Correctness-Preserving Derivation of Concurrent Garbage Collection Algorithms

Martin T. Vechev
Cambridge University
mv270@cam.ac.uk

Eran Yahav
IBM Research
eyahav@us.ibm.com

David F. Bacon
IBM Research
dfb@us.ibm.com

Abstract

Constructing correct concurrent garbage collection algorithms is notoriously hard. Numerous such algorithms have been proposed, implemented, and deployed – and yet the relationship among them in terms of speed and precision is poorly understood, and the validation of one algorithm does not carry over to others.

As programs with low latency requirements written in garbage-collected languages become part of society's mission-critical infrastructure, it is imperative that we raise the level of confidence in the correctness of the underlying system, and that we understand the trade-offs inherent in our algorithmic choice.

In this paper we present correctness-preserving transformations that can be applied to an initial abstract concurrent garbage collection algorithm which is simpler, more precise, and easier to prove correct than algorithms used in practice — but also more expensive and with less concurrency. We then show how both pre-existing and new algorithms can be synthesized from the abstract algorithm by a series of our transformations. We relate the algorithms formally using a new definition of precision, and informally with respect to overhead and concurrency.

This provides many insights about the nature of concurrent collection, allows the direct synthesis of new and useful algorithms, reduces the burden of proof to a single simple algorithm, and lays the groundwork for the automated synthesis of correct concurrent collectors.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]; D.4.2 [*Storage Management*]: garbage collection

General Terms Verification, Algorithms

Keywords concurrent garbage collection, concurrent algorithms, verification, synthesis

1. Introduction

As garbage-collected languages like Java and C# become more and more widely used, the long pauses introduced by traditional synchronous ("stop the world") collection are unacceptable in many domains. This is true both at the high end, where the collection of multi-gigabyte heaps causes very long pauses, and at the low end, where systems are used for real-time, embedded, and sensor applications requiring very low latency. As a result, concurrent collectors are now available in most major production virtual machines.

However, concurrent collectors are extremely complex and error-prone. Since such collectors now form part of the trusted computing base of a large portion of the world's mission-critical software infrastructure, such unreliability is unacceptable.

The study of concurrent collectors began with Steele [41], Dijkstra [21], and Lamport [31].

Concurrent collectors were quickly recognized as paradigmatic examples of the difficulty of constructing correct concurrent algorithms: Steele's algorithm contained an error which he subsequently corrected [42], and Dijkstra's algorithm contained an error discovered and corrected by Stenning and Woodger [21]. Doligez and Leroy developed a multiprocessor collector for ML [23] which was subsequently found to contain an error [22]. Furthermore, some correct algorithms [9] had informal proofs that were found to contain errors [37].

Much later, Yuasa [46] introduced the snapshot-based algorithm, which is conceptually simpler and trades earlier termination and increased concurrency for reduced precision.

Many additional incremental and concurrent algorithms have been introduced over the last 30 years [30, 1, 2, 3, 4, 5, 6, 11, 13, 16, 17, 24, 26, 27, 29, 32, 34, 35, 40, 45], but there has been very little experimental comparison of the algorithms and no formal study of their relative merits. While there is now a well-established "bag of tricks" for concurrent collectors, each algorithm composes them differently based on the intuition and experience of the designer. However, because of the complex interactions of the invariants required by the different "tricks," many potential combinations of techniques are not used, leading to an underexplored design space. Furthermore, since each algorithm is different, a correctness proof for one algorithm cannot be re-used for others.

All concurrent collectors must decide how to answer the following basic questions:

- Where is the collector in its progress through the heap?
- Which objects must be traced to guarantee that all live objects will be found?
- How does the collector terminate in spite of allocation?
- Which interleavings are allowed between the mutator and the collector?

Our long-term research agenda is to be able to generate provably correct concurrent garbage collectors to meet the particular needs of the different target systems with respect to latency, throughput, and space consumption, from a simple base algorithm.

In previous work [44] we hypothesized that the way in which the above questions are answered could be expressed as transformations of a single base algorithm, informally described some such

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

transformations, and evaluated their relative performance experimentally.

In this work we substantiate that hypothesis: we present a single algorithm (which we call the *Apex* algorithm) and a set of composable transformations corresponding to each of the above questions. Each transformation can be applied to arbitrary subsets of the objects, or to restricted subsets for which we present precise formulations.

Transformations can be applied at the granularity of a single object in the heap. This allows enormous flexibility since different transformations can reliably be applied to different objects depending on their characteristics.

Furthermore, we formalize for the first time the notion of the *relative precision* of concurrent collectors, and express the transformations as correctness-preserving and precision-reducing. We also discuss informally how the reduction in precision provides useful tradeoffs in terms of implementation cost, speed of convergence, and level of concurrency.

Our transformational approach yields a wide range of algorithms. We show derivations for some well-known existing algorithms, and also derive some new algorithms which we expect will have desirable properties in practice. In particular, our generalization and formalization of the tradeoff between incremental-update and snapshot-at-the-beginning approaches allows a novel approach to newly allocated objects which yields high precision combined with rapid termination.

The contributions of this paper are:

- A formal framework for describing concurrent garbage collection algorithms;
- The simple "Apex" algorithm from which all others are derived;
- A set of transformations that can be applied to it to yield an enormous number of potential algorithms with different precision, concurrency, and efficiency properties;
- A formal definition of what constitutes relative precision of concurrent collectors;
- A proof of correctness for the transformations, which are shown to be precision-reducing (while improving other aspects of the algorithm);
- The application of the methodology to yield incremental update collectors in the styles of Dijkstra and Steele, snapshot collectors like that of Yuasa, as well as previously unknown algorithms with high precision, rapid termination, and high concurrency.

This work is presented in the context of a mark-and-sweep style of collector. While we show the synchronization with the sweep phase, we do not consider the details of its implementation, which contains a number of its own complexities. We have also simplified the design space by using a write barrier which is always atomic. This corresponds to some but not all implementations used in practice (for instance on a uni-processor safe point based virtual machine). So in general some manual transformation may still be required to achieve desired performance in the resulting algorithms.

A key aspect of this work is the *modularization* of the proof obligations. Although we have not yet proved the correctness of the Apex collector, we have proved the correctness of a broad variety of transformations needed for the creation of an efficient algorithm. This breaks the requirements for the creation of a correct algorithm and implementation into small modular proof components which can be re-used across an enormous range of algorithms, rather than requiring a monolithic proof of each new algorithm.

2. A Parametric Concurrent Collector

In this section we present a parametric concurrent collection algorithm that is used to instantiate all of the algorithms in our framework. The parametric algorithm is parameterized by a function called *expose* that determines how the collector handles objects that may be "hidden" due to concurrent mutations.

The *expose* function determines which objects should be used as starting points for additional tracing, and is based on a log of mutator and collector operations. This log is the formal analogue to the information captured by the write barrier (see e.g., [30, Sec 8.2]) in real-world implementations.

2.1 A Trace Model for Concurrent Collectors

We model the heap of a program that uses a (fixed) set of field identifiers $Fields = \{f_1, \ldots, f_F\}$ as an (unbounded) set of objects $AL \subseteq \mathcal{U}$ (where $\mathcal{U}$ is the infinite universe of all objects) and a function $h : AL \rightharpoonup Fields \to (AL \cup \{\mathbf{null}\})$ mapping fields of allocated objects to their values, which may be other allocated objects or the designated value **null**.

For convenience, we use $obj.f_i$ to denote the value $h(obj)(f_i)$.

All reachable objects are reachable from a finite set of $R \subseteq AL$ root objects, denoted $root_1, \ldots, root_R$.

For the time being we assume that stack frames are heap-allocated objects – some systems, especially for functional languages are in fact implemented in this way.

A global state of the program consists of: (i) the heap; (ii) the state of the mutator; (iii) the state of the collector. We model the mutator as a sequence of allocations and mutations of pointers over a given heap. The state of the mutator is its position in the sequence of allocations and mutations. The state of the collector consists of an assignment of values to all its variables.

A program trace is a potentially infinite sequence of program global states, where a *sequence* is defined in a standard way as a map from natural numbers to an alphabet Σ. We denote the empty sequence by ϵ. Given a sequence $S = S_0, S_1, \ldots$, we define its finite prefix P of length $|P| = k$, to be the first k letters in the sequence $S_0, S_1, \ldots, S_{k-1}$, and denote it by $\mathrm{pre}(S, k)$.

Given a finite sequence prefix P and a sequence S, we denote the concatenation of P and S by $P \bullet S$. Similarly, given a finite sequence prefix P and a letter $\tau \in \Sigma$, we denote by $P \bullet \tau$ the concatenation of P and τ.

Our algorithm uses an *interaction log* to record information about the combined behavior of the collector and the mutator. This log is used by the collection algorithm to select the objects to be marked. A log of the interleaving of mutator/collector operations is natural for a concurrent collector because it closely matches the use of write barriers (see Fig. 2) in practical implementations: the function of the write barrier is to synchronize the mutator with the collector, which in some collectors is done using a log of writes.

The interaction log is a sequence of log entries of the following kinds: (i) a *tracing* entry recording a tracing action of the collector as it traverses the heap during the marking phase; (ii) a *mutation* entry recording a pointer redirection action by the mutator; (iii) an *allocation* entry recording an allocation of a new object by the mutator. This is formally defined as follows:

DEFINITION 2.1. *A* log entry *is a tuple* $\langle \mathbf{k}, source, fld, old, new\rangle \in \{\mathbf{T}, \mathbf{M}, \mathbf{A}\} \times AL \times Fields \times (AL \cup \{\mathbf{null}\}) \times (AL \cup \{\mathbf{null}\})$ *where:*

- *k identifies the kind of action as one of tracing, mutation, or allocation, denoted by* **T, M,** *and* **A**, *respectively.*
- *$source$ is the object affected by the action.*
- *fld is the field of $source$ affected by the action.*
- *old is the value of the field $source.fld$ prior to the action.*
- *new is the value of $source.fld$ subsequent to the action.*

```
collect() {
  atomic
    marked ← {root_1, ..., root_R}
    pending ← ⋃_{x∈marked} fields(x)
    log ← ϵ

  do {
    mark()
    addOrigins()
  } while (?)

  atomic
    addOrigins()
    mark()

  sweep()
}

mark() {
  while (pending ≠ ∅) {
    (obj, fld) ← removeElement(pending)
    atomic
      dst ← obj.fld
      log ← log • ⟨T, obj, fld, dst, dst⟩
    if (dst ≠ null ∧ dst ∉ marked){
       marked ← marked ∪ {dst}
       pending ← pending ∪ fields(dst)
     }
  }
}

addOrigins() {
  atomic
    origins ← expose(log) \ marked

  marked ← marked ∪ origins
  pending ← pending ∪ (⋃_{x∈origins} fields(x))
}
```

Figure 1. Parametric Mark-and-Sweep Collector.

```
mutate(source, fld, new) {
  atomic
    log ← log • ⟨M, source, fld, source.fld, new⟩
    source.fld ← new
}

mutateAlloc(source, fld) {
  atomic
    new ← allocate new object
    log ← log • ⟨A, source, fld, source.fld, new⟩
    source.fld ← new
}
```

Figure 2. Mutator write-barrier and allocation-barrier.

Tracing actions do not change the structure of the heap; therefore $old = new$ for all tracing entries. Allocation actions allocate the object new, which must not appear previously in the trace.

For convenience, we define selectors for log entry tuples. Given a tuple $\tau = \langle \mathbf{K}, s, f, o, n \rangle$, we define $\tau.kind = \mathbf{K}$, $\tau.source = s$, $\tau.field = f$, $\tau.old = o$, and $\tau.new = n$.

2.2 The Parametric Algorithm

Fig. 1 presents the pseudo-code for a parametric concurrent mark-and-sweep collector. The operation of this collector is defined over a prefix of the interaction log, recording the collector and mutator interaction. Recording mutator actions in the log is performed by the mutator's write and allocation barriers, as shown in Fig. 2.

Before describing the parametric algorithm in more detail, we first describe the assumptions we have made for clarity of presentation and the assumptions under which the algorithm operates:

- we do not specify how the `sweep()` operation proceeds, except to ensure that there is the proper synchronization between the mark and sweep phases. We also do not consider compaction, which requires the dynamic relocation of objects. While these are both important issues, they are beyond the scope of this work.
- we assume that there is only a single execution of a collection cycle at any given point in time. That is, the **T** operations in the log all belong to a single collection cycle. Multiple (even overlapped) collections can be performed by differentiating **T** entries accordingly.

The parametric collection algorithm does not specify how objects are selected to be marked. This is defined to be a parameter of the collector. The algorithm, however, does restrict concurrency by assuming that write barriers are atomic with respect to collector operations. This assumption is inline with practical systems as mutators and the collector are only allowed to interleave at *safe points*, which do not include the write-barriers. Effectively, this means that a collector cannot preempt a mutator during a write barrier. Under this assumption, we can restrict attention to a system with a single mutator thread without loss of generality.

The collection cycle of the algorithm is described in the `collect()` procedure. The collection cycle consists of two phases: (i) the *marking phase*, in which the collector marks potentially live objects; (ii) the *sweeping phase*, in which unmarked objects are reclaimed.

The collection cycle starts by atomically selecting the set of root objects as origins. This operation is executed atomically, and thus no concurrent mutations could be performed by the mutator.

After selecting the root objects as origins, the collector proceeds by repeatedly tracing heap objects and marking them (`mark()` procedure), and adding origins to be considered by the collector due to concurrent mutations performed by the mutator (`addOrigins()` procedure). These two steps are repeated until a non-deterministic choice (denoted by '?' in the figure) triggers a move to an atomic phase in which the remaining origins and objects to be marked are processed atomically. This atomic phase guarantees the termination of the algorithm, and is in line with some practical collector implementations (e.g., [7]). Nevertheless, in Section 5.5, we show how to derive algorithms in which this atomic marking phase can be eliminated.

After the marking phase has completed, the sweep phase reclaims all objects that are not marked.

2.3 Marking Traversal

The `mark()` procedure implements a collector traversal of the heap. In the algorithm, we use a pair (obj, fld) to denote the field fld of an object obj. We use $obj.fld$ to denote the object pointed to by the field fld of the object obj, and $fields(obj) = \{(obj, f_1), \ldots, (obj, f_F)\}$ to denote the set of all object fields for a given object obj.

The procedure uses a set $pending$ of pending fields to be traversed, and performs a transitive traversal of the heap by iteratively removing an object field from the set $pending$ and tracing from it. Whenever an object field is traced-from, the procedure inserts a tracing entry into the log. When the traced object field points to an unmarked object, the object is marked, and its fields are added to the pending set. Note that the collector is able to trace object fields in an arbitrary order, rather than scanning fields of each object in order.

During this traversal, the mutator might concurrently modify the heap. These concurrent mutations might cause reachable objects to be *hidden* from the traversal, and thus may remain unmarked by the traversal.

2.3.1 The Collector Wavefront

All collectors discussed in this paper rely on cooperation between the collector and the mutator to guarantee correctness in the presence of concurrency. A key part of the cooperation is tracking the progress of the collector through the heap, since mutations can be treated differently depending on whether they happened in the portion of the heap already scanned by the collector (behind the wavefront) or not yet scanned (ahead of the wavefront). The wavefront consists of the set of object fields (that is, *not* the values of the pointers in those fields) that have been traced by the collector thus far.

DEFINITION 2.2. *Given a log prefix P, the* collector wavefront *is the set of object-fields that have been traced by collector operations in P, that is:*

$$\mathcal{W}(P) = \{(P_i.source, P_i.field) \mid P_i.kind = \mathbf{T} \wedge 0 \leq i < |P|\}$$

Given a log prefix P, we say that an object field (o, f) is behind the wavefront *when $(o, f) \in \mathcal{W}(P)$, and* ahead of the wavefront *when $(o, f) \notin \mathcal{W}(P)$.*

Most practical collectors use conservative abstractions of the wavefront rather than the precise definition provided here. That is, the wavefront is tracked at an object granularity. However, the precise wavefront is not merely theoretical and has recently been used in the hardware-assisted collector for the Azul Java server, which has a "not marked-through" bit in every pointer [18].

EXAMPLE 2.3. Fig. 3 shows an example of a possible mutator and collector interleaving. In this figure, the progress of collector tracing through the heap is shown by the tracing actions $T_1, \ldots, T_6$, and by using a darker color for traced fields. The sequence of mutations is shown as $M_1, \ldots, M_7$. For brevity, we only present part of the states and show the effect of multiple operations in a single step.

The interaction log prefix P^e for this example is:

$$\langle \mathbf{T}, r1, f2, A, A\rangle, \langle \mathbf{T}, A, f1, null, null\rangle, \langle \mathbf{T}, r1, f3, null, null\rangle, \langle \mathbf{M}, r1, f1, null, B\rangle, \langle \mathbf{M}, A, f1, null, B\rangle, \langle \mathbf{M}, r1, f3, null, E\rangle, \langle \mathbf{M}, A, f2, C, null\rangle, \langle \mathbf{M}, r1, f1, B, null\rangle, \langle \mathbf{T}, A, f2, null, null\rangle, \langle \mathbf{T}, r1, f1, null, null\rangle, \langle \mathbf{M}, A, f3, D, null\rangle, \langle \mathbf{M}, A, f1, B, null\rangle, \langle \mathbf{T}, A, f3, null, null\rangle$$

The wavefront at the end of the shown prefix P^e is: $\mathcal{W}(P^e) = \{(r1, f2), (A, f1), (r1, f3), (A, f2), (r1, f1), (A, f3)\}$

2.4 Adding Origins

The `addOrigins()` procedure uses the interaction log to select a set of additional objects to be considered as origins. When this procedure is invoked by the collector, it is possible that a number of reachable pointers were hidden by the mutator behind the wavefront during the `mark()` procedure. The `addOrigins()` procedure finds a safe over-approximation of these hidden, but reachable objects.

The core of `addOrigins()` is the function $expose$ which takes a log prefix and returns a set of objects that should be considered as additional origins. Each object returned by $expose$ is then marked, and its fields are inserted into the $pending$ set.

In the next sections, we present various choices of $expose$ corresponding to different garbage-collection algorithms.

2.4.1 Mutator Barriers

Fig. 2 shows the write-barrier and allocation-barrier used by the mutator. The procedure `mutate(source,fld,new)` is called by the mutator to mutate a pointer in the heap. The procedure `mutateAlloc(source,fld)` is called by the mutator to allocate a new object and store it in the given object field. To collaborate with the collector, the mutator barriers append their actions to the interaction log.

When the mutator performs an assignment $source.fld \leftarrow new$ with $new \neq$ **null**, we say that a pointer is *installed* from $(source, fld)$ to new. When the object field $(source, fld)$ is behind the wavefront, we say that the pointer is *installed behind the wavefront*. Otherwise, we say that the pointer is installed ahead of the wavefront.

Similarly, whenever we assign a value to a field $(source, fld)$ containing an existing pointer, we say that the existing pointer is *deleted*. If the field $(source, fld)$ is ahead (behind) of the wavefront, we say that the pointer is *deleted ahead (behind) of the wavefront*.

EXAMPLE 2.4. In Fig. 3 the mutation (M_2) results in a pointer from $(A, f1)$ to B *installed behind the wavefront*, and the mutation (M_6) results in the pointer from $(A, f3)$ to D being *deleted ahead of the wavefront*.

3. The Apex Algorithm

The **Apex** algorithm is an instance of the parametric collector presented in Fig. 1 and is the starting point for the derivation steps described in the rest of the paper.

The Apex algorithm uses a technique called *rescanning*. Rescanning is a technique which given a set of objects, identifies the object fields that were modified behind the collector wavefront. It then returns the pointers to the objects residing in those modified fields, which are subsequently marked and traced from. This approach is necessary to $expose$ reachable objects that are hidden by a sequence in which: (i) a pointer to an object is stored in a field behind the wavefront; and (ii) all other paths to the object ahead of the wavefront are removed before the collector reaches them. Rescanning provides a high degree of accuracy, since all hidden pointers are identified precisely.

The specific $expose$ used by the Apex algorithm is the following:

$$\begin{aligned} expose^{apex}(P) = \{o.f \mid\ & P_i.source = o \\ & \wedge P_i.field = f \wedge 0 \leq i < |P| \\ & \wedge P_i.kind \in \{\mathbf{M}, \mathbf{A}\} \wedge (o, f) \in \mathcal{W}(\mathrm{pre}(P, i))\} \end{aligned}$$

Given a log prefix P, $expose^{apex}(P)$ returns the current contents of all of the mutated fields behind wavefront.

Note that because the execution of *expose* is performed inside of an **atomic** block in the algorithm in Figure 1, the rescanning of *all* of the fields of *all* of the objects that were modified are scanned atomically. This means that a pure rescanning algorithm has very low concurrency.

EXAMPLE 3.1. Consider the example, interleaving of Fig. 3. The function $expose^{apex}(P^e)$ atomically performs rescanning of the fields $(A, f1)$ and $(r1, f3)$. This results with $expose^{apex}(P^e) = \{E\}$. Unless returned by $expose^{apex}$, object E would have been lost. Assuming there are no further mutations, the objects marked by the Apex algorithm in this collection cycle will be: $\{r1, A, E\}$. In future sections, we see that other (less precise) collectors will consider additional objects as live.

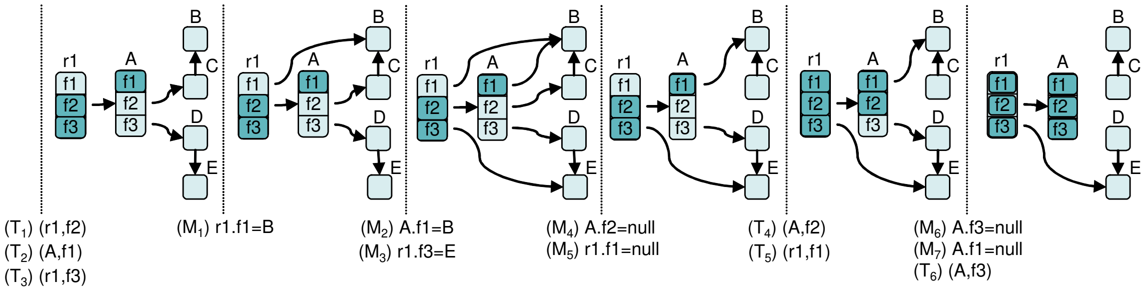

Figure 3. Example interleaving of mutations and tracing operations.

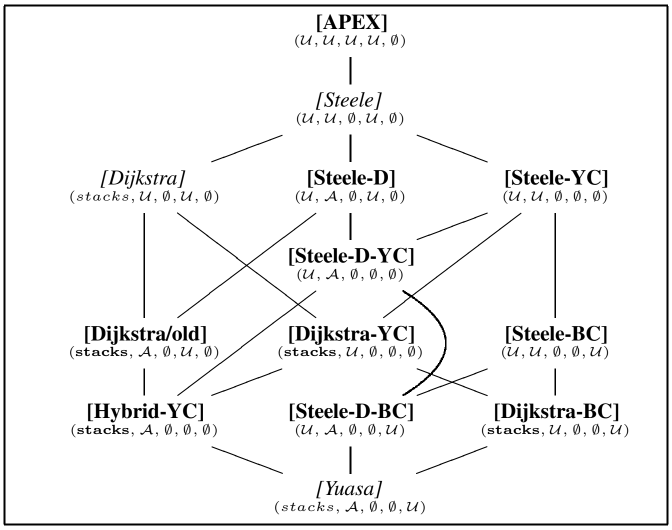

Figure 4. Relative precision of part of the existing and newly derived algorithms. New algorithms are shown in boldface. The most precise algorithm is shown at the top. For readability, only derived algorithms with abstracted wavefront are shown. Tuples are of the form (SR, IS, FL, WC, BC).

Conceptually, Apex is very similar to the Steele algorithm [41] algorithm, but with an accurate $\mathcal{W}$ definition.

In our framework, the Apex algorithm is used as the base algorithm from which all other algorithms are derived using correctness-preserving transformations. Fig. 4 shows part of the derived algorithms, ordered by relative precision.

In this paper, our focus is on unifying the various collection algorithms, and relating them in terms of precision. We show that our transformations are correctness-preserving, but assume that the Apex algorithm is a correct starting point.

4. Precision of Collection Algorithms

In this section, we introduce the notion of relative precision of concurrent collection algorithms. This allows us to formally relate the various algorithms which are instantiated in our parametric framework.

A correct marking algorithm must satisfy both a *safety* property that it marks at least all live objects, and a *liveness* property that it terminates. These requirements can be satisfied by a variety of correct collectors with varying degrees of precision, efficiency, and atomicity.

The algorithmic differences between the various collectors are manifested in the additional unreachable objects that they mark (and thus retain). It is therefore natural to define relative precision between two collectors by comparing their $marked$ sets at the end of the marking phase.

There is a trade-off between the precision of a collector and the degree of concurrency it provides. For example, a stop-the-world collector trades all concurrency for obtaining maximal precision (all unreachable objects are collected). Other algorithms provide a higher degree of concurrency (finer grained atomicity), at the expense of retaining more unreachable objects at the end of the marking phase.

In this paper we focus on the relative precision of algorithms under a given predetermined kind of atomicity. That is, the atomicity constraints of the generic algorithm are fixed and all instantiated algorithms in our parametric framework follow the same atomicity restrictions, namely the **atomic** blocks used in Figures 1 and 2. The precision-reducing transformations presented in the next section create opportunities for concurrency-increasing transformations. Although we do not deal with concurrency transformations which alter atomic sections, the instantiated algorithms do have a shorter duration of these atomic sections. As future work we plan on extending the parametric framework in this paper to include atomicity transformations.

An *algorithm* in our framework consists of the skeletons of Figures 1 and 2, instantiated with an arbitrary $expose$ function. Note that in general an arbitrary choice for the $expose$ function might yield an incorrect algorithm, but as we show later, all of the $expose$ functions used in our framework result with correct algorithms.

The correctness of algorithms in our framework hinges on $expose$ exposing all hidden origins (e.g., object "E" in Example 3.1). Intuitively, given a correct algorithm (exposing all hidden origins), any algorithm that exposes a superset of these origins is also a correct algorithm. In the next section we present *correctness-preserving* transformations that when given a correct algorithm maintain the property that (at least) all hidden objects are exposed.

We now consider the question of the relative precision of two algorithms. Intuitively, a more precise algorithm should always mark fewer objects. However, this is *not* the case because the actual set of objects marked depends on the specific interleaving of mutator and collector. In fact, there exist algorithms and interleavings such that an algorithm that *always* selects more objects from the mutation log as origins for transitive marking will in fact mark fewer objects during the collection as a whole.

This apparent anomaly arises because what such a notion of precision compares is not necessarily the effect of the algorithm but may be the effect of arbitrary interleavings. Thus, a meaningful comparison must factor out the non-deterministic effects of particular executions.

We therefore consider algorithm C_2 to be less precise when for *any given global state* it exposes more objects for marking than algorithm C_1. We therefore consider the effect of $expose_{C_1}$ on any interaction log obtained by C_2, and show that when transitive marking is complete (when $pending = \emptyset$), $expose_{C_1}$ returns a subset of the objects returned by $expose_{C_2}$.

DEFINITION 4.1. *Given two collection algorithms C_1 and C_2, we say that C_1 is* more precise *than C_2, denoted $C_1 \sqsubseteq C_2$, when given any global state of C_2 with an interaction log l and where the set* pending *is empty, $expose_{C_1}(l) \subseteq expose_{C_2}(l)$.*

In the following section we present precision-reducing and correctness-preserving transformations of algorithms in our framework, and show that if the initial algorithm C_1 is correct, then the resulting algorithm C_2 is also correct.

5. Correctness-preserving Transformations

In this section, we present various transformations that can be combined to systematically derive safe collection algorithms from the Apex collector. For each transformation, we show that its application is *correctness-preserving* and *precision-reducing*.

Each transformation is applied across a *dimension*. Dimensions are the formal analogue of the basic variables in the design of a collector as presented informally in the introduction. Specifically, we parameterize the collector with the following dimensions:

- **Wavefront:** how far has the collector progressed?
- **Policy:** how are modified objects behind the wavefront treated?
- **Threshold:** how large are cross-wavefront counts allowed to grow before they are "stuck"?
- **Protection:** which objects are traced to guarantee that all live objects are found?
- **Allocation:** how does the collector handle newly allocated objects to ensure timely termination?

A dimension is described by an ordered partition $\langle P_1, \ldots, P_n \rangle$ of the objects in $\mathcal{U}$, where each subset of the partition corresponds to a different manner of handling objects in the dimension.

The subsets of a partition have the property that moving an object to a subset "to the right" yields an algorithm of lower precision. A *transformation* in our framework is defined as moving an object to the right within a partition.

Formally, the relation between subsets is such that if $i < j$, $x \in P_i$, algorithm C uses partition $\langle \ldots, P_i, \ldots, P_j, \ldots \rangle$ and algorithm C' uses partition $\langle \ldots, P_i \setminus \{x\}, \ldots, P_j \cup \{x\}, \ldots \rangle$, then $C \sqsubseteq C'$.

For each dimension, we have generalized the algorithm so that all of the mechanisms represented by the subsets can be used simultaneously within a collector. In some dimensions, there is no restriction on the partitioning of $\mathcal{U}$; for others, we specify a restriction formally.

All theorems stated in the paper have been proved, but due to space restrictions these proofs are provided in an online supplement [43].

5.1 The Wavefront Dimension

The wavefront denotes the progress of the collector through the heap. We defined the precise wavefront in Section 2.3.1: it consists of the set of object fields that have thus far been traced by the collector. The *expose* function determines how to handle mutations to the heap depending on whether they occur behind or in front of the wavefront.

The wavefront dimension represents different choices for the granularity at which the collector progress is tracked. Tracking the wavefront precisely may be inefficient because it requires per-field information, so it is sometimes desirable to sacrifice some precision in exchange for a more efficient implementation.

The wavefront dimension is an ordered partition of the objects in $\mathcal{U}$ into:

$$D_W = \langle FL, OL \rangle$$

where objects in the FL subset have their wavefront tracked precisely (at "Field Level") as in Definition 2.2, while objects in the OL subset do not distinguish between fields within an object (and are tracked at "Object Level").

One could further generalize this dimension to include all possible subsets of fields for all objects, but we do not consider this here for simplicity of presentation.

There are no restrictions on how the objects may be partitioned in the wavefront dimension.

5.1.1 Wavefront Abstraction Transformation

This abstracts the exact collector wavefront, and tracks the collector's progress at the granularity of an object rather than at the granularity of individual fields of objects. Given the wavefront partition, we define the abstracted wavefront as follows:

$$\begin{array}{rcl} \mathcal{W}^{>}(P) & = & \{(o,f) \mid (o,f) \in \mathcal{W}(P) \wedge o \in FL\} \cup \\ & & \{(o,f) \mid \exists_{f' \in Fields} \colon (o,f') \in \mathcal{W}(P) \wedge \\ & & \quad f \in Fields \wedge o \in OL\} \\ \mathcal{W}^{<}(P) & = & \{(o,f) \mid (o,f) \in \mathcal{W}(P) \wedge o \in FL\} \cup \\ & & \{(o,f) \mid \forall_{f' \in Fields} \colon (o,f') \in \mathcal{W}(P) \wedge \\ & & \quad f \in Fields \wedge o \in OL\} \end{array}$$

The abstracted wavefront consists of two functions, $\mathcal{W}^{>}(P)$ and $\mathcal{W}^{<}(P)$. The function $\mathcal{W}^{>}(P)$ over-approximates the set of object fields behind the wavefront. The function $\mathcal{W}^{<}(P)$ under-approximates the set of objects fields behind the wavefront (and thus over-approximates the set of object fields ahead of the wavefront). Both functions are needed because *expose* functions that depend on an object's field being behind or ahead of the wavefront must each use a conservative approximation.

EXAMPLE 5.1. Consider a prefix of the example interleaving of Fig. 3 just before tracing action ($T6$) is performed and the field $(A, f3)$ is traced. For this prefix P assuming $FL = \emptyset$ we get the following:

$$\begin{array}{l} \mathcal{W}(P) = \{(r1,f2),(r1,f3),(A,f1),(A,f2),(r1,f1)\} \\ \mathcal{W}^{>}(P) = \{(r1,f2),(r1,f3),(A,f1),(A,f2),(r1,f1),(A,f3)\} \\ \mathcal{W}^{<}(P) = \{(r1,f2),(r1,f3),(r1,f1)\} \end{array}$$

The wavefront abstraction transformation moves an object o from FL to OL.

THEOREM 5.2. *The wavefront abstraction transformation is a correctness-preserving and precision-reducing transformation.*

5.2 The Policy Dimension

Traditionally, when deciding how to protect against lost objects, implementers have thought in terms of the three classical types of write barriers: those of Dijkstra [21], which records the new pointer stored into an object; of Steele [41], which records the pointer to the object being modified; and of Yuasa [46], which records the old pointer that was overwritten.

However, while this decomposition may seem intuitive it does not in fact capture the essential properties of the design space in an orthogonal manner. Therefore, we introduce two separate and orthogonal dimensions which determine how objects are protected and do so in a manner that allows the different mechanisms to be composed.

In the Apex collector of Section 3, we used *rescanning* as a uniform policy for protecting all objects. The key to the simplicity of rescanning is that it finds all pointers to traverse in an atomic step of the $expose$ operator.

However, this atomicity is costly and therefore rescanning is generally applied to some minimal portions of the memory, such as the stacks, for which it is practical to do so. The rest of the objects are processed incrementally from the log.

The policy dimension determines whether the modifications to a field are found by atomically scanning the heap (called "Scan-based Reachability") or by examining the log (called "Log-based Reachability").

This dimension is an ordered partition of objects in $\mathcal{U}$ into

$$D_P = \langle SR, LR \rangle$$

The objects in SR are rescanned as described previously; the objects in the LR are discovered solely by processing the log, without accessing the contents of the heap.

There are no restrictions on how the objects may be partitioned in the policy dimension.

5.2.1 Rescanning

In order to define an $expose$ function that works along this dimension, we first have to refine the simplified definition of a rescanning collector we presented in Section 3 and parameterize it according to the potentially imprecise wavefront and the partitioned policy dimension:

$$\begin{aligned} expose^r(P) = \{o.f \mid\ & P_i.kind \in \{\mathbf{M}, \mathbf{A}\} \\ & \wedge P_i.source = o \wedge P_i.field = f \\ & \wedge (o, f) \in \mathcal{W}^{>}(\text{pre}(P, i)) \wedge o \in SR \wedge P_i.new \in IS \\ & \wedge 0 \leq i < |P| \wedge (o.f) \in IS\} \end{aligned}$$

For the time being the set $IS = \mathcal{U}$; non-trivial use will be made of the set IS below for the protection dimension.

5.2.2 Maintaining Cross-Wavefront Counts

When a field of an object is modified repeatedly, rescanning will only see the final value when it processes the log. That is, there may have been intermediate values stored by the mutator and subsequently overwritten. We now describe a different way of discovering the resulting pointers, which is based on counting the number of references to an object from behind the wavefront.

In particular, we observe that if a field initially contains a pointer p_0 and then has a sequence of pointers $p_1, \ldots, p_n$ written to it, then rescanning will find only p_n. If we were to apply a specialized form of reference counting, then the reference counts of pointers $p_1, \ldots, p_{n-1}$ would remain unchanged: they would be first incremented and then decremented. In the end, only the reference counts of p_0 and p_n would change, being decremented and incremented, respectively. This means that the intermediate reference counting operations can be ignored.

This observation is originally due to Barth [8], and is central to the multiprocessor reference counting algorithm of Levanoni and Petrank [33]. In our formulation, we use this approach to show how rescanning can be replaced by log-based reachability which keeps a count of references from behind the wavefront in order to identify exactly the same objects as are found by rescanning.

Note that this is *not a general form of reference counting*. Our framework only covers tracing algorithms. In particular, since counts are only maintained from behind to in front of the wavefront, there can be no cycles of objects with non-zero counts.

Existing approaches will subsequently be shown to be degenerate cases of reference counting in which the cross-wavefront count is a single sticky bit (expressed by the threshold dimension).

The counting-based approach has the potential advantage of not requiring the synchronization of the $expose$ function; on the other hand it may perform n steps for the mutated field described above, whereas rescanning would perform exactly one. One area in which this tradeoff manifests itself is in the treatment of stacks, whose high mutation rate makes them unsuitable for write barriers – instead, they are rescanned atomically.

Mutator Count: The mutator count is the number of pointers to an object from object fields behind the wavefront. This quantity is computed with respect to a given wavefront. We assume that some objects in the heap are *rescanned objects* that do not affect the mutator count. The mutator count computation is therefore parameterized by a set of objects LR *from* which the count should be computed. To compute the mutator count from a given log prefix P, we define the mutator-count increment and decrement as follows:

$$\begin{aligned} M^{+}(o, P) = |\{P_i \mid\ & P_i.kind \in \{\mathbf{M}, \mathbf{A}\} \wedge P_i.new = o \\ & \wedge (P_i.source, P_i.field) \in \mathcal{W}^{>}(\text{pre}(P, i)) \\ & \wedge P_i.source \in LR \wedge 0 \leq i < |P|\}| \end{aligned}$$

$$\begin{aligned} M^{-}(o, P) = |\{P_i \mid\ & P_i.kind \in \{\mathbf{M}, \mathbf{A}\} \wedge P_i.old = o \\ & \wedge (P_i.source, P_i.field) \in \mathcal{W}^{<}(\text{pre}(P, i)) \\ & \wedge P_i.source \in LR \wedge 0 \leq i < |P|\}| \end{aligned}$$

The value $M^{+}(o, P)$ is the number of new references introduced by the mutator from object fields that are behind the wavefront. Similarly, the value $M^{-}(o, P)$ is the number of references removed by the mutator from object fields behind of the wavefront. The mutator count $M(o, P)$ is computed by combining the mutator-count increments and decrements as follows:

$$M(o, P) = M^{+}(o, P) - M^{-}(o, P)$$

EXAMPLE 5.3. Consider the example of Fig. 3 and its corresponding interaction log P^e as shown in Example 2.3. Assuming that $LR = \{A\}$ and $FL = \mathcal{U}$, the mutator count for B increases to 1 when the pointer from $(A, f1)$ is installed, and is decreased back to 0 as the pointer is deleted. Therefore, at the end of the prefix P^e, $M(B, P^e) = 0$. Note that the installation from $(r1, f1)$ does not increment the count, as the installation takes place ahead of the wavefront.

Collection by Counting: Using the formulation of Section 2.2, a counting-based collector can be instantiated using the following $expose$ function. We use the superscript c to denote the fact that this function is based on counting, and name the function $expose^c$.

$$\begin{aligned} expose^c(P) = \{n \mid\ & n = P_i.new \wedge M(n, P) > 0 \\ & \wedge n \in IS \wedge 0 \leq i < |P|\} \end{aligned}$$

EXAMPLE 5.4. Consider the example of Fig. 3. Assuming $LR = \mathcal{U}$, $FL = \mathcal{U}$, the function $expose^c(P^e) = \{E\}$.

However, since counting depends on the wavefront, taking a less precise wavefront can result with more objects being exposed. For example, taking $FL = \mathcal{U} \setminus \{r1\}$ results with $expose^c(P^e) = \{E, B\}$ as the count for B is incremented on the installation from $(r1, f1)$ that is behind the (overapproximated) wavefront, but cannot be decremented when the mutation $r1.f1 = null$ takes place, as $(r1, f1)$ is not behind the (underapproximated) wavefront.

Note that using a less precise wavefront resulted with additional objects exposed by the algorithm. In particular, in this case $expose^c(P^e) = \{E, B\}$ is a superset of the origins exposed by Apex algorithm on the same prefix as (see in Example 3.1).

We now formally define an $expose$ function that works along the D_P dimension:

$$expose^{rc}(P) = expose^r(P) \cup expose^c(P)$$

The following theorem shows that moving along the D_P dimension is a precision reducing transformation.

THEOREM 5.5. *The rescanning to counting transformation, moving an object from SR to LR, is a correctness-preserving and precision-reducing transformation.*

It is interesting to note that in the special case in which a precise wavefront is maintained for all objects, and under an infinite mutator count, the precision of any partition along the D_P dimension is identical.

5.3 The Threshold Dimension

The threshold dimension represents different choices for the precision of maintaining the mutator count introduced in the previous section. In real systems, these counts are usually very small. Therefore, it would be wasteful to have a very large reference count per object.

The threshold limits the mutator count to a maximum value, at which it "sticks" and is not subsequently decremented. This allows counts to be implemented with a fixed (small) number of bits while still maintaining the correctness properties provided by reference counting.

The threshold dimension is an ordered partition of the objects in $\mathcal{U}$ into:

$$D_T = \langle \mathcal{C}_\infty, \ldots, \mathcal{C}_k, \ldots, \mathcal{C}_1 \rangle$$

where the subsets represent the count with successively less and less precision, which leads to collectors which are successively less precise, as we will show below. There are no restrictions on how the objects may be partitioned in the threshold dimension.

5.3.1 Abstracting Mutator Count

The mutator count $M(o, P)$ can be abstracted to range over an interval $[0, k)$ and ∞, defined as follows:

$$M_k(o, P) = \begin{cases} \infty, & \exists i.M(o, \mathrm{pre}(P, i)) \geq k; \\ M(o, P), & \text{otherwise.} \end{cases}$$

To the best of our knowledge, all existing algorithms use the degenerate case where $k = 1$ and the mutator count is either 0 or ∞, in which case the count is simply a flag that indicates whether a pointer to the object has been stored behind the wavefront. That is, immediately after a pointer to an object is stored behind the wavefront, the mutator count is set to a value that cannot be decremented.

Thus we will use this special case when presenting transformations to pre-existing algorithms such as those of Dijkstra, but it should be noted that 2- or 3-bit counts ($k = 3$ or $k = 7$) could be implemented efficiently and would likely provide most of the potential increase in precision available in practice.

EXAMPLE 5.6. Consider the example of Fig. 3 and its corresponding interaction log P^e. Assuming that $LR = \mathcal{U} \setminus \{r1\}$, $IS = \mathcal{U}$, and $FL = \mathcal{U}$, the function $expose^c(P^e)$ using $M_1(o, P)$ exposes B since $M(B, \mathrm{pre}(P^e, 3)) = 1$.

We now show that the mutator count abstraction transformation preserves correctness and reduces precision.

THEOREM 5.7. *The mutator count abstraction, moving an object from $\mathcal{C}_k$ to $\mathcal{C}_{k-1}$ is a correctness-preserving and precision-reducing transformation.*

5.4 The Protection Dimension

Fundamentally, the safety problem of a concurrent collector is to prevent the mutator from "hiding" an object by moving pointers to that object that are ahead of the wavefront to locations behind the wavefront, and then deleting all paths to the object ahead of the wavefront. Therefore, safety can be guaranteed either by considering pointers installed behind the wavefront (installation-based protection), or by considering pointers deleted ahead of the wavefront (deletion-based protection).

Previously known collectors treat all pointers uniformly: so-called incremental update collectors (such as that of Dijkstra) use installation-based protection; snapshot collectors (such as that of Yuasa) use deletion-based protection. However, our framework allows the two approaches to be mixed, subject to an additional restriction.

The protection dimension is an ordered partition of $\mathcal{U}$ into an Installation Set IS and a Deletion Set DS:

$$D_\pi = \langle IS, DS \rangle$$

The objects in IS are said to be *I-protected*, while the objects in DS are said to be *D-protected*.

The partition is restricted such that every live D-protected object is reachable from a sequence of D-protected objects (this is formalized below).

5.4.1 Snapshot-Based Collector

A snapshot-based collector marks as live all objects that were reachable at the start of the collection cycle; objects that become unreachable during the collection cycle are still treated as live [46].

Using the formulation of Section 2.2, a snapshot-based collector can be defined using the following $expose$ function. We use the superscript d to denote the fact that this function is based on deletion, and name the function $expose^d$.

$$expose^d(P) = \{o \mid P_i.kind = \mathbf{M} \wedge P_i.old = o$$
$$(P_i.source, P_i.field) \notin \mathcal{W}^<(\mathrm{pre}(P, i)) \wedge o \in DS \wedge 0 \leq i < |P|\}$$

Given a log prefix P, $expose^d(P)$ returns all objects in DS that were pointed-to by a field that was assigned a new value (possibly **null**) before its was scanned by the collector.

EXAMPLE 5.8. Consider the example of Fig. 3. Assuming that $DS = \mathcal{U}$, and $FL = \mathcal{U}$, $expose^d(P^e) = \{B, C, D\}$.

5.4.2 Combinations of I-protected and D-protected Objects

Using the formulation of Section 2.2, a collector combining protection policies at the granularity of objects can be defined using the following $expose$:

$$expose^{rcd}(P) = expose^{rc}(P) \cup expose^d(P)$$

More importantly, we introduce a transformation which changes an object from I-protected to D-protected.

However, we must place an additional constraint on which objects in a given graph can be transformed from I-protected to D-protected. To guarantee that an object can be safely transformed from I-protected to D-protected, the object has to be transitively protected by a path of D-protected objects.

DEFINITION 5.9 (Valid Protection Sequence). *A valid protection sequence to an object x is a sequence of objects $o_1, \ldots, o_k = x$ such that o_1 is a root object, and for every $1 \leq i < k$, there is a field f of o_i such that $o_i.f = o_{i+1}$ and $o_i \in DS$.*

DEFINITION 5.10 (Eligibility). *Given an object $x \in IS$, we say that x is* eligible *for membership in DS if there exists a valid protection sequence to x.*

Transformation along the protection dimension is significantly more complex than previous transformations because the transformed algorithm makes decisions in its $expose$ function which may be locally more precise and yet are globally less precise. In particular, if a pointer to object o is stored behind the wavefront, and o is I-protected, then o will be exposed. But if o is D-protected, it will not be exposed. But since the object is D-protected, it will either be discovered directly or through an overwritten pointer in its protection sequence.

In practice, all known algorithms assume a strict partition: either all objects existing at the start of a collection cycle are in DS or they are in IS. In our framework it is possible to have a mix of the two. For example, we can use static knowledge such as type information to select all leaf objects and place them in IS (and hence all other objects would then belong to DS).

In order to consider relative precision of algorithms with non-local effects, we need to refine Definition 4.1:

DEFINITION 5.11 (Weak Precision). *Given two collection algorithms C_1 and C_2, we say that C_1 is* weakly more precise *than C_2, denoted $C_1 \trianglelefteq C_2$, when given any global state of C_2 with an interaction log l and where the set* pending *is empty, $expose_{C_1}(l)^\star \subseteq expose_{C_2}(l)^\star$.*

That is, the transitive closure of the objects exposed by C_1 is a subset of the transitive closure of the objects exposed by C_2.

Weak precision is implied by the strong precision of Definition 4.1, which only consider the exposed objects and not their transitive closure. Since the previous transformations have been shown to be strongly precision reducing, they are also weakly precision reducing.

Note that there is a direct analogue between strong and weak precision, and the strong and weak white-black invariants of incremental update and snapshot collectors. In incremental update collectors all objects are I-protected; in snapshot collectors all objects are D-protected.

Under this refined definition, we show that the protection transformation is weakly precision reducing.

THEOREM 5.12. *Given an eligible object $x \in IS$, changing D_π from $\langle IS, DS\rangle$ to $\langle IS \setminus \{x\}, DS \cup \{x\}\rangle$ is a correctness-preserving and weakly precision-reducing transformation.*

5.5 The Allocation Dimension

To guarantee termination, an algorithm must provide a certain level of progress on each collector marking step. However, when objects are allocated white (unmarked), the collector may need to trace through these newly allocated objects. In the worst-case the collector will trace through all of the newly allocated objects, precluding predictable termination with respect to the live data at the start of the collection cycle.

The parametric collector of Fig. 1 uses a non-deterministic choice to exit the main collection loop into a synchronous termination phase. This phase guarantees termination by atomically tracing from all remaining origins, but therefore introduces an unbounded atomic phase which is undesirable.

In this section, we explore alternatives that provide a more predictable termination without requiring an unbounded atomic phase. This is of particular importance for real-time collectors where it is vital to guarantee worst-case pause time.

A more predictable termination can be achieved by avoiding the need for tracing through newly allocated objects. Typically, objects have been allocated white (require tracing-through) or black (assumed to be marked, and thus require no tracing-through). We consider an additional color (yellow) that provides an intermediate point in the trade-off space between precision and termination.

The allocation dimension is an ordered partition of allocated objects from $\mathcal{U}$ into:

$$D_A = \langle WC, YC, BC\rangle$$

In our framework, without restriction, it is assumed that all newly allocated objects are considered to be members of IS, that is, they are I-protected objects.

5.5.1 White Objects

The first approach is well-known and is the least conservative towards marking allocated objects. Objects are allocated *white*, that is, unmarked and unprocessed.

The Apex algorithm allocates *all* objects white. As previously mentioned, the negative impact on termination when allocating white is that the collector may need to trace through these objects. Allocating white is the primary reason for allowing the collector to non-deterministically enter the synchronous termination phase following the **while** loop in Fig. 1. It could enter that phase after a fixed number of iterations of the while loop. In the Apex algorithm, this would result in the worst-case pause time being proportional to the size of the heap.

Unpredictable pause times in the termination of algorithms that allocate white, which is common for incremental update collectors, have been shown experimentally to lead to significant variation in termination time, making them unsuitable for real-time applications.

5.5.2 Yellow Objects

The termination problem introduced by white allocated objects is that the collector needs to trace through these objects to find other objects that are allocated white.

In order to avoid tracing through these white allocated objects, we introduce *yellow* objects. Yellow objects are allocated unmarked, but any references to objects in IS stored into a yellow object will be treated as if the yellow object is behind the wavefront, effectively, detecting references stored into this yellow object. In particular, it means that it is not possible for the mutator to create chains of unmarked allocated objects that the collector must "chase".

Yellow objects are not to be confused with grey objects, where the object is marked upon allocation yet the wavefront is not updated [30]. Unlike grey objects and similarly to white objects, yellow objects can die during the collection cycle.

The WC to YC transformation can significantly reduce the termination problems associated with allocating white: it eliminates the requirement on the collector to trace through a yellow object. To that end, when the collector encounters an unmarked yellow object, it marks the object, but unlike white objects, does not place its fields in $pending$.

In the special case where all objects are allocated yellow predictable termination can be achieved without the synchronous termination phase after the while loop.

While it is possible to place yellow objects in SR, they are meant to be in LR. In our framework, placing some of them in SR could be thought as a way to process some allocated objects atomically and some incrementally. Placing all of them in SR would effectively mean scanning through these objects atomically and that would make the worst-case pause time similar to that of white allocated objects.

To intuitively understand why the WC to YC transformation leads to a less precise algorithm, consider the following simple example.

EXAMPLE 5.13. Consider an object A for which the WC to YC transformations is applied. Now consider the following sequence of operations: (i) allocate the object A; (ii) store a pointer from A to an unmarked object $B \in IS$; (iii) delete all other pointers to B except the pointer from A; (iv) delete all pointers to the object A, making A unreachable.

After the object A becomes unreachable in the last step, if A is treated as a yellow object, object B will be retained. However, object B will not be retained if A is allocated white.

THEOREM 5.14. *The WC to YC transformation is a correctness-preserving and precision-reducing transformation.*

5.5.3 Black Objects

Termination predictability can be further improved by taking allocated objects out of consideration and allocating them as *black*. Allocating objects as *black* means that these objects are assumed to be live for the current collection cycle. Informally, such objects could be thought of as yellow objects which are allocated marked.

Moving an object from YC to BC will lead to even less work for the collector as it does not need to do additional work for the object. Real-time collectors such as Metronome [4] choose to allocate all objects black.

THEOREM 5.15. *The YC to BC transformation is a correctness-preserving and precision-reducing transformation.*

6. Collector Instantiations

In this section, we explore a small subset of the space of concurrent collection algorithms along the dimensions of Section 5. The space we consider is depicted in Fig. 4. We will typically explore algorithms at the end points of a dimension. That is, we consider the sets to be either $\mathcal{U}$ or $\emptyset$. For simplicity of presentation, we assume that all derived algorithms use an abstracted wavefront (see Section 5.1.1) and that the mutator count is abstracted with $k = 1$.

We first instantiate several collectors which are very similar to some of the well-known algorithms and we show where they fit into the lattice. We then discuss several new practical algorithms. The names of the new algorithms are depicted in boldface in Fig. 4.

In our parametric framework constructing new algorithms is a matter of choosing values over the various dimensions. For example, only recently a collector which uses a precise wavefront definition has been introduced in [18]. We can instantiate similar collectors which use a precise wavefront definition by setting FL to $\mathcal{U}$.

In the figure, we use a tuple of the form (SR, IS, FL, WC, BC) to define the point of the algorithm along the dimensions of Section 5. The values for other sets along each dimension are defined as complements using the values in the tuple, e.g., if WC= $\emptyset$ and BC= $\emptyset$, then YC= $\mathcal{U}$.

In the tuple, we use the set $stacks$ to denote the set of stack objects, and $\mathcal{A}$ to denote the set of newly allocated objects. Additionally, every edge represents a precision order relation (strong or weak depending on which algorithms we are comparing).

From a performance engineering point of view, it is important to choose the values across each dimension appropriately. This by itself presents an important item for future work : which concurrent collector should be used based on the particular application characteristics. For example, it may be known that certain fields have a high mutation rate and therefore it is preferable to perform rescanning on these fields, rather than counting from them. Also, it may be known that certain types of objects tend to die young and therefore it may be preferable to allocate these as white objects.

6.1 Classical Algorithms

In this section, we show how variations of well-known algorithms are expressed in our framework. It should be noted that these are adaptations to our concurrency model. That is, the algorithms follow the predefined skeleton of Fig. 1. The classical algorithms as presented in [30] are usually more elaborate due to complications arising from a non-atomic write barrier.

A Steele-style algorithm can be derived from the Apex collector by applying the *wavefront abstraction* transformation to all objects. Such transformation makes sense when the cost of tracking the progress of the collector at the field level outweighs the benefits of increased precision.

A Dijkstra-style algorithm is derived from the Steele-style collector by moving all objects except stacks along the D_P dimension, that is, from SR to LR. It is less precise than the Steele-style collector. The transformation makes sense when a significant amount of heap behind the wavefront has been mutated by a relatively small number of mutations and rescanning will require atomic processing of that memory.

The Yuasa-style algorithm is the least precise of the three existing algorithms, it allocates all objects black (i.e. BC= $\mathcal{U}$) and in addition all existing objects are D-protected.

6.2 New Algorithms

Fig. 4 contains several new algorithms of practical importance. In this section we informally describe some of those new collectors.

The Steele-YC collector is derived from the Steele-like algorithm by applying the *WC to YC* transformation to all allocated objects. This algorithm reduces the likelihood that the synchronous termination phase will be required, thus addressing the main issue of algorithms that use a Steele-like write barrier (regardless of the granularity of rescanning) such as [11, 7]. The disadvantage of this collector is that it might retain more unreachable objects than a pure Steele-like collector.

The Steele-BC algorithm makes an even more conservative assumption in regards to allocated objects, and allocates them as black. This leads to an opportunity to reduce the work for termination even further while still retaining relatively high precision for existing objects. This algorithm could be beneficial for applications where most of the allocated objects are long lived (i.e. do not die during the collection cycle), such as the mature space of generational collectors.

The precision of the Steele-BC algorithm can be reduced further (and hence the potential for concurrency is increased) by moving along the D_P dimension, moving all objects but stacks from SR to LR resulting with the Dijkstra-BC algorithm.

The Steele-D algorithm, derived from Steele by moving "to the right" on the D_π dimension. The algorithm Steele-D-YC which is derived from Steele-D speeds up termination of the collector but at the cost of reduced precision due to yellow allocated objects.

7. Related Work

In previous work [44] we observed a common structure between concurrent collectors and suggested that they can possibly be viewed as instances of a more abstract collector. However, the paper effectively contained two very complex abstract algorithms, and a few discontinuous "transformations" where their application was only described informally. Moreover, the resulting collectors could not be related.

In [10], separation logic is used to prove the correctness of a stop-the-world copying garbage collector. However, with the extension of separation logic to concurrency [14], it may be possible to formally prove useful algorithms generated from this work.

Another work modeling collectors is [12]. In this paper, the authors use CCS to specify a stop-the-world collector, and temporal logic to specify its liveness and safety properties. However, the presented algorithm is not concurrent and although the collector is specified in CCS, there is no attempt at verifying the presented algorithm. The authors do note however that proving the correctness of a concurrent collector would be even more challenging.

Several works formally verify the correctness of Ben-Ari's and Dijkstra's algorithms [9, 21]. The focus of Ben-Ari's algorithm is correctness rather than efficiency. Ben-Ari's algorithm has made further simplifications to Dijkstra's algorithm with the sole purpose of having an algorithm which is easier to prove. However, both of these algorithms are not practical because their worst-case time complexity is quadratic in the size of the heap.

In [38], Ben-Ari's algorithm is verified for both single and multi-mutator systems using Owicki-Gries's logic in the HOL theorem proving system. In the work of [25], again Ben-Ari's algorithm is verified using the PVS theorem proving system. Similar work has been done by [39], where he proves Ben-Ari's algorithm but this time in Boyer-Moore's theorem prover. In [28], Dijkstra's algorithm has been verified again in the PVS theorem prover. The paper of [15], proves Ben-Ari's algorithm using the B and Coq systems. These works are complementary to ours in the sense that they concentrate on formally proving a particular collector algorithm. The works of [19, 20] define a framework to describe generational and conservative collectors. However, it only deals with stop-the-world algorithms. In the future, we plan on extending our work to deal with moving collectors and it may be possible to integrate some of the ideas of [19].

Another complementary approach is the work of [36] which uses the notion of evolving specifications. It starts with a simple and non-executable declarative specification and extends it to a more complicated and executable design. This strategy does not explore an algorithm space nor does it provide any insight on the relationships between algorithms. However, it presents a structured approach for deriving correct algorithmic specifications starting from simpler and more intuitive models.

Another transformational approach to collectors can be found in [20]. The authors use the SETL wide spectrum language to specify an initially correct and inefficient implementation of a stop-the-world collector. Through loop fusion and formal differentiation transformations, they obtain a more precise implementation of a well-known stop-the-world algorithm. The transformations in our work are specific to the world of concurrent marking collectors.

8. Conclusions and Future Work

In this paper we presented a parametric framework for deriving various correct concurrent garbage collection algorithms. Our framework is based on an initial algorithm serving as a starting point for the derivation process, and a set of correctness-preserving object-level transformations. We explore a space of concurrent GC algorithms by repeated application of our transformations. We also introduce a definition of relative precision which allows us to formally relate algorithms obtained in this framework.

In the future, we plan on extending this work to automatically derive practical synchronization skeletons from our trace-based collectors, as well as relaxing the atomicity constraints of the current parametric algorithm.

Acknowledgements

We would like to thank Noam Rinetzky for reading earlier drafts of this paper and for many helpful comments. We would also like to thank the anonymous reviewers whose comments significantly improved this paper.

A. Proofs

In the following, we abuse notation and use the Kleene star to denote the transitive closure of objects reachable from a given set of objects. For example, we write S^* to denote all objects that are transitively reachable from the set of objects S.

A.1 Collector Invariants and Proof Methodology

The correctness of algorithms in our framework hinges on $expose(l)$ exposing all hidden origins. An algorithm in our framework is correct if and only if the following invariants hold immediately after computing the set of origins by performing $expose(l)$:

I-Invariant $expose(l)$ contains all unmarked objects in IS pointed-to by a marked object.

D-Invariant any unmarked object in DS is either returned by $expose(l)$ or is reachable from a DS object in $expose(l)$ by a path of objects in DS.

These invariants imply the intuitive notion of the algorithm marking a superset of all objects which are required to determine correct transitive reachability.

Our proofs work by showing that the transformations preserve the above invariants. We will denote as C_1 the algorithm before the transformation is applied to a single object o, and C_2, the algorithm after the transformation is applied. The proofs use the correctness of C_1 to show the correctness of C_2. Moreover, the correctness proofs also show that C_1 is more precise than C_2.

We compare the executions of C_1 and C_2 starting from the same initial heap H, sequence of mutations M, and set of roots R.

Besides certain interleavings in the white-to-yellow transformation, in our framework, algorithms only differ in their $expose(l)$ function and therefore we compare their executions by comparing the corresponding $expose(l)$ at the point of divergence of the two traces (that is, when $pending$ is empty). For all transformations except white-to-yellow, the divergence point occurs when the `mark()` procedure has finished and the algorithms proceed into the `addOrigins()` procedure and compute $expose(l)$.

Let $expose_{C1}$ denote the origins computed by C_1 and $expose_{C2}$ the origins computed by C_2 with the same log l.

Certainly, if $expose_{C1} = expose_{C2}$, then the algorithms do not diverge and they can continue to execute in a lock-step. If throughout the entire execution, the algorithms do not diverge, then the execution of C_2 is identical to the execution of C_1 and hence is shown to be correct. Intuitively, in this case, the algorithms are of the same precision.

However, if $expose_{C1} \neq expose_{C2}$, the following invariants hold (we use the subscripts C_1 and C_2 to denote the various sets in C_1 and C_2 respectively):

- $marked_{C_1} = marked_{C_2}$
- the I-invariant and the D-invariant hold for C_1

The first step to showing that C_2 is correct is proving that I- and D- invariants hold for C_2 at the point of $expose$. For all but the protection and white-to-yellow transformations, we prove this by showing that $expose_{C1} \subset expose_{C2}$. This also shows that $C_1 \sqsubseteq C_2$.

The second step of the proof involves reasoning about the continuation of C_2. That is, we need to find a corresponding witness trace which is also correct so that at the next point of $expose$, we can repeat this process. In all proofs except the protection and white-to-yellow transformations, the correct witness trace is basically the restart of C_1 with the $origins$ resulting from $expose_{C2}$ of C_2. The C_1 algorithm can be restarted with the new state, because the I- and D- invariants are preserved by the transformations.

Due to space restrictions we only include the proof of the wavefront transformation in this paper, other proofs follow the same methodology and can be found in [43].

A.2 Wavefront Abstraction

The following proof shows the correctness of the wavefront abstraction transformation which takes a single object $o \in FL$ in C_1

and moves that object so that $o \in OL$ in C_2. It also shows that C_2 is less precise than C_1 according to 4.1.

Proof:(Theorem 5.2) (sketch)

The application of this transformation on object o potentially affects the marking decision for heap objects other than o, but does not affect the marking decision for o itself. This is because the transformation takes effect once object $o \in marked$ and at least one $(o, field) \in \mathcal{W}^>$.

For this transformation, the divergence point always occurs after a call to $expose$ in `addOrigins()`.

Let P be the common prefix of C_1 and C_2 just before $expose$ is called with an interaction log l. At the point after the call to $expose$ where $expose_{C2} \neq expose_{C1}$, the computation $M^-(x, P)$ for any object $x \in IS$ indicates that any pointer installation to x into o will be returned by $expose$, provided that at least one $(o, field) \in \mathcal{W}$ and not all fields of $o \in \mathcal{W}$ at the time of the mutator operation. When object o is in that state, no destruction of a pointer to x in o can affect $M^-(x, P)$. This is indicated by the requirement for $(P_i.o, P_i.field) \in \mathcal{W}^<(\mathrm{pre}(P, i))$ in $M^-(x, P)$.

In addition, operations on objects in DS also affect the return of $expose$. If a mutator destroys a pointer to an object $d \in DS$ such that $(o, field) \in \mathcal{W}$ and $(o, field) \notin \mathcal{W}^<$, then this pointer would not be returned by $expose_{C1}$, but will be returned by $expose_{C2}$.

In the case where $o \in SR$, due to installations of IS objects into o, it is possible that more rescanning work will be done in $expose_{C2}$ for o. However, processing the additional fields can only return a superset of the objects returned by $expose_{C1}$.

It is worth noting that although $expose_{C1} \subset expose_{C2}$, we cannot detect whether the objects in $expose_{C2}$ - $expose_{C1}$ are unreachable. We have constructed examples which show that it is possible for all such objects to be unreachable or for all objects to be reachable or for the mix of the two to occur. However, because $expose_{C1} \subset expose_{C2}$, we can deduce that at the point in the trace right after the call to $expose$ in C_2, the I- and D- invariants are satisfied. This is clear because the invariants are satisfied at the same point in $expose$ of C_1 and by the subset relation we can now trivially conclude that they also hold at the corresponding point in C_2 since adding pointers cannot cause an invariant violation.

Because the invariants are satisfied at this point, we can restart the execution of C_1 with the origins as returned by $expose_{C2}$. Additional origins cannot violate the invariants and we therefore consider this to be a safe witness trace. We compare the possible continuations of C_2 to the restarted trace of C_1 with additional origins. However, from this point on, all fields of $o \in \mathcal{W}^<$ and $o \in marked$. That is, o will behave identically in both, the set of restarted C_1 traces versus the set of traces representing the continuations of C_2. Subsequently, the traces of the safe restarted algorithm of C_1 are exactly the same as the continuations of C_2 and hence we can deduce that C_2 is correct.

The effect of this transformation is manifested at the first point of divergence (where $pending(C_2)$ is empty) where $expose_{C1} \subseteq expose_{C2}$. In any subsequent points where $pending(C_2)$ is empty, $expose_{C1} = expose_{C2}$ which is the necessary condition to establish that $C_1 \sqsubseteq C_2$. However, the converse statement clearly does not hold, that is, $C_2 \not\sqsubseteq C_1$.

References

[1] Appel, A. W., Ellis, J. R., and Li, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices*, *23*, 7 (July), 11–20.

[2] Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (Oct 2003), ACM Press, pp. 269–281.

[3] Bacon, D. F., Attanasio, C. R., Lee, H. B., Rajan, V. T., and Smith, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 92–103.

[4] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[5] Baker, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[6] Baker, H. G. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (Mar. 1992), 66–70.

[7] Barabash, K., Ossia, Y., and Petrank, E. Mostly concurrent garbage collection revisited. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (Oct 2003), ACM Press, pp. 255–268.

[8] Barth, J. M. Shifting garbage collection overhead to compile time. *Commun. ACM 20*, 7 (July 1977), 513–518.

[9] Ben-Ari, M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6*, 3 (1984), 333–344.

[10] Birkedal, L., Torp-Smith, N., and Reynolds, J. C. Local reasoning about a copying garbage collector. In *Proc. of the Thirty-first ACM Symposium on Principles of Programming Languages* (Venice, Italy, 2004). *SIGPLAN Notices*, *39*, 1, 220–231.

[11] Boehm, H.-J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, 1991). *SIGPLAN Notices*, *26*, 6, 157–164.

[12] Bowman, H., Derrick, J., and Jones, R. E. Modelling garbage collection algorithms. In *Proceedings of International Computing Symposium* (January 1994). Also in ALP-UK International Workshop on Concurrency in Computational Logic.

[13] Brooks, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

[14] Brooks, S., and O'Hearn, P. Resources, concurrency and local reasoning. *Fifteenth International Conference on Concurrency Theory 3170* (2004).

[15] Burdy, L. B vs. Coq to prove a garbage collector. In *Fourteenth International Conference on Theorem Proving in Higher Order Logics: Supplemental Proceedings* (Sept. 2001), R. J. Boulton and P. B. Jackson, Eds., Report EDI-INF-RR-0046, Division of Informatics, University of Edinburgh, pp. 85–97.

[16] Cheadle, A. M., Field, A. J., Marlow, S., Peyton Jones, S. L., and While, R. L. Non-stop Haskell. In *Proc. of the Fifth International Conference on Functional Programming* (Montreal, Quebec, Sept. 2000). *SIGPLAN Notices*, *35*, 9, 257–267.

[17] Cheng, P., and Blelloch, G. E. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (Jun 2001), ACM Press, pp. 125–136.

[18] Click, C., Tene, G., and Wolf, M. The pauseless GC algorithm. In *Proc. of the First ACM/Usenix Conference on Virtual Execution Environments* (Chicago, Illinois, June 2005).

[19] Demmers, A., Weiser, M., Hayes, B., Boehm, H., Bobrow, D., and Shenker, S. Combining generational and conservative garbage collection: framework and implementations. 261–269.

[20] Dewar, R. B. K., Shirar, M., and Weixelbaum, E. Transformational derivation of a garbage collection algorithm. *ACM Trans. Program. Lang. Syst. 4*, 4 (1982), 650–667.

[21] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM 21*, 11 (1978), 966–975.

[22] Doligez, D., and Gonthier, G. Portable, unobtrusive garbage

collection for multiprocessor systems. In *Conf. Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Jan. 1994), pp. 70–83.

[23] DOLIGEZ, D., AND LEROY, X. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conf. Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Jan. 1993), pp. 113–123.

[24] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.

[25] HAVELUND, K. Mechanical verification of a garbage collector. In *Fourth International Workshop on Formal Methods for Parallel Programming: Theory and Applications* (San Juan, Puerto Rico, 1999), J. D. P. Rolim et al., Eds., vol. 1586 of *Lecture Notes in Computer Science*, pp. 1258–1283.

[26] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[27] HUDSON, R. L., AND MOSS, E. B. Incremental garbage collection for mature objects. In *Proc. of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*.

[28] JACKSON, P. B. Verifying a garbage collection algorithm. In *Theorem Proving in Higher Order Logics, 11th International Conference* (1998), LNCS, Springer-Verlag, pp. 225–244.

[29] JOHNSTONE, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[30] JONES, R., AND LINS, R. *Garbage Collection*. John Wiley and Sons, 1996.

[31] LAMPORT, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.

[32] LAROSE, M., AND FEELEY, M. A compacting incremental collector and its performance in a production quality compiler. In *Proc. of the First International Symposium on Memory Management* (Vancouver, B.C., Oct. 1998). *SIGPLAN Notices*, *34*, 3 (Mar., 1999), 1–9.

[33] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (Oct 2001), ACM Press, pp. 367–380.

[34] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[35] NORTH, S. C., AND REPPY, J. H. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture* (Portland, Oregon, Sept. 1987), G. Kahn, Ed., vol. 274 of *Lecture Notes in Computer Science*, pp. 113–133.

[36] PAVLOVIC, D., PEPPER, P., AND SMITH, D. R. Colimits for concurrent collectors. In *Verification: Theory and Practice, essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday* (2003), vol. 2772 of *LNCS*, Springer-Verlag, pp. 568–597.

[37] PIXLEY, C. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing 3, 1 6*, 3 (Dec. 1988), 41–49.

[38] PRENSA NIETO, L., AND ESPARZA, J. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In *Mathematical Foundations of Computer Science (MFCS 2000)* (2000), M. Nielsen and B. Rovan, Eds., vol. 1893 of *LNCS*, Springer-Verlag, pp. 619–628.

[39] RUSSINOFF, D. M. A mechanically verified incremental garbage collector. *Formal Aspects of Computing 6*, 4 (1994), 359–390.

[40] SAGONAS, K., AND WILHELMSSON, J. Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language. In *Proc. of the Fourth International Symposium on Memory Management* (Vancouver, Canada, 2004), ACM Press, pp. 1–12.

[41] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[42] STEELE, G. L. Corrigendum: Multiprocessing compactifying garbage collection. *Commun. ACM 19*, 6 (June 1976), 354.

[43] VECHEV, M., YAHAV, E., AND BACON, D. Parametric generation of concurrent collection algorithms: Online supplement. http://www.cl.cam.ac.uk/users/mv270/cgcproofs.ps.

[44] VECHEV, M. T., BACON, D. F., CHENG, P., AND GROVE, D. Derivation and evaluation of concurrent collectors. In *Proceedings of the Nineteenth European Conference on Object-Oriented Programming* (Glasgow, Scotland, July 2005), A. Black, Ed., Lecture Notes in Computer Science, Springer-Verlag.

[45] WADLER, P. L. Analysis of an algorithm for real time garbage collection. *Commun. ACM 19*, 9 (1976), 491–500.

[46] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.