find}(\widehat{\pi}_{\widehat{\varsigma}}, \widehat{l}_1) = \mathsf{find}(\widehat{\pi}_{\widehat{\varsigma}}, \widehat{l}_2) \iff (\widehat{l}_1 \mathrel{\overset{*}{\widehat{\rightsquigarrow}_{\widehat{\varsigma}}}} \widehat{l}_2 \wedge \widehat{l}_2 \mathrel{\overset{*}{\widehat{\rightsquigarrow}_{\widehat{\varsigma}}}} \widehat{l}_1) \qquad \bot_{\widehat{\pi}} = \{\{\widehat{l}\} \mid \widehat{l} \in \widehat{\mathsf{Loc}}\}$$

Transition relation $(\widehat{\rightarrow_0})$ of Figure 8 needs to be updated to maintain the $\widehat{\pi}$ and $\widehat{\phi}$ components. Previously in $(\widehat{\rightarrow_0})$, whenever a reference from address $\widehat{l}_{from}$ to a set of addresses $\widehat{S}$ was added to the store, $\widehat{\phi}$ was updated to $\widehat{\phi} \sqcup \bigsqcup_{\widehat{l}_{to} \in \widehat{S}} [\widehat{l}_{to} \mapsto \{\widehat{l}_{from}\}]$. For the new auxiliary transition relation $(\widehat{\rightarrow_0})$, we replace this update with a call to the function $\textsc{extend}(\widehat{l}_{from}, \widehat{S}, \widehat{\phi}, \widehat{\pi})$. The full definition of $(\widehat{\rightarrow_{0++}})$ can be found in Appendix B.

The $\textsc{extend}$ function in Algorihm 1 checks if SCCs are merged together when either the store or continuation store is extended. For every new reference from $\widehat{l}_{from}$ to $\widehat{l}_{to}$ that is added to the store, it calls the function $\textsc{update}$ to detect if a new cycle has been created and to update $\widehat{\phi}$ and $\widehat{\pi}$ accordingly. The $\textsc{update}$ function initiates a backward search for the SCC of $\widehat{l}_{to}$, starting from the SCC of $\widehat{l}_{from}$. All strongly connected components that can be traversed in a path from $\widehat{l}_{from}$ to $\widehat{l}_{to}$ need to be merged together in $\widehat{\pi}$ (using union), since such a path implies that a cycle is created as $\widehat{l}_{to} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}} \widehat{l}_{from}$ *and* $\widehat{l}_{from} \mathrel{\widehat{\rightsquigarrow}} \widehat{l}_{to}$ (due to the new reference from $\widehat{l}_{from}$ to $\widehat{l}_{to}$ that was added to the store). During this traversal, $\textsc{update}$ also keeps track of all incoming references to this newly created SCC, so that it can immediately update $\widehat{\phi}$ as well. If no such path is found, $\widehat{l}_{from}$ and $\widehat{l}_{to}$ are not part of the same SCC, so that $\widehat{\pi}$ remains unchanged and $\widehat{\phi}$ is updated as in $(\widehat{\rightarrow_0})$. Note that the check $\widehat{scc} \in V$ is only there to prevent redundant work when a SCC has already been visited through a different path; its purpose is *not* to prevent an infinite looping of $\textsc{search}$, which can not happen as there are no cyclic paths between SCCs.

Of course, performing cycle detection through a search comes at a cost, especially when looking at the theoretical worst-case behaviour. In practice, however, the advantages of being able to reclaim garbage cycles appear to outweigh this cost (cf. Section 5.2). Note that the backward search is only initiated when an address is being reused, as for a fresh address $\widehat{l}$, there are no incoming references to traverse (i.e., $\widehat{\phi}(\mathsf{find}(\widehat{\pi}, \widehat{l})) = \emptyset$). As address reuse often creates a cycle, incorporating cycle detection usually pays off. In fact, for the continuation store, we can state this more formally in Theorem 10.

Algorithm 1: Cycle detection in $(\widehat{\rightarrow}_{0++})$.

```
function UPDATE(l̂_from, l̂_to, φ̂, π̂):
    V ← ∅ ; I ← φ̂(find(π̂, l̂_to))
    function SEARCH(l̂):
        ŝcc ← find(π̂, l̂)
        if ŝcc = find(π̂, l̂_to) then return true
        if ŝcc ∈ V then return false
        V ← V ∪ {ŝcc}
        reachable ← false
        incoming ← ∅
        foreach l̂' ∈ φ̂(ŝcc) do
            if SEARCH(l̂') then reachable ← true
            else incoming ← incoming ∪ {l̂'}
        end
        if reachable then
            π̂ ← union(ŝcc, find(π̂, l̂_to))
            I ← I ∪ incoming
        end
        return reachable
    if SEARCH(l̂_from) then
        return ⟨φ̂|_π̂ ⊔ [find(π̂, l̂_to) ↦ I], π̂⟩
    else
        return ⟨φ̂ ⊔ [find(π̂, l̂_to) ↦ {l̂_from}], π̂⟩
function EXTEND(l̂_from, Ŝ, φ̂, π̂):
    foreach l̂_to ∈ Ŝ do
        if l̂_from ∉ φ̂(find(π̂, l̂_to)) then
            ⟨φ̂, π̂⟩ ←UPDATE(l̂_from, l̂_to, φ̂, π̂)
    end
    return ⟨φ̂, π̂⟩
```

▶ **Theorem 10.** *In a garbage-free abstract interpreter, extending abstract reference counting with cycle detection for the continuation store only requires amortized $\mathcal{O}(1)$ additional operations per continuation that is added to the continuation store.*

Proof. The proof is detailed in Appendix A. ◀

The intuition here is that every reference that is inserted into $\widehat{\sigma_k}$ can only be traversed once during the backward search of the cycle detection: once traversed, we can show that it is guaranteed to become part of a cycle (cf. proof of Theorem 10 in Appendix A). Since the internal references of a cycle are not traversed by the backward search, it can no longer add to the cost of a future traversal once the SCCs are merged. That is, when a backward search is triggered in the continuation store, we are guaranteed that all traversed edges between SCCs will become part of the same SCC after cycle detection, hence the path is shortened for future traversals. In practical terms, Theorem 10 states that cycle detection can be applied to the continuation store "for free", i.e., without much of an additional cost in performance.

Finally, we can define $(\widehat{\rightarrow}_{\mathsf{arc}++})$ as we did previously for $(\widehat{\rightarrow}_{\mathsf{arc}})$:

$$\frac{\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{0++}} \widehat{\varsigma}' \qquad \widehat{\varsigma}'' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}')}{\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\mathsf{arc}++}} \widehat{\varsigma}''}$$

where $\widehat{\mathsf{collect}}$ remains mostly unchanged, save for some trivial modifications to work at the level of SCCs. We refer to Appendix B for the updated definition of $\widehat{\mathsf{collect}}$.

Unlike $(\widehat{\rightarrow}_{\mathsf{arc}})$, we can show that using the transition relation $(\widehat{\rightarrow}_{\mathsf{arc++}})$ results in a garbage-free abstract interpreter, even in the presence of cycles. This is stated by Theorem 11.

▶ **Theorem 11.** $(\widehat{\rightarrow}_{arc++})$ *is garbage-free.*

Proof. The proofs of Lemma 5 and 7 can be repeated for $(\widehat{\rightarrow}_{\mathsf{arc++}})$, replacing addresses with their strongly connected components where necessary. The premise of Lemma 7 that disallows cycles can then be omitted, since cycles by definition do not occur between SCCs. ◀

Moreover, we can now claim full equivalence of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ and $(\widehat{\rightarrow}_{\mathsf{arc++}})$. First, define $\widehat{\mathsf{rc}}$ as:

$$\widehat{\mathsf{rc}}(\widehat{\varsigma}) = \langle e_{\widehat{\varsigma}}, \widehat{\rho}_{\widehat{\varsigma}}, \widehat{\sigma}_{\widehat{\varsigma}}, \widehat{\sigma_{k\widehat{\varsigma}}}, \widehat{a}_k, \widehat{\phi}, \widehat{\pi}\rangle \text{ where}$$
$$\widehat{\phi}(\widehat{scc}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l'} \notin \widehat{scc} \wedge \exists \widehat{l} \in \widehat{scc}, \widehat{l'} \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}}} \widehat{l}\} \text{ for } \widehat{scc} \in \widehat{\pi}$$
$$\forall \widehat{l_1}, \widehat{l_2} \in \widehat{\mathsf{Loc}},\ \mathsf{find}(\widehat{\pi}, \widehat{l_1}) = \mathsf{find}(\widehat{\pi}, \widehat{l_2}) \iff (\widehat{l_1} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{l_2} \wedge \widehat{l_2} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{l_1})$$

▶ **Theorem 12.** $(\widehat{\rightarrow}_{arc})$ *is equivalent to* $(\widehat{\rightarrow}_{\widehat{\Gamma}}) : \forall \widehat{\varsigma}, \widehat{\varsigma'} \in \widehat{\Sigma}, \widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}} \widehat{\varsigma'} \iff \widehat{rc}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_{arc++}} \widehat{rc}(\widehat{\varsigma'})$.

Proof. The proof is analogous to that of Theorem 9. ◀

5 Evaluation

We have already proven the GC soundness and GC completeness of our approach (Lemmas 5 and 7), as well as a theoretical efficiency claim of the cycle detection technique for the continuation store (Theorem 10). We now present the empirical evaluation. Section 5.2 measures the impact of the cycle detection technique on abstract reference counting, while Section 5.3 compares its precision and performance to existing approaches to abstract GC.

5.1 Experimental Setup

We ran all experiments using Scala version 2.12.3 on a server with a 3.5GHz Intel Xeon 2637 processor and 256GB of RAM. We use a similar setup throughout our evaluation.

Implementation. We implemented both abstract reference counting and the existing variants of abstract tracing GC (surveyed in Section 1.2) using the Scala-AM framework for abstract interpretation of Scheme programs [54, 53]. This resulted in 5 abstract interpreters[5], each incorporating the Scheme equivalent of the λ_{ANF} transition relations $(\widehat{\rightarrow})$ and $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ (cf. Section 2), $(\widehat{\rightarrow}_{\mathsf{arc}})$ (cf. Section 3) and $(\widehat{\rightarrow}_{\mathsf{arc++}})$ (cf. Section 4), as well as the additional $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ and $(\widehat{\rightarrow}_{\mathsf{arc+}})$ relations defined below:

- $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ adopts the GC policy of ΓCFA [42], which applies abstract GC whenever values need to be joined in the store, so that values are by design never merged with garbage. While this approach is not garbage-free (cf. Section 1.2), it is interesting to examine how this common policy affects the precision and performance of the abstract interpreter.
- $(\widehat{\rightarrow}_{\mathsf{arc+}})$ adopts abstract reference counting, but only applies cycle detection to the continuation store $\widehat{\sigma}_k$. As garbage cycles will remain in value store $\widehat{\sigma}$, this relation is not garbage-free. However, its empirical evaluation is motivated by our observation that garbage cycles are mainly an issue in the continuation store, and by our proof for Theorem 10 that cycle detection should have no major performance overhead there.

[5] Publicly available at `https://github.com/noahvanes/scala-am-abstractgc`

Benchmarks. We evaluate the above implementations using a total of 16 Scheme programs; 11 from the Gabriel benchmark suite [18][6] and 5 from the built-in test suite of Scala-AM. Table 1 lists the size of each Scheme program, with the Gabriel ones depicted in **bold**. For each benchmark program, we configure each abstract interpreter with a monovariant allocator (i.e., 0CFA), and a lattice that abstracts concrete values to the set of their possible types.

Table 1 Lines of code (LOC) for each benchmark. The Gabriel benchmarks are in **bold**.

Benchmark	LOC	Benchmark	LOC	Benchmark	LOC	Benchmark	LOC
boyer	568	**deriv**	39	**takl**	18	gcipd	12
browse	78	**destruc**	37	**puzzle**	144	primtest	35
cpstak	19	**diviter**	8	**triangl**	40	rsa	41
dderiv	83	**divrec**	11	collatz	22	nqueens	34

Measurements. As the implementations differ only in their approach to abstract garbage collection, we follow Earl et al. [16] in the use of the number of explored states as a measure of *precision*. As garbage can cause an abstract interpreter to explore spurious states (cf. Section 1.2), fewer states implies higher precision when all configuration parameters are kept constant. For *performance*, we measure the total running time of the abstract interpreter. Note that running time is impacted by two factors. First, the number of number states the interpreter has to explore. Higher precision can often lead to higher performance. Second, the rate at which the abstract interpreter is able to explore those states. This rate is influenced by the *overhead* caused by abstract garbage collection, which we therefore also measure per state. For the tracing GC approaches ($(\widehat{\rightarrow}_{\widehat{\Gamma}})$ and $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$), this measure corresponds to the time spent in the function $\widehat{\Gamma}$. For approaches based on abstract reference counting ($(\widehat{\rightarrow}_{\mathsf{arc}})$, $(\widehat{\rightarrow}_{\mathsf{arc+}})$ and $(\widehat{\rightarrow}_{\mathsf{arc++}})$), this measure includes the time spent on updating the $\widehat{\phi}$ and $\widehat{\pi}$ components (including cycle detection for $(\widehat{\rightarrow}_{\mathsf{arc+}})$ and $(\widehat{\rightarrow}_{\mathsf{arc++}})$) and in the $\widehat{\mathsf{collect}}$ function. We ran every benchmark 30 times after a warm-up period of 2 minutes. For the time-sensitive measurements, we report the mean of all runs.

5.2 Impact of Cycle Detection

Section 3 introduced transition relation $(\widehat{\rightarrow}_{\mathsf{arc}})$ which performs abstract reference counting, but cannot reclaim cycles. Section 4 extended this relation into $(\widehat{\rightarrow}_{\mathsf{arc++}})$, which can reclaim cyclic garbage by applying cycle detection to both stores. Transition relation $(\widehat{\rightarrow}_{\mathsf{arc+}})$, introduced above, is an in-between: it uses abstract reference counting, but only applies cycle detection to the continuation store. Table 2 compares the corresponding abstract interpreters to evaluate the impact of cycle detection on abstract reference counting.

Impact on precision. The results from Table 2 show that reclaiming garbage cycles from the continuation store only, can already result in important precision improvements. For most benchmarks, $(\widehat{\rightarrow}_{\mathsf{arc+}})$ nearly achieves the same optimal precision as $(\widehat{\rightarrow}_{\mathsf{arc++}})$. For instance, all of the non-Gabriel benchmarks significantly improve precision with cycle detection for the continuation store only, and adding cycle detection to the value store does not increase

[6] We omitted the *ctak* benchmark program due to its use of `call/cc`, which is not yet supported by the abstract interpretation framework.

precision any further. In addition, for the *triangl* benchmark, not reclaiming garbage cycles from the continuation store (using ($\widehat{\rightarrow}_{\mathsf{arc}}$)) causes the state space to blow up. However, it is clear that reclaiming them from the continuation store does not always suffice: benchmarks *destruc* and *puzzle* both create cycles in the value store, which are not reclaimed by ($\widehat{\rightarrow}_{\mathsf{arc}}$) nor ($\widehat{\rightarrow}_{\mathsf{arc+}}$). Programs that are written in CPS-style, such as *cpstak*, do not benefit from the extra precision in the continuation store either, as their control flow is encoded in closures that reside in the value store. The garbage-free transition relation ($\widehat{\rightarrow}_{\mathsf{arc++}}$) improves precision more consistently, due to its reclamation of all garbage cycles from both stores. These results underline that cycles cannot be neglected in abstract reference counting, and that a technique for their detection is required to realize the benefits of garbage-free abstract interpretation.

Table 2 Comparison of time taken in milliseconds (t; lower means better performance) and number of states explored (#s; lower means better precision). A time of ∞ means that the benchmark exceeded the time limit of 30 minutes; in this case, we report the number of states explored when the timeout was reached.

	($\widehat{\rightarrow}_{\mathsf{arc}}$)		($\widehat{\rightarrow}_{\mathsf{arc+}}$)		($\widehat{\rightarrow}_{\mathsf{arc++}}$)	
	t	#s	t	#s	t	#s
cpstak	1	94	1	94	1	88
diviter	13	163	14	163	14	162
divrec	15	153	14	146	16	145
destruc	1 208	61 021	1 160	61 021	304	17 344
triangl	∞	>2 186 761	2 044	2 639	1 613	2 639
puzzle	23 950	1 060 585	22 142	1 018 345	2 351	117 572
takl	14 375	862 833	3 581	188 552	2 791	188 549
browse	∞	>4 028 742	∞	>2 586 788	∞	>2 559 221
boyer	∞	>18 544 242	∞	>12 690 008	∞	>12 715 066
deriv	∞	>8 650 787	∞	>28 905 027	∞	>29 513 647
dderiv	∞	>11 561 462	∞	>32 582 390	∞	>32 803 534
collatz	1	60	1	60	1	60
gcipd	1	151	1	91	1	91
primtest	40	5 347	11	1 537	12	1 537
rsa	87	10 646	12	1 538	12	1 538
nqueens	1 207	112 579	319	31 883	338	31 857

Impact on performance. Figure 10 compares the time spent on GC per computed state for the reference-counting transition relations. For most benchmarks, we observe that cycle detection increases the GC overhead slightly. However, this overhead is negligible compared to the corresponding improvements to precision. That is, the overheads in Figure 10 remain comparable for different configurations on each benchmark, while the number of states explored reported in Table 2 decreases significantly when using cycle detection. We observe proportionally large improvements to performance, as the abstract interpreter has fewer states to explore. Again, the garbage-free transition relation ($\widehat{\rightarrow}_{\mathsf{arc++}}$) can most consistently improve performance by completely avoiding any unnecessary computation, with only a negligible increase in overhead. We conclude that the benefits of cycle detection outweigh its cost, as ($\widehat{\rightarrow}_{\mathsf{arc++}}$) is superior to both ($\widehat{\rightarrow}_{\mathsf{arc}}$) and ($\widehat{\rightarrow}_{\mathsf{arc+}}$) in terms of precision and performance.

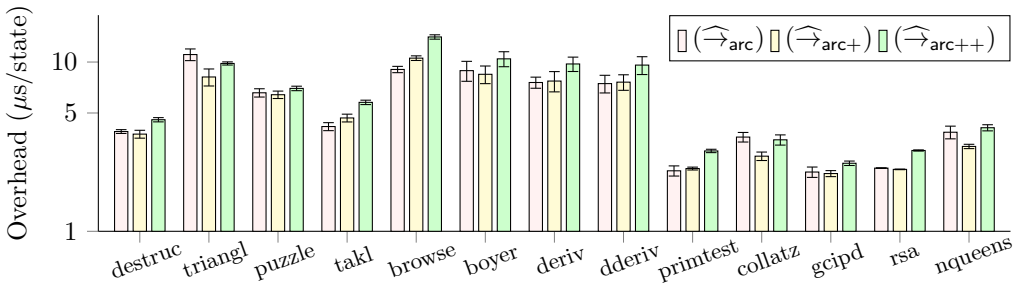


Figure 10 Comparison of overhead measured in time spent on GC per state (lower is better). Error bars indicate the standard deviation of the mean of our measurements (for 30 runs). Note the usage of a logarithmic scale on the y-axis.

Note that Table 2 also lists the number of states that were explored for the benchmarks that did time out. This gives a rough estimate of the interpreters state exploration rate. However, these rates should be interpreted with caution, as some paths in the state space can be significantly cheaper to explore than others. One should therefore only compare the rates of interpreters with the same precision, or look for a pattern of major discrepancies. We do not discern such a pattern among the rates, and deem the differences in precision too outspoken to warrant a direct rate comparison.

5.3 Comparison to Existing Policies for Abstract GC

Table 3 compares $(\widehat{\rightarrow}_{\mathsf{arc++}})$ to other approaches to abstract garbage collection. On one end of the spectrum, there is $(\widehat{\rightarrow})$, which never applies abstract GC. On the other end, there is $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, which applies abstract GC on every evaluation step. An interesting tradeoff is made by $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$, which applies GC before every store join.

Table 3 Comparison of time taken in milliseconds (t; lower means better performance) and number of states explored (#s; lower means better precision) by an abstract interpreter for different approaches to abstract GC. A time of ∞ means that the benchmark exceeded the time limit of 30 minutes; in this case, we report the number of states explored when the timeout was reached.

	$(\widehat{\rightarrow})$		$(\widehat{\rightarrow}_{\widehat{\Gamma}})$		$(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$		$(\widehat{\rightarrow}_{\mathsf{arc++}})$	
	t	#s	t	#s	t	#s	t	#s
cpstak	1	120	7	88	2	94	1	88
diviter	14	175	29	162	13	163	14	162
divrec	24	219	28	145	16	148	16	145
destruc	9 671	381 949	2 436	17 344	735	22 444	304	17 344
triangl	∞	>4 826 189	2 018	2 639	1 635	2 767	1 613	2 639
puzzle	∞	>104 243 260	24 704	117 572	14 732	293 167	2 351	117 572
takl	86 830	7 072 820	72 325	377 389	32 356	539 233	2 791	188 549
browse	∞	>6 608 260	∞	>1 928 827	∞	>2 706 491	∞	>2 559 221
boyer	∞	>27 731 639	∞	>232 889	∞	>1 456 963	∞	>12 715 066
deriv	∞	>107 328 548	∞	>5 773 009	∞	>13 262 924	∞	>29 513 647
dderiv	∞	>90 849 930	∞	>3 320 410	∞	>12 954 310	∞	>32 803 534
collatz	1	431	4	60	2	159	1	60
gcipd	3	1 098	8	91	2	147	1	91
primtest	2 249	334 944	166	1 537	75	3 622	12	1 537
rsa	1 840	247 915	176	1 538	63	2 375	12	1 538
nqueens	346 349	37 499 432	4 583	37 847	1 556	52 766	338	31 857

Comparison of precision. Our results confirm the findings of previous work [42, 43, 16, 28]: abstract GC reduces the state space to explore substantially. Looking at the number of explored states in Table 3 reveals that the interpreter without abstract GC using ($\widehat{\rightarrow}$) has the lowest precision for every benchmark compared to all other approaches. The interpreter using ($\widehat{\rightarrow}_{\Gamma\mathsf{CFA}}$) sacrifices some precision by invoking abstract GC less frequently than ($\widehat{\rightarrow}_{\widehat{\Gamma}}$). It explores more states than necessary because it is not garbage-free, which is the most noticeable on more complex programs such as *destruct*, *puzzle* and *takl*.

Our approach, ($\widehat{\rightarrow}_{\mathsf{arc++}}$), achieves the same optimal precision as ($\widehat{\rightarrow}_{\widehat{\Gamma}}$). As we have proven that both transition relations are garbage-free and equivalent, they should explore the same number of states. Note, however, that this is not the case for the *takl* benchmark – an apparent contradiction of Theorem 12. The reason is that the implementation of ($\widehat{\rightarrow}_{\mathsf{arc++}}$) for Scheme is able to collect more garbage than its formalization for λ_{ANF} in Sections 3 and 4. The definition of garbage-free only requires the absence of garbage from the stores *in between* transitions. However, more complex state transitions can create garbage and immediately bring that garbage back to life *within* a single transition. Unlike our formalization for λ_{ANF}, some of the transition rules in our implementation for Scheme both lookup and insert continuations at the same address $\widehat{a}_k$ in a single step. Abstract reference counting can automatically deallocate $\widehat{a}_k$ (and addresses in the same SCC) if no references remain to $\widehat{a}_k$ after the lookup, which possibly prevents a precision loss when $\widehat{a}_k$ (or another address in the same SCC) is reallocated in the same step. Abstract reference counting can trivially reclaim garbage within a single transition due to its continuous nature. Realizing the same effect with abstract tracing GC (i.e., using ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) and ($\widehat{\rightarrow}_{\mathsf{arc++}}$)) would require a full GC application both within and after such transitions – increasing its overhead further. We investigated the applicability of this optimization on abstract reference counting for all benchmarks, and found it limited to *takl*, *browse*, *deriv*, *dderiv* and *nqueens*. For the others, ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) and ($\widehat{\rightarrow}_{\mathsf{arc++}}$) exhibit the same precision and explore the same state space.

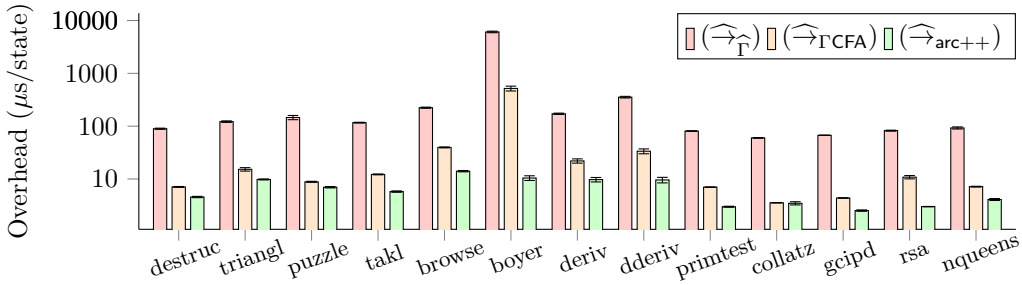


Figure 11 Comparison of overhead measured in time spent on GC per state (lower is better). Error bars indicate the standard deviation of the mean of our measurements (for 30 runs). Note the usage of a logarithmic scale on the y-axis.

Comparison of performance. Figure 11 compares the overhead of abstract GC for ($\widehat{\rightarrow}_{\widehat{\Gamma}}$), ($\widehat{\rightarrow}_{\Gamma\mathsf{CFA}}$) and ($\widehat{\rightarrow}_{\mathsf{arc++}}$). The ($\widehat{\rightarrow}$) interpreter does not suffer such overhead. However, without any abstract GC, its state space becomes polluted (leading to repeated exploration of equivalent states) and explodes (due to infeasible paths). It exhibits the worst performance.

The overhead of abstract GC is an issue for ($\widehat{\rightarrow}_{\widehat{\Gamma}}$). It is clear that a tracing GC application is generally more expensive than a single evaluation step. Applying tracing GC after every step leads to the interpreter's running time being dominated by that of the GC applications. This is outspoken on complex benchmarks that require larger heaps such as *boyer*: since

tracing GC needs to traverse the entire heap, the overhead grows with the size of the heap. For the *boyer* benchmark, the interpreter using ($\widehat{\rightarrow}_{\mathsf{arc}++}$), which explores the same state space as ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) for this benchmark, explores 12.7M states before timeout, whereas with the excessive GC overhead of ($\widehat{\rightarrow}_{\widehat{\Gamma}}$), it can only explore 233K states with the same time limit. While the garbage-freeness of ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) significantly improves overall performance of ($\widehat{\rightarrow}$) by reducing the state space, it is not as fast as ($\widehat{\rightarrow}_{\mathsf{arc}++}$) due to the higher overhead. In Figure 11, for all benchmarks the overhead of ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) differs to that of ($\widehat{\rightarrow}_{\mathsf{arc}++}$) by orders of magnitude, which results in significant performance improvements in Table 3 while exploring the same amount of states. The transition relation ($\widehat{\rightarrow}_{\Gamma\mathsf{CFA}}$) avoids this overhead of ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) by applying abstract GC less frequently. As such, its overhead is significantly lower than that of ($\widehat{\rightarrow}_{\widehat{\Gamma}}$). For all our benchmarks, this results in a performance improvement over ($\widehat{\rightarrow}_{\widehat{\Gamma}}$), despite having to explore a larger state space. Note that ($\widehat{\rightarrow}_{\Gamma\mathsf{CFA}}$) is no longer garbage-free and explores more states than necessary. For instance, on the *puzzle* benchmark it almost explores 3 times as many states as ($\widehat{\rightarrow}_{\mathsf{arc}++}$). In addition, Figure 11 reveals that the overhead remains considerably higher than that of ($\widehat{\rightarrow}$) and even ($\widehat{\rightarrow}_{\mathsf{arc}++}$). This can also be seen in the benchmarks that timeout, where the difference in states explored is significant compared to ($\widehat{\rightarrow}$) and even ($\widehat{\rightarrow}_{\mathsf{arc}++}$). In particular, the *boyer* benchmark also shows that it still scales poorly towards larger heap sizes, an inherent downside of tracing GC approaches. Since both overhead and number of states explored are higher than for ($\widehat{\rightarrow}_{\mathsf{arc}++}$), the overall performance is worse.

In general, when determining an application policy for abstract tracing GC, sacrificing garbage-freeness for increased performance implies lower precision. Using abstract reference counting with ($\widehat{\rightarrow}_{\mathsf{arc}++}$) gives the best of both worlds. Its low overhead is not detrimental to performance: although there is definitely some compared to ($\widehat{\rightarrow}$), it is consistently lower than all other approaches that actually apply abstract GC. Compared to the equivalent transition relation ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) with optimal precision, overhead is reduced by orders of magnitude. The overall performance of the *triangl* benchmark is an exception to the rule: even though the GC overhead is greatly reduced when using ($\widehat{\rightarrow}_{\mathsf{arc}++}$), its running time is mostly dominated by expensive evaluation steps, but even in this case none of the other interpreters outperform ($\widehat{\rightarrow}_{\mathsf{arc}++}$). Since ($\widehat{\rightarrow}_{\mathsf{arc}++}$) achieves the optimal precision, so that overall performance is not impacted by the exploration of spurious states. Hence, our experiments show that ($\widehat{\rightarrow}_{\mathsf{arc}++}$) is superior both in terms of precision and performance.

6 Related Work

The focus of our work is a more efficient approach to abstract garbage collection. It relates to previous work on concrete GC, abstract GC, and abstract interpretation in general.

Reference Counting. Our approach is based on the original formulation of reference counting [10], without any common optimizations [50] such as deferred [15] and coalesced [35] reference counting or "lazy freeing" [8]. As these optimizations postpone the reclamation of garbage, they cannot be used for garbage-free abstract interpretation.

We perform abstract reference counting at the level of SCCs, so that garbage cycles can be reclaimed. For concrete reference counting, two major techniques have been proposed to handle cycles: using a back-up tracing garbage collector [56] and trial deletion [9]. We deem both too expensive for frequent applications during abstract interpretation, which is required to guarantee garbage-freeness. Nevertheless, it could be interesting to investigate a variant of abstract reference counting where abstract tracing GC is applied at the evaluation steps that might render a cycle into a garbage cycle. In the concrete, this has been shown [37] to occur when a memory location loses some, but not all of its incoming references. A cheap pre-analysis as in [3] could identify the structures guaranteed to be acyclic beforehand.

Confusingly, the term "abstract reference counting" has been used with different meanings. Hudak [24] formalizes reference counting for a concrete interpreter. After abstraction, he obtains a static analysis that approximates the *concrete* reference count, enabling program optimizations [25, 45] and compile-time GC [32]. The analysis exemplifies static analysis *of* reference counting, rather than static analysis analysis *with* reference counting for the purpose of improving precision and performance. Finally, Jones et al. [31] use the term to abstract over the low-level details concrete implementations differ on.

Tracing Garbage Collection. Our main issue with tracing GC is that it requires frequent heap traversals to keep the abstract interpreter garbage-free. Various techniques exist for concrete interpreters to avoid the correspondingly long GC pauses and to improve their throughput. Incremental algorithms, such as Baker's incremental stop-and-copy algorithm [33], only remove garbage lazily, and can therefore not be used in our setting. Moreover, they would need to be redesigned for abstract interpreters which do not always allocate a fresh address. Generational algorithms [36] can increase throughput by limiting heap traversals to the young generations only. However, as garbage also arises in older ones, it does not suffice to keep the abstract interpreter garbage-free. A full GC would be required to prevent precision loss for addresses that are re-allocated in the old generation.

Abstract Garbage Collection. The ΓCFA analysis [42, 43] pioneered the concept of abstract garbage collection. The term "garbage-free" was later coined in [23] to describe an abstract interpreter that applies abstract GC at every evaluation step, eliminating all garbage. ΓCFA also incorporated *abstract counting* —not to be confused with abstract reference counting – to count how often an address has been allocated. Abstract counting combines well with abstract GC, as collecting garbage results in more precise (i.e., lower) abstract counts. It can be combined directly with abstract reference counting to the same effect.

We showed that garbage-free abstract interpretation prevents imprecision and improves the interpreter's performance by limiting exploration to garbage-free states. As such, it improves what is known as *localization* [48]; unnecessary store bindings are removed so that states with irrelevant store differences need not be explored multiple times. For similar purposes, Oh et al. [47] eliminate store bindings in their analysis based on which addresses can be used. The main difference with our and other approaches to abstract GC is the use of a pre-analysis to conservatively approximate which bindings are never used and can safely be removed as garbage. The technique can be combined with reachability-based approaches, such as abstract reference counting, to further improve localization and precision. Might et al. [41] proposed a similar extension to abstract GC, named *conditional* abstract GC, which only keeps bindings in the store that are reachable and satisfy certain conditions. However, it relies on a theorem prover and has to the best of our knowledge not yet been implemented.

In Knauel's abstract interpretation framework [34], the overhead of tracing GC turned proved performance-detrimental on complex programs. The proposed solution is a *global garbage collector* which collects garbage for multiple states at once with a global root set. Of course, this sacrifices many of the precision benefits of the GC. To our knowledge, no prior work has employed abstract reference counting as a more efficient approach to garbage-free abstract interpretation, despite reference counting being mentioned in the future work of [42]. Bergstrom et al. [6, 7] employ a primitive form of reference counting as a cheaper alternative to full abstract GC. Their control-flow analysis (without explicit store component) tracks for every variable if it has been captured by the lexical environment of another closure; if not, this variable can safely be collected once it goes out of scope. Compared to our approach,

this corresponds to abstract reference counting with reference counts from $\widehat{\mathbb{N}} = \{0, 1, \infty\}$. We have shown that such an approximation is unnecessary, as the precise reference count can be maintained without having to increase the complexity of the state space.

Abstract Interpretation. Many abstract interpreters incorporate global store widening [23] to terminate within reasonable time. Unfortunately, previous work [28] has shown that store widening and abstract GC do not combine well: abstract GC can collect fewer addresses, as it needs to account for reachability from multiple states, and store widening can no longer guarantee fast convergence, as the GC violates the monotonicity of the global store. In fact, an often overlooked consequence of abstract GC is that it no longer suffices for lattices to conform to the ascending chain condition; as GC can "descend" into lattices, the analysis is no longer monotonous, and the only way to guarantee termination is to use finite lattices.

Other work [16, 28] has been able to combine abstract GC successfully with pushdown analysis, which substantially increases control-flow precision by properly matching call and return sites. The combination yields "better-than-both-worlds" improvements, although the techniques' inherent incompatibilities need to be overcome. We leave an analogous combination with abstract reference counting open for future work.

Finally, abstract counting can be exploited to enable strong updates [38, 4]. Replacing join operations in the store with overwrites when possible increases precision. Note, however, that such strong updates break the assumption in Section 4 that stores only grow monotonically without GC. To support strong updates, our cycle detection technique therefore needs to be adapted to take into account the possibility that a SCC can break up when a strong update occurs within a single SCC (although this cannot happen in the continuation store).

Incremental Cycle Detection. To reclaim garbage cycles, our approach detects cycles as references are added to the store. We implement such *incremental cycle detection* using a simple backward search, which has suboptimal worst-case performance in theory. More advanced algorithms have been proposed [22, 5] with better asymptotic performance. However, empirical evidence suggests [52] that more complicated algorithms do not always perform better in practice for some situations. Indeed, in Section 4.2 we argued that backward searching usually works well for an abstract interpreter, and we verified this empirically in Section 5.2. Nevertheless, it would be interesting for future work to explore whether other incremental cycle detection algorithms can be more efficient for an abstract interpreter.

7 Conclusion

We have introduced abstract reference counting as a more efficient approach to abstract garbage collection. Existing approaches are based on tracing GC, which is non-continuous in nature, and therefore need to trade-off either precision or performance to strike the balance between the benefits and overhead of abstract GC. Our approach based on reference counting, in contrast, requires but minor bookkeeping yet is provably sound and complete in terms of abstract GC. The result is a garbage-free abstract interpreter that is as precise as one that applies tracing GC at every step – in the absence of cycles.

However, we showed that cycles are a major threat to the garbage-freeness of abstract reference counting. Next to those arising under concrete interpretation, the address allocator of an abstract interpreter can create many artificial cycles. We proposed cycle detection as a solution, maintaining the reference counts at the level of the strongly connected components in the stores. This enables the reclamation of garbage cycles, which is necessary to make abstract reference counting garbage-free.

In terms of precision, our empirical experiments confirm our claim on the garbage-freeness of the abstract interpreter, showing that it achieves the optimal precision for abstract garbage collection. In terms of performance, abstract reference counting – even with cycle detection – greatly reduces overhead by avoiding frequent heap traversals. Without sacrificing precision, it can therefore realize significant performance improvements in abstract interpretation.

References

1 Leif Andersen and Matthew Might. Multi-core Parallelization of Abstracted Abstract Machines. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, DC*. Citeseer, 2013.

2 David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 50–68, 2004. doi:10.1145/1028976.1028982.

3 David F. Bacon and V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pages 207–235, 2001. doi:10.1007/3-540-45337-7_12.

4 Gogul Balakrishnan and Thomas W. Reps. Recency-Abstraction for Heap-Allocated Storage. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pages 221–239, 2006. doi:10.1007/11823230_15.

5 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.

6 Lars Bergstrom. Arity raising and control-flow analysis in Manticore. Master's thesis, University of Chicago, 2009.

7 Lars Bergstrom, Matthew Fluet, Matthew Le, John H. Reppy, and Nora Sandler. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 81–93, 2014. doi:10.1145/2628136.2628153.

8 Hans-Juergen Boehm. The space cost of lazy reference counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 210–219, 2004. doi:10.1145/964001.964019.

9 Thomas W. Christopher. Reference Count Garbage Collection. *Softw., Pract. Exper.*, 14(6):503–507, 1984. doi:10.1002/spe.4380140602.

10 George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960. doi:10.1145/367487.367501.

11 Patrick Cousot. The Verification Grand Challenge and Abstract Interpretation. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 189–201, 2005. doi:10.1007/978-3-540-69149-5_21.

12 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977. doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*, pages 269–295, 1992. doi:10.1007/3-540-55844-6_142.

14 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *PACMPL*, 1(ICFP):12:1–12:25, 2017. doi:10.1145/3110256.

15 L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM*, 19(9):522–526, 1976. doi:10.1145/360336.360345.

16 Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 177–188, 2012. doi:10.1145/2364527.2364576.

17 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 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*, pages 237–247, 1993. doi:10.1145/155090.155113.

18 Richard P. Gabriel. *Performance and evaluation of LISP systems*, volume 263. MIT press Cambridge, Mass., 1985.

19 Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

20 Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 407–420, 2016. doi:10.1145/2951913.2951936.

21 Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 691–704, 2016. doi:10.1145/2837614.2837631.

22 Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012. doi:10.1145/2071379.2071382.

23 David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 51–62, 2010. doi:10.1145/1863543.1863553.

24 Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363. ACM, 1986.

25 Paul Hudak and Adrienne G. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 300–314, 1985. doi:10.1145/318593.318660.

26 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009. doi:10.1007/978-3-642-03237-0_17.

27 J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 443–454, 2013. doi:10.1145/2500365.2500604.

28 J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *J. Funct. Program.*, 24(2-3):218–283, 2014.

29 James Ian Johnson. *Automating abstract interpretation of abstract machines.* PhD thesis, Northeastern University, 2015.

30 James Ian Johnson and David Van Horn. Abstracting abstract control. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 11–22, 2014. doi:10.1145/2661088.2661098.

31 Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management.* Chapman and Hall/CRC, 2016.

32 Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 54–74. ACM, 1989.

33 Henry G. Baker Jr. List Processing in Real Time on a Serial Computer. *Commun. ACM*, 21(4):280–294, 1978. doi:10.1145/359460.359470.

34 Eric Jean Knauel. *A flow analysis framework for realistic scheme programs*. PhD thesis, University of Tübingen, Germany, 2008. URL: http://tobias-lib.ub.uni-tuebingen.de/volltexte/2008/3363/.

35 Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006. doi:10.1145/1111596.1111597.

36 Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM*, 26(6):419–429, 1983. doi:10.1145/358141.358147.

37 Alejandro D. Martínez, Rosita Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34(1):31–35, 1990.

38 Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2007. URL: http://hdl.handle.net/1853/16289.

39 Matthew Might. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 185–198, 2007. doi:10.1145/1190216.1190247.

40 Matthew Might. Abstract Interpreters for Free. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, pages 407–421, 2010. doi:10.1007/978-3-642-15769-1_25.

41 Matthew Might, Benjamin Chambers, and Olin Shivers. Model Checking Via GammaCFA. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, pages 59–73, 2007. doi:10.1007/978-3-540-69738-1_4.

42 Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 13–25, 2006. doi:10.1145/1159803.1159807.

43 Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *J. Funct. Program.*, 18(5-6):821–864, 2008. doi:10.1017/S0956796808006941.

44 Matthew Might, Yannis Smaragdakis, and David Van Horn. 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*, pages 305–315, 2010. doi:10.1145/1806596.1806631.

45 Alan Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD thesis, University of Edinburgh, UK, 1982. URL: http://hdl.handle.net/1842/6602.

46 Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Detecting function purity in JavaScript. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*, pages 101–110, 2015. doi:10.1109/SCAM.2015.7335406.

47 Hakjoo Oh and Kwangkeun Yi. Access-based abstract memory localization in static analysis. *Sci. Comput. Program.*, 78(9):1701–1727, 2013. doi:10.1016/j.scico.2013.04.002.

48 Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 296–309, 2005. doi:10.1145/1040305.1040330.

49 Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pages 399–410, 2013. doi:10.1145/2491956.2491979.

50 Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? Getting reference counting back in the ring. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, pages 73–84, 2012. doi:10.1145/2258996.2259008.

51 Olin Shivers. Control-Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 164–174, 1988. doi:10.1145/53990.54007.

52 Ragnar Lárus Sigurðsson. Practical performance of incremental topological sorting and cycle detection algorithms. Master's thesis, Chalmers University of Technology, 2016.

53 Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Building a modular static analysis framework in Scala (tool paper). In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 105–109, 2016. doi:10.1145/2998392.3001579.

54 Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. Scala-AM: A Modular Static Analysis Framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 85–90, 2016. doi:10.1109/SCAM.2016.14.

55 Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

56 Joseph Weizenbaum. Recovery of reentrant list structures in SLIP. *Commun. ACM*, 12(7):370–372, 1969. doi:10.1145/363156.363159.

A Proofs

We detail here the proofs of Theorems 1, 9 and 10, as well as the proofs of Lemmas 5 and 7.

▶ **Theorem 1.** *If* $\varsigma_0 \xrightarrow{n}_\Gamma \varsigma_1$ *and* $\varsigma_0 \xrightarrow{n} \varsigma_2$*, then* $\varsigma_1 = \Gamma(\varsigma_2)$.

Proof. This follows directly from theorems in [42]: both $(\to)$ and $(\to_\Gamma) = (\Gamma \circ (\to))$ are subsets of the non-deterministic transition relation $(\Rrightarrow)$. Hence, by Theorem 6.10 of [42], $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$ and by Lemma 6.8 of [42] and the definition of $(\to_\Gamma)$, $\Gamma(\varsigma_1) = \varsigma_1$, so that $\varsigma_1 = \Gamma(\varsigma_2)$. Both proofs of Theorem 6.10 and Lemma 6.8 in [42] are presented for a simple CPS-language, but can be repeated *mutatis mutandis* for λ_{ANF}. ◀

▶ **Lemma 5.** $(\widehat{\to}_{arc})$ *is sound.*

Proof. Assuming $\widehat{\varsigma}$ is sound and $\widehat{\varsigma} \mathrel{\widehat{\to}_{\mathsf{arc}}} \widehat{\varsigma}'$, by inversion we have that $\exists \widehat{\varsigma}_0$ so that $\widehat{\varsigma} \mathrel{\widehat{\to}_0} \widehat{\varsigma}_0$ and $\widehat{\varsigma}' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}_0)$. A trivial case analysis of $(\widehat{\to}_0)$ shows that $(\widehat{\to}_0)$, and therefore also $\widehat{\varsigma}_0$ are sound. We show that whenever $\widehat{\mathsf{collect}}$ adds an address $\widehat{l}$ to $\widehat{G}$, we have that $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}_0)$, so that by induction $\forall \widehat{l} \in \widehat{G} : \widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}_0)$. It is clear that whenever we add $\widehat{l}$ to $\widehat{G}$, every $\widehat{l}' \in \widehat{\phi}_{\hat{\varsigma}_0}(\widehat{l})$ must have already been deleted from $\widehat{\phi}_{\hat{\varsigma}_0}(\widehat{l})$, which implies that $\widehat{l}' \in \widehat{G}$. Hence, for every predecessor $\widehat{l}' \in \widehat{\phi}_{\hat{\varsigma}_0}(\widehat{l})$ we have that $\widehat{l}' \notin \widehat{\mathcal{R}}(\widehat{\varsigma}_0)$, and since $\widehat{l}$ can only deallocated when $\widehat{l} \notin \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}_0)$, it must be that $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}_0)$. By the soundness of $\widehat{\varsigma}_0$, we have that $\widehat{\mathcal{R}}(\widehat{\varsigma}_0) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma}_0)$. Since we have just shown that $\widehat{\mathsf{collect}}$ only removes addresses from $\widehat{\varsigma}_0$ that are not in $\widehat{\mathcal{R}}(\widehat{\varsigma}_0)$, we get $\widehat{\mathcal{R}}(\widehat{\varsigma}_0) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma}')$ and $\widehat{\mathcal{R}}(\widehat{\varsigma}') = \widehat{\mathcal{R}}(\widehat{\varsigma}_0)$, so that $\widehat{\mathcal{R}}(\widehat{\varsigma}') \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma}')$, hence $\widehat{\varsigma}'$ is sound. ◀

For convenience, we define $\widehat{\phi}^*_{\hat{\varsigma}}(\widehat{l}) = \{\widehat{l}' \mid \widehat{l}' \mathrel{\widehat{\rightsquigarrow}^*_{\hat{\varsigma}}} \widehat{l}\} \setminus \{\widehat{l}\}$, so that $\widehat{\phi}^*_{\hat{\varsigma}}(\widehat{l})$ is the set of all addresses that can directly or indirectly reach $\widehat{l}$ (excluding $\widehat{l}$ itself).

▶ **Lemma 7.** *In the absence of cycles,* $(\widehat{\to}_{arc})$ *is complete.*

Proof. Assume that $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\varsigma}'$, where $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_0} \widehat{\varsigma}_0$ and $\widehat{\varsigma}' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}_0)$. We need to prove that $\widehat{\mathcal{S}}(\widehat{\varsigma}') \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma}')$. Let $\widehat{l}$ be an address in $\widehat{\mathcal{S}}(\widehat{\varsigma}')$. If $\widehat{l} \notin \widehat{\mathcal{S}}(\widehat{\varsigma})$, then $\widehat{l}$ was added to the store or continuation store by one of the transition rules in $(\mathrel{\widehat{\rightarrow}_0})$: it is clear that in any of those cases, we also have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. If $\widehat{l} \in \widehat{\mathcal{S}}(\widehat{\varsigma})$, then by completeness of $\widehat{\varsigma}$, we have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$. Proceed by induction on $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l})|$. For the case $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l})| = 0$, by definition of $\widehat{\phi}^*$ and $\widehat{\phi}$ we also have that $|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| = 0$. If $|\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l})| = 0$, then $\widehat{l}$ was checked by $\widehat{\mathsf{collect}}$ because it must have been that $\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ (since $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$); if $|\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l})| > 0$, then $\widehat{l}$ was also checked because every address in $\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l})$ got removed, which caused $\widehat{l}$ to be added to $\widehat{C}$. Since in both cases $\widehat{l}$ was not removed after the check (where $\widehat{\phi}(\widehat{l}) = \emptyset$) it must have been that $\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$, and hence $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. For the case $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l})| = k > 0$, we have that $|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| > 0$. Pick any predecessor $\widehat{l}' \in \widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})$; in the absence of cycles, a predecessor always has fewer ancestors, i.e. $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l}')| < k$, so that by the induction hypothesis we get that $\widehat{l}' \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. Since $\widehat{l}' \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$ and $\widehat{l}' \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}'}} \widehat{l}$ (by definition of $\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})$), we have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. Hence, $\widehat{\mathcal{S}}(\widehat{\varsigma}') \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma}')$, i.e. $\widehat{\varsigma}'$ is complete. ◀

▶ **Theorem 9.** *In the absence of cycles, the transition relation, $(\mathrel{\widehat{\rightarrow}_{\mathsf{arc}}})$ is equivalent to $(\mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}})$ in the sense that: $\forall \widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\Sigma}$, $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}} \widehat{\varsigma}' \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\mathsf{rc}}(\widehat{\varsigma}')$.*

Proof. It is clear that the transition rules of $(\mathrel{\widehat{\rightarrow}_0})$ are isomorphic to those of $(\mathrel{\widehat{\rightarrow}})$, and that equivalence holds between both transition relations in that $\forall \widehat{\varsigma}, \widehat{\varsigma}_0 \in \widehat{\Sigma}$, $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}} \widehat{\varsigma}_0 \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_0} \widehat{\mathsf{rc}}(\widehat{\varsigma}_0)$. Both $(\mathrel{\widehat{\rightarrow}_{\mathsf{arc}}})$ and $(\mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}})$ then remove addresses from $\widehat{\sigma}$ and $\widehat{\sigma}_k$, and since both are garbage-free, they produce the same states (since $\widehat{\mathcal{S}}(\widehat{\varsigma}) = \widehat{\mathcal{R}}(\widehat{\varsigma})$ in both cases). ◀

Note that we can never have a reference from an address $\widehat{a} \in \widehat{\mathsf{Addr}}$ to an address in $\widehat{a}_k \in \widehat{\mathsf{KAddr}}$; This implies that is $\forall \widehat{a}_k \in \widehat{\mathsf{KAddr}}$, $\widehat{a}_k \in \widehat{\mathcal{R}}(\widehat{\varsigma}) \iff \widehat{a}_{k\widehat{\varsigma}} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{a}_k$. Moreover, it implies that a SCC consists either exclusively of addresses in $\widehat{\mathsf{Addr}}$ or addresses in $\widehat{\mathsf{KAddr}}$. We denote $\widehat{\pi}_k$ for the partitioning $\widehat{\pi}$ limited to $\widehat{\mathsf{KAddr}}$, i.e. $\widehat{\pi}_k = \widehat{\pi} \setminus \mathcal{P}(\widehat{\mathsf{Addr}})$.

▶ **Theorem 10.** *In a garbage-free abstract interpreter, extending abstract reference counting with cycle detection for the continuation store only requires amortized $\mathcal{O}(1)$ additional operations per continuation that is added to the continuation store.*

Proof. We use the *potential method* for amortized analysis [55]. Assume a state $\widehat{\varsigma}$ which is garbage-free, i.e. $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \widehat{\mathcal{S}}(\widehat{\varsigma})$. We define its potential Φ as the amount of references between SCCs in the continuation store $\widehat{\sigma}_k$ (, i.e. $\Phi(\widehat{\varsigma}) = \sum_{\widehat{scc}_k \in \widehat{\pi}_{k\widehat{\varsigma}}} |\widehat{\phi}_{\widehat{\varsigma}}(\widehat{scc}_k)|$. Cycle detection is only triggered for the continuation store on a transition $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{0++}} \widehat{\varsigma}'$ where we insert a new continuation $\widehat{\kappa}$ into the continuation store $\widehat{\sigma_k}_{\widehat{\varsigma}}$ (in the case of λ_{ANF}, this only happens in the transition rule E-Let). Since it is clear that $\Phi(\widehat{\varsigma}) \geq 0$, we can formulate the amortized cost $\widehat{c}$ of such an insertion as $c + (\Phi(\widehat{\varsigma}') - \Phi(\widehat{\varsigma}))$, where c is the actual cost of the insertion (including cycle detection, whose cost is proportional to the traversal of the backward search). If the address $\widehat{a_k}_{\widehat{\varsigma}'}$ is fresh (i.e. $\widehat{a_k}_{\widehat{\varsigma}'} \notin \widehat{\mathcal{S}}(\widehat{\varsigma})$), then the cycle detection requires no traversal, hence $\widehat{c} = 1 + (\Phi(\widehat{\varsigma}') - \Phi(\widehat{\varsigma}))$. We have that $\Phi(\widehat{\varsigma}') = \Phi(\widehat{\varsigma}) + 1$, since the insertion adds $\widehat{a_k}_{\widehat{\varsigma}'}$ to the set $\widehat{\phi}(\mathsf{find}(\widehat{\pi}, \widehat{a_k}_{\widehat{\varsigma}}))$, which results in $\widehat{c} = 2$. If the address $\widehat{a_k}_{\widehat{\varsigma}'}$ is reused (i.e. $\widehat{a_k}_{\widehat{\varsigma}'} \in \widehat{\mathcal{S}}(\widehat{\varsigma})$), then cycle detection will perform a backward search starting from $\widehat{a_k}_{\widehat{\varsigma}'}$. The key insight is that every address $\widehat{l}$ traversed by SEARCH will lead to $\widehat{a_k}_{\widehat{\varsigma}}$, hence we are guaranteed to have a cycle. For every such $\widehat{l}$, we have that $\widehat{l} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{a_k}_{\widehat{\varsigma}'}$ due to the backward search, and also that $\widehat{a_k}_{\widehat{\varsigma}} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{l}$ due to the garbage-free property of $\widehat{\varsigma}$ (since $\widehat{l} \in \widehat{\mathcal{S}}(\widehat{\varsigma})$ implies $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$, hence it must be that $\widehat{a_k}_{\widehat{\varsigma}} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{l}$). As a result, all references that are traversed will become part of the same SCC, and therefore be removed from $\widehat{\phi}$. Hence, if cycle detection traverses k references,

we have that $c = k + 1$ (due to the traversal of k references) and $\Phi(\widehat{\varsigma}') = \Phi(\widehat{\varsigma}) - k$ (due to the removal of k references), which results in $\widehat{c} = c + (\Phi(\widehat{\varsigma}') - \Phi(\widehat{\varsigma})) = 1$. In both cases, the insertion only requires amortized $\mathcal{O}(1)$ operations. ◀

B Supplementary Definitions for ($\widehat{\rightarrow}_{\mathsf{arc++}}$)

Figure 12 shows the updated auxiliary transition relation ($\widehat{\rightarrow}_{0++}$).

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{a}'_k = \widehat{\mathsf{alloc}}_k(e_1, \widehat{\varsigma}) \qquad \widehat{\rho}' = \widehat{\rho}[x \mapsto \widehat{a}] \qquad \widehat{\kappa} = \langle \widehat{a}, e_2, \widehat{\rho}', \widehat{a}_k \rangle \qquad \langle \widehat{\phi}', \widehat{\pi}' \rangle = \textsc{extend}(\widehat{a}'_k, \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa}), \widehat{\phi}, \widehat{\pi})}{\underbrace{\langle \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle}_{\widehat{\varsigma}} \widehat{\rightarrow}_{0++} \langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k \sqcup [\widehat{a}'_k \mapsto \widehat{\kappa}], \widehat{a}'_k, \widehat{\phi}', \widehat{\pi}' \rangle} \quad \text{(E-Let)}$$

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{\mathcal{A}}(f, \widehat{\rho}, \widehat{\sigma}) \ni \langle \lambda x.e', \widehat{\rho}' \rangle \qquad \widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \langle \widehat{\phi}', \widehat{\pi}' \rangle = \textsc{extend}(\widehat{a}, \widehat{\mathcal{T}}_{\widehat{\mathcal{P}(\mathsf{Clo})}}(\widehat{v}), \widehat{\phi}, \widehat{\pi})}{\underbrace{\langle f\, ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle}_{\widehat{\varsigma}} \widehat{\rightarrow}_{0++} \langle e', \widehat{\rho}'[x \mapsto \widehat{a}], \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}', \widehat{\pi}' \rangle} \quad \text{(E-Call)}$$

$$\frac{\widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\sigma}_k(\widehat{a}_k) \ni \langle \widehat{a}, e', \widehat{\rho}', \widehat{a}'_k \rangle \qquad \langle \widehat{\phi}', \widehat{\pi}' \rangle = \textsc{extend}(\widehat{a}, \widehat{\mathcal{T}}_{\widehat{\mathcal{P}(\mathsf{Clo})}}(\widehat{v}), \widehat{\phi}, \widehat{\pi})}{\langle ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle \widehat{\rightarrow}_{0++} \langle e', \widehat{\rho}', \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}'_k, \widehat{\phi}', \widehat{\pi}' \rangle} \quad \text{(E-Return)}$$

Figure 12 Auxiliary transition relation using abstract reference counting with cycle detection.

The updated definition for $\widehat{\mathsf{collect}}$ is given below. We slightly abuse notation for set removal here: for a partitioning $\widehat{\pi}$, $\widehat{\pi} \setminus S$ really means $(\widehat{\pi} \setminus S) \cup \bigcup_{s \in S} \bigcup_{e \in s} \{\{e\}\}$, while for a store $\widehat{\sigma}$, $\widehat{\sigma} \setminus S$ means $\widehat{\sigma} \setminus \bigcup_{s \in S} s$ (and analogous for $\widehat{\sigma}_k$).

$$\widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}') = \langle e_{\widehat{\varsigma}'}, \widehat{\rho}_{\widehat{\varsigma}'}, \widehat{\sigma}_{\widehat{\varsigma}'} \setminus \widehat{G}, \widehat{\sigma_k}_{\widehat{\varsigma}'} \setminus \widehat{G}, \widehat{a_k}_{\widehat{\varsigma}'}, \widehat{\phi}', \widehat{\pi}_{\widehat{\varsigma}'} \setminus \widehat{G} \rangle$$
$$\text{where } \langle \widehat{\phi}', \widehat{G} \rangle = \widehat{\mathsf{check}^*}(\widehat{C}_0, \emptyset, \widehat{\phi}_{\widehat{\varsigma}'}) \quad \widehat{C}_0 = \{\mathsf{find}(\widehat{\pi}_{\widehat{\varsigma}'}, \widehat{l}) \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})\}$$

$$\widehat{\mathsf{check}^*}(\widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \langle \widehat{\phi}, \widehat{G} \rangle & \text{if } \widehat{C} = \emptyset \\ \widehat{\mathsf{check}}(\widehat{scc}, \widehat{C} \setminus \{\widehat{scc}\}, \widehat{G}, \widehat{\phi}) & \text{otherwise, where } \widehat{scc} \in \widehat{C} \end{cases}$$

$$\widehat{\mathsf{check}}(\widehat{scc}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \widehat{\mathsf{dealloc}}(\widehat{scc}, \widehat{C}, \widehat{G}, \widehat{\phi}) & \text{if } |\widehat{\phi}(\widehat{scc})| = 0 \wedge \widehat{scc} \notin \widehat{R} \\ \widehat{\mathsf{check}^*}(\widehat{C}, \widehat{G}, \widehat{\phi}) & \text{otherwise} \end{cases}$$
$$\text{where } \widehat{R} = \{\mathsf{find}(\widehat{\pi}_{\widehat{\varsigma}'}, \widehat{l}) \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')\}$$

$$\widehat{\mathsf{dealloc}}(\widehat{scc}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \widehat{\mathsf{check}^*}(\widehat{C} \cup \widehat{S}, \widehat{G} \cup \{\widehat{scc}\}, \widehat{\phi} \ominus \bigsqcup_{\widehat{scc}' \in \widehat{S}} [\widehat{scc}' \mapsto \widehat{scc}])$$
$$\text{where } \widehat{S} = \{\mathsf{find}(\widehat{\pi}_{\widehat{\varsigma}'}, \widehat{l}') \mid \exists \widehat{l} \in \widehat{scc} \wedge \widehat{l} \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}'}} \widehat{l}' \wedge \widehat{l}' \notin \widehat{scc}\}$$

C Supplementary Code Listings

The following code is used to generate Figure 1, and is taken from [1]. It is also used for the `collatz` benchmark in Section 5.

```
(define (div2* n s)
  (if (= (* 2 n) s)
      n
      (if (= (+ (* 2 n) 1) s)
          n
          (div2* (- n 1) s))))
(define (div2 n)
  (div2* n n))
(define (hailstone* n count)
  (if (= n 1)
      count
      (if (even? n)
          (hailstone* (div2 n) (+ count 1))
          (hailstone* (+ (* 3 n) 1) (+ count 1)))))
(define (hailstone n)
  (hailstone* n 0))
(hailstone 5)
```