Portable, Unobtrusive Garbage Collection for Multiprocessor Systems

Damien Doligez
École Normale Supérieure
INRIA Rocquencourt
École Polytechnique
Damien.Doligez@inria.fr

Georges Gonthier*
INRIA Rocquencourt
78153 LE CHESNAY CEDEX
FRANCE
Georges.Gonthier@inria.fr

Abstract

We describe and prove the correctness of a new concurrent mark-and-sweep garbage collection algorithm. This algorithm derives from the classical on-the-fly algorithm from Dijkstra *et al.* [9]. A distinguishing feature of our algorithm is that it supports multiprocessor environments where the registers of running processes are not readily accessible, without imposing any overhead on the elementary operations of loading a register or reading or initializing a field. Furthermore our collector never blocks running mutator processes except possibly on requests for free memory; in particular, updating a field or creating or marking or sweeping a heap object does not involve system-dependent synchronization primitives such as locks. We also provide support for process creation and deletion, and for managing an extensible heap of variable-sized objects.

1 Introduction

Concurrent garbage collection has a well-deserved reputation for being a tough problem. This is evidenced by the discrepancies between the state of theory and practice in this area. As we shall see below, the published proven algorithms often contain simplifying assumptions that cannot be met in practice in a multiprocessor system, because this would either impose unbearable overhead on the mutator processes, or require a degree of hardware and/or operating system support that compromises portability. Implemented systems that do not fall in the latter two categories often rely on incompletely formalized algorithms, which generally means buggy algorithms, given the subtleness of the correctness proofs.

To our knowledge, and as we shall argue below, all published concurrent collectors fall in one of the above categories, and thus fail to meet at least one of the basic requirements for portable, effective garbage collection on multiprocessors. In fact the only proposal that even attempts to meet these requirements is the Doligez-Leroy hybrid collector [10]. Unfortunately, the algorithm they proposed was incompletely specified and, perhaps not unexpectedly, buggy. In this paper, we redress this state of affairs by fully describing and proving a concurrent garbage collection algorithm that meets the requirements for the Doligez-Leroy collector architecture. This turns out to be much more intricate than the simple adaptation of the concurrent mark-and-sweep algorithm [9] outlined in [10]. We still expect the experimental results of [10] to hold for our model, because a slightly modified (debugged) version of their algorithm fits in our model.

In the next section we spell out the basic portability and efficiency requirements for a collector for multiprocessors. We show why previous algorithms fail at least one these requirements, and how these requirements coincide with those of the Doligez-Leroy architecture. In section 3 we describe the basic algorithm of [9], and expose a series of counterexamples to explain why a straightforward adaptation of this algorithm to multiple mutators would not work; we also address some efficiency issues. In section 4 we describe the basic procedures of our algorithm. In section 5 we describe the extensions to handle process and heap management. Finally, in section 6 we present a sketch of the correctness proof of the algorithm. This proof is based on a formal model of the algorithm, expressed in a Unity/TLA-like format; this model, listed in the appendix, also covers the extensions to the basic algorithm.

2 The requirements

Our basic requirements are essentially shaped by the following "facts of life" about multiprocessors:

1 **Registers are local.** Even on a uniprocessor, it can be hard to track the machine registers of a running process. On a multiprocessor this is next to impossible; furthermore this impossibility extends to the local memory of each processor.

2 **Synchronization is expensive.** Of course any multiprocessor system will provide semaphores and other synchronization devices, but often these will only be available through expensive system calls. Thus a portable collector should use as little synchronization as possible.

3 **Resources are not bounded.** It is unreasonable to forbid system calls to grow the heap. And just as unreasonable to make the liveness of the collector depend on exhaustion of the system memory.

[*] This work was partly funded by the ESPRIT Basic Research Action No. 6454 (Project CONFER)

2.1 Actions and overhead

Let us classify the various actions that can be taken by a mutator thread:

a) **move** data (including heap pointers) between registers and/or local memory
b) **load** a field in a register (dereference a heap pointer)
c) **reserve** free memory for future new heap objects
d) **create** a heap object in previously reserved memory
e) **fill** a field in a new heap object
f) **update** a field in an existing heap object
g) **cooperate** with the collector (see below)
h) **mark** heap objects referenced by registers and local memory (this is a special case of *g*)

We break up the usual "**allocate** a heap object" action into separate *c* (**reserve**), *d* (**create**), and *e* (**fill**) actions. The *c* actions are a necessary evil: the *c* actions of all active mutator threads all contend for the free memory provided either by the collector or the system. Hence *c* actions must call on synchronization primitives, which may be expensive (fact 2). Having separate *d* and *c* actions allows us to amortize the synchronization overhead by keeping local pools for each thread, and "batching" operations on the free list.

The *e* (**fill**) and *f* (**update**) action types correspond to the same physical operation—a **store** in the heap. We distinguish them because they have different frequencies and preconditions. In a *e* action the modified heap object is still private to the mutator thread, while in an *f* action it may shared with other threads. Hence *f* actions are harder to implement, so it is fortunate that in practice they are not very frequent: having an efficient garbage collector encourages the creation of new objects to hold new results, rather than the hazardous reuse of temporaries.

The *g* and *h* actions (**cooperate** and **mark**) are an unavoidable consequence of fact 1. Obviously, a reasonable algorithm should ensure that they do not disrupt the mutator threads significantly.

Let us say a garbage collection algorithm is *unobtrusive* if it meets the following conditions:

(i) It adds *no* overhead to the very frequent mutator actions of type *a*, *b*, and *e*.

(ii) It only imposes synchronization overhead on type *c* mutator actions, for which it is unavoidable.

(iii) Mutator actions of type *g* and *h* are executed only at a mutator thread's convenience.

(iv) For any mutator, the total overhead of *g* and *h* actions for a full collection cycle is bounded, a "full collection cycle" being the period that ends when the collector has reclaimed all currently unused heap objects.

(v) Full collection cycles always terminate, regardless of increases in the heap size or the number of processes.

Requirement (i) is a basic efficiency constraint. Any useful overhead has to include at least one heap reference, which would take as much time as a **load** or **fill** action, and possibly twenty times as much as a **move** action. Requirement (ii) is a direct consequence of fact 2: less frequent actions of type *d*, *f*, *g*, or *h* can incur moderate overhead, but by fact 2 synchronization cannot be considered "moderate overhead". Requirement (iii) means a mutator does not have to be ready to cooperate with the collector at all times: it can restrict cooperation to well-defined points in its code. As a consequence, transient states are allowed in the mutators, which is required by efficient code. Another consequence is that real-time garbage collection becomes possible: a mutator may exclude cooperation overhead for some time-critical part of its code. Requirement (iv) bounds the amount of **cooperate** and **mark** overhead a mutator must incur before getting any significant payback from the collector. Requirement (v) simply takes fact 3 into account.

Put all together, requirements (i)–(v) state that the performance of the concurrent collection algorithm should be roughly comparable to that of a sequential collector, but without the disrupting pauses (requirements (iii) and (iv)).

On the other hand, we limit ourselves to rather weak efficiency requirements on the collector. The basic collector actions of marking, unmarking, or reclaiming a heap object should not require synchronization; the total amount of collector work for a full cycle should be proportional to the maximal heap size and the total number of processes, at least in absence of the (presumed infrequent) **update** actions. Stronger requirements (e.g., removing the provision for **update** actions) would imply additional mutator overhead that is not justified in practice, as the collector is rarely the bottleneck, especially in the setting of [10] (which we outline in subsection 2.3.) Also, allocating more memory will compensate for a slower algorithm, up to a certain point [13]. As a last resort, parallelizing the collector is also an option [15, 19].

2.2 Where all else fails

As elementary as constraints (i–v) seem to be, they all but rule out copying garbage collection algorithms that relocate objects in order to eventually reclaim entire areas at once. By fact 1, the collector cannot update the local memory of running processes to reflect the relocation, so it must arrange for the processes to do this updating on their own. The known schemes for doing this invariably break at least one of (i)–(iii). Doing a test [3] or a second indirection [6, 20] for each heap access obviously breaks (i) for **load** actions. Using virtual memory page protections to bypass the test [2] breaks (ii) and (iii): mutators incur the possibly high page fault overhead at random times.

Some promising systems were recently proposed for incremental and concurrent copying collection [5, 18]; however they require a global rendez-vous of all the mutators in each collection cycle, breaking either (ii) or (iii).

So it seems we must give up relocation to get an unobtrusive concurrent collector, which leaves us with mark-and-sweep algorithms. Unfortunately, the basic on-the-fly mark-and-sweep algorithm [9] does not account for fact 1—it assumes that the local pointers of a thread only point to otherwise accessible objects. This could only be enforced by imposing overhead on **move** actions [12], and thus breaking (i). All the derivatives of [9] suffer this fatal flaw [4, 14]. Furthermore these algorithms only support a single mutator; the multiple mutator version [15] explicitly requires synchronization overhead, breaking (ii).

2.3 The Doligez-Leroy design

It may seem a pity that we had to rule out the copying algorithms, as only they can deal with the large amount of short-term garbage generated by functional languages such as ML [1]. As was shown in [10], this dilemma can be solved

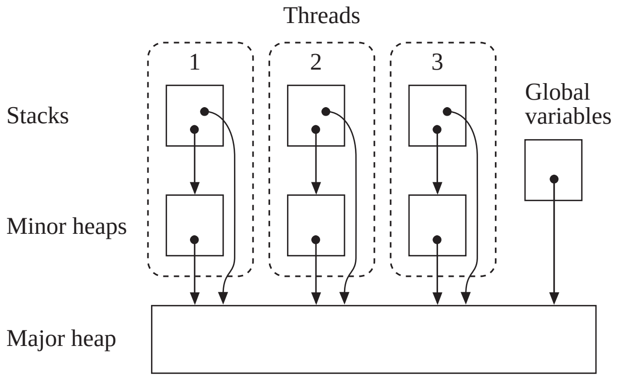

Figure 1: **The Doligez-Leroy architecture**

by organizing the local memory of each processor into a stack and heap (Figure 1), and running a stop-and-copy collector locally to do generation scavenging. This stop-and-copy collector does not break requirement (ii) because it only stops the local thread. Copying is especially adapted to the young generation because most of the young objects are garbage, and a copying collector works best in that case. Thus most of the garbage is reclaimed by the mutator threads themselves and the major collector is only concerned with long-lived and mutable objects.

With this architecture the previously overlooked fact 1 becomes glaringly obvious: clearly the global collector cannot trace the local heap without the cooperation of the mutator thread. In a highly portable system such as Caml-Light [17], fact 2 is a matter of course. So [10] had implicitly laid out requirements (i)–(iv) for their global collector. They only missed on requirement (v) and, as we will see below, on some subtle implications of requirements (i)–(iv).

3 The basic algorithm and its shortcomings

In this section, we expose the Dijkstra *et al.* algorithm [9] (hereafter called the "basic algorithm") and a series of counterexamples that show why a straightforward adaptation to multiple mutators cannot work, and we describe some efficiency problems of this algorithm.

3.1 The basic algorithm

First we describe the heap data structure, then the operations of the collector and mutators.

The heap is a fixed array of objects, each of which has a fixed number of fields.

```
const $End, MaxIndex \in NAT$
type  $ADDR \equiv \{0, \ldots, End-1\}$
      $INDEX \equiv \{0, \ldots, MaxIndex\}$
var $heap \in$ array $[ADDR, INDEX]$ of $OBJECT$
```

There is a fixed set of globally accessible locations. One of them is the head of the free list, which uses the usual linked list implementation.

```
const $Globals \in$ set of $ADDR$
```

The tracing status of heap objects is modeled by a separate *color* array:

```
Mark:        foreach $x \in Globals$ do $MarkGray(x)$
Scan:        repeat
               $dirty \leftarrow$ false
               foreach $x \in ADDR$ do
                 if $color[x] = Gray$ then
                   $dirty \leftarrow$ true
                   foreach $i \in INDEX$ do
                     $MarkGray(heap[x, i])$
                   $color[x] \leftarrow Black$
             until $\neg dirty$
Sweep&Clear: foreach $x \in ADDR$ do
               if $color[x] = White$ then
                 append $x$ to the free list
               else if $color[x] = Black$ then
                 $color[x] \leftarrow White$
```

Figure 2: **The basic collector**

```
type $COLOR \equiv \{White, Gray, Black\}$
var  $color \in$ array $[ADDR]$ of $COLOR$
init $\forall x \in ADDR, color[x] = White$
```

White objects are unmarked. Their reachability status is unknown. *Black* objects are traced. They are marked (reachable) and their sons are marked. *Gray* objects are marked but their sons have not been marked yet.

Because the free list head is one of the globally accessible values, the free list is traced by the garbage collector, and allocation is a special case of assignment. In fact, **fill**, **reserve**, **create**, and **update** operations are all instances of a generic **store** operation, implemented as follows:

```
$MarkGray(x) \equiv$
    if $color[x] = White$ then $color[x] \leftarrow Gray$
$Store(x, i, y) \equiv$
    $heap[x, i] \leftarrow y$
    $MarkGray(y)$
```

The collector cycle is divided in four steps (Figure 2):

Mark: Mark objects referenced by global variables.

Scan: Scan the heap for marked (*Gray*) objects, and trace them by marking their sons and *Black*ening them. Repeat the scan as needed to ensure all reachable objects are marked and traced.

Sweep: Reclaim all white objects.

Clear: Unmark all marked objects, establishing the preconditions of the next collector cycle.

The two invariants used in [9] to prove this algorithm are:

- During the *Scan* step, every *White* reachable object is reachable from a *Gray* object.
- At the beginning of the *Sweep* step, there is no *Gray* object.

From these invariants, one can deduce that all reachable objects must be *Black* at the beginning of the *Sweep* step. The second invariant is easy to prove, assuming that *MarkGray* is atomic. For the first invariant, one proves that at most one *Black*-to-*White* pointer exists: the pointer from x to y when the mutator is between the two lines of *Store*.

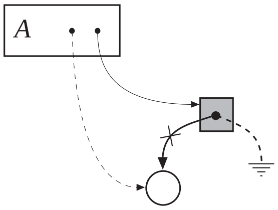

Figure 3: **If we don't mark the old value...**

The algorithm still works when *MarkGray* is not atomic, but the proof is more complex. The detailed proofs can be found in [9].

This algorithm is subject to *floating garbage*, i.e. garbage created during a collector cycle, which will not be reclaimed by this cycle, but by the next one. This means that a "full collection cycle" is composed of two collector cycles.

3.2 Local memory

The basic algorithm fails to take fact 1 into account: the mutator must make sure that its local variables only point to objects that are already reachable from *Globals*. Not only does this mean that all the temporary variables of each mutator must remain visible to the collector at all times, but also that all assignments to these temporary variables incur the overhead of *MarkGray*; thus the basic algorithm fails our requirement (i).

To correct this problem, we decided that the local variables (the *roots*) of the mutators are hidden to the collector and we added, as in [10], a handshake between the collector and each mutator. The *Mark* step of the collector becomes:

```
Mark: foreach $x \in Globals$ do MarkGray($x$)
      issue a call for roots
      wait until all the mutators have answered
```

And the mutators must execute the *Cooperate* procedure from time to time:

```
Cooperate ≡
    if a call for roots is pending then
      call MarkGray on all the roots
      answer the call
```

This constraint seems to preclude calls to foreign functions, which will not call *Cooperate*; the implementation solves this problem by adding a wrapper around such functions. The wrapper synchronizes with the collector and delegates the cooperation work to the collector thread itself. The synchronization is only needed for long-running functions, so its overhead is negligible compared to the running time of the function.

In this new setting, as stated in [10], we have to mark the "old" value of a field before the update, or the collector could reclaim objects that are still in use. This is illustrated by the following counterexample with a single mutator *A* (Figure 3):

C calls for the roots
A grays its only root, □
answers to the call
loads the value ○ of field 0 of □ in a register
sets field 0 of □ to **nil**
C notices that all mutators have answered
blackens □, which completes the *Mark* step
reclaims ○, which is still in use by *A*

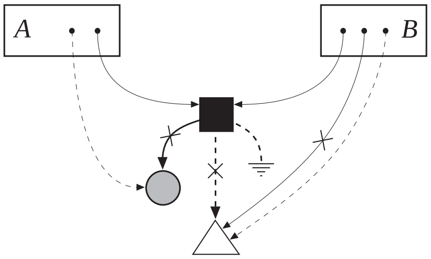

Figure 4: **If we don't pause before marking...**

Marking the old value was an option in the basic algorithm; it was rejected early on, because it obviously generated more floating garbage [9]. However it has also been noted that most of the garbage is floating anyway [21]. This holds even if the garbage were generated randomly, whereas in practice most of the garbage consists of recently allocated objects, which are always marked in the basic algorithm. Hence controlling the allocation color to prevent almost all floating garbage in the *Sweep* and *Clear* steps is more likely to reduce floating garbage effectively.

3.3 Multiple mutators

While [10] concluded correctly that old values have to be marked, they missed an important point: with multiple mutators, it is impossible to get the value of the old object reliably! To do so would require at least an atomic compare&swap [11] and thus violate (ii). The *Store* operation of [10] was:

```
Store($x, i, y$) ≡
    MarkGray($heap[x, i]$)
    $heap[x, i] \leftarrow y$
    MarkGray($y$)
```

This does not work because the assignment is not guaranteed to overwrite the value that was just shaded, as demonstrated by this counterexample with *A*,*B* (Figure 4):

C calls for the roots
A grays its only root □
answers the call
loads field 0 of □ (○)
grays ○ in preparation to a *Store* into field 0 of □
B grays □ and △
answers the call
grays ○, which is already gray
sets field 0 of □ to △
grays △
clears the register pointing to △
C blackens all objects, completing the *Scan* step
performs its *Sweep* step (whitens all objects)
starts a new cycle by calling for the roots
B grays its only root, □
answers the call
reloads field 0 of □ (△)
A resumes its *Store* by setting field 0 of □ to **nil**
grays its roots ○ and □ (□ is already gray)
answers the call
C blackens □ and ○, completing its *Scan* step
reclaims △, which is still in use by *B*

To sum up, first *B* lays a trap for *A* by putting a white object in field 0 of □, then *A* trips the trap by overwriting that field. Our algorithm uses a second handshake before the call for roots to ensure that all traps laid during the *Sweep* step are tripped before the *Mark* step begins.

We have two other counterexamples (only one of which appears in [10]) that show the existence of a trade-off between:

1. adding a third handshake
2. always marking the new value
3. adding some overhead to *Store*

We chose 1 over 2 to avoid the creation of floating garbage during the *Sweep* step, and over 3 because the collector is not the bottleneck, and because 1 enables us to concentrate all the overhead of *Store* before the actual assignment.

3.4 Scan termination

There is one obvious efficiency problem with the basic algorithm: it scans the heap many times to find *Gray* objects during the *Scan* phase. The worst case is even quadratic in the heap size, and it is easily attainable with a list whose cells are in decreasing order, a common case when allocation is in increasing order. Since most of these *Gray* objects were marked by the collector itself, it is easy to add a cache of *Gray* objects to the collector. As long as this cache is not empty, the collector does not need to scan the heap to find *Gray* objects. A further improvement is to turn *dirty* into a global variable and have the mutators set it when they mark an object. This only avoids the last scan, but in practice we only have one or two scans most of the time, so saving one scan is a big win.

Kung and Song [14] use a double-ended queue to avoid the scans completely, but their solution does not work with multiple mutators without synchronization on **update** operations. All the other proved algorithms use repeated scans of the heap.

In the design of [14], objects are marked *Gray* as they are inserted in the queue, which plays the same role as our cache. This policy does not work in our case, as the following counterexample shows (Figure 5):

The last three objects in the heap are △, ○, and □; field 0 of ○ is △ and field 0 of □ is ○; □ is gray while △ and ○ are white, and the collector scan has reached □ while *dirty* is still **false** and the cache is empty. The following can occur:

A loads field 0 of □ (○)
  grays ○ (first step of an assignment to field 0 of □)
C blackens □, as its only field is already gray
  ... thus completing the *Scan* step
A sets *dirty* ← **true** (too late!)
  sets field 0 of □ to **nil**
  loads field 0 of ○ (△)
C reclaims △, which is still used by *A*

Note that it does not help if *A* sets *dirty* ← **true** before marking, as *C* can reset *dirty* and repeat the scan at any time. Hence *C* must add any gray object it encounters to the cache, and must mark those objects *Black* to avoid duplicates. Furthermore, if *C* removes objects from an overflowing cache, it must reset their *color* to *Gray*. (This implies that mutators should *never* write back the *color* of a *Black* object, which prohibits "logical or" implementations of *MarkGray*.)

We have a two-mutator version of this example to show that mutators must set *dirty* ← **true** even when the old value is already gray. This compels us to make x a global variable, and make the mutators test the position of the gray object compared to x before they change *dirty*, so that repeated assignments of the same values do not cause a spurious scan of the entire heap (or even prevent termination of the *Scan* step).

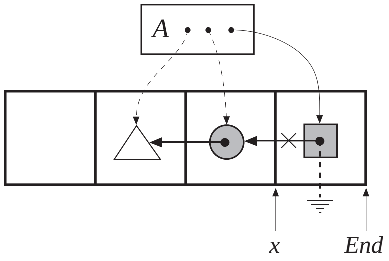

Figure 5: **If we don't trace gray objects...**

4 Our algorithm

In this section we will describe the algorithm in the simple case where the sizes of the heap, the objects, and the set of mutators are all constant. We delay the discussion of heap extension and requirement (v) to section 5, as well as the discussion of variable-sized objects.

We divide the description of our algorithm as follows: first the heap model and the description and evolution of the global variables, then the mutator primitives and finally the collector code.

4.1 The heap and global variables

We will reuse most of the heap model of the basic algorithm, with a number of additions and changes. First of all, there is a fixed set of processes:

const $MaxPid \in NAT$
type $PID \quad \equiv \{0, \ldots, MaxPid\}$

Unlike the basic algorithm, we abstract from the representation of the free memory list, using a multiset rather than a set so that we can prove there are no double insertions. The initial live data must not point to the free list.

var $free \in$ **multiset of** $ADDR$
init $Globals \cap free = \emptyset$
  $\forall x \in ADDR \setminus free\ \forall i \in INDEX,$
   $heap[x, i] \in ADDR \setminus free$

We will use a fourth color, *Blue*, to indicate free locations where a heap object may be created. The corresponding areas are ignored by the collector. All locations in *free* must be *Blue*, but the converse is not always true, as processes may withhold some free memory.

type $COLOR \equiv \{White, Gray, Black, Blue\}$

$$\textbf{init}\quad \forall x \in ADDR, color[x] = \begin{cases} Blue & \textbf{if } x \in free \\ White & \textbf{otherwise} \end{cases}$$

The collector cycle is still divided in the same four steps: *Mark*, *Scan*, *Sweep*, and *Clear*.

All the complex handshake synchronization takes place during the *Mark* and *Clear* steps, although these steps were

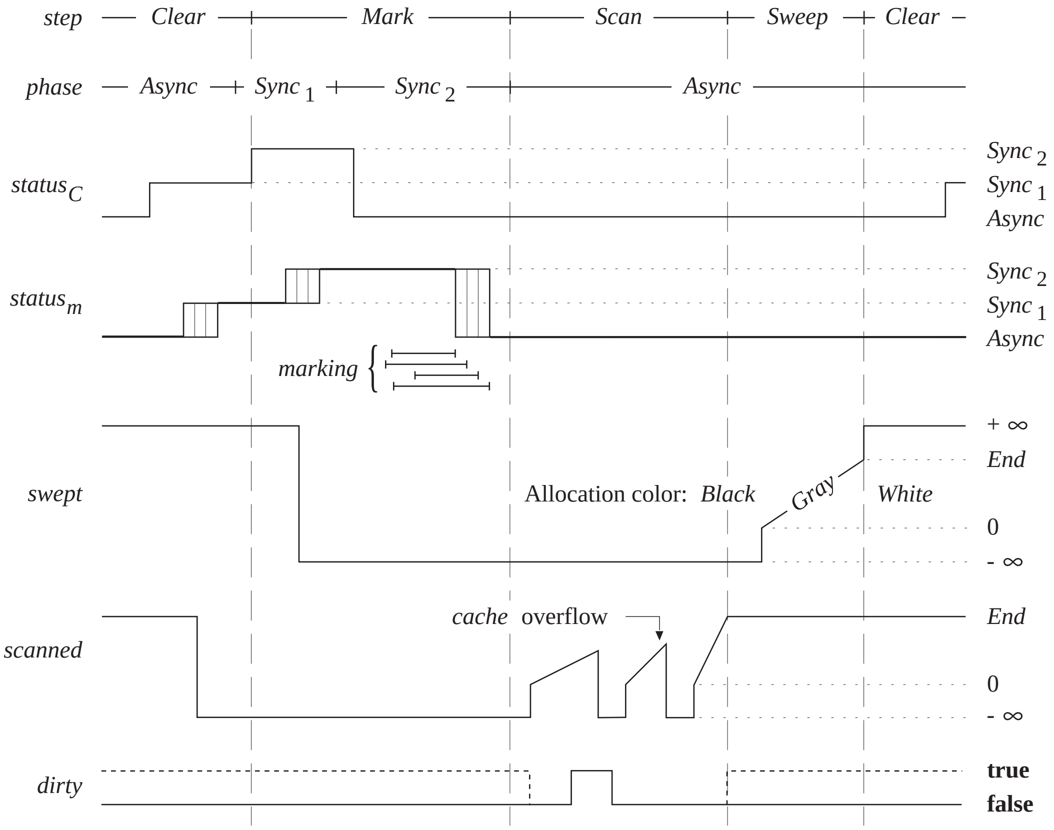

Figure 6: **Timing diagram for global variables**

rather trivial in the basic algorithm. The collector keeps track of the handshake status in a *phase* variable, whose values after the first, second, and third handshake are, respectively,

$Sync_1$ indicating that mutators will only **update** fields with pointers to objects that will be marked in this cycle.

$Sync_2$ indicating that mutators will only **update** fields that point to objects that will be marked in this cycle.

Async indicating that all mutators have marked the objects referenced by their registers at one point in this cycle, whence no reachable object will be reclaimed during this cycle. We will have $phase = Async$ at the beginning of the next cycle.

As illustrated in Figure 6, the *Clear* and *Mark* steps end with the first and third handshakes, respectively; in practice these steps are quite short, hence most of the time we have $phase = Async$.

The collector and all the mutators use a global *status* variable to implement the handshakes:

type $STATUS \equiv \{Async, Sync_1, Sync_2\}$
var $status_C \;= Async \in STATUS$
$\forall m \in PID$, **var** $status_m = Async \in STATUS$

At rest all statuses equal *phase*. The collector initiates a handshake by advancing $status_C$, the mutators respond by following suit, and finally the collector moves *phase* when all have responded. The first two handshakes require no other action from the collectors than completing pending actions (especially **update**); before completing the third handshake mutators must mark all the objects they reference (Figure 6).

The collector must perform a polling loop to complete the handshake. Fortunately, the collector need not be idle during that period for the last two handshakes, as it has other marking or tracing work to perform, and the first handshake is likely to be extremely brief, as it requires almost no work from the mutators.

Finally, three global variables are used to implement the efficiency refinements described in subsections 3.2 and 3.4.

var $swept = +\infty \in ADDR \uplus \{-\infty, +\infty\}$

tracks the progress of the *Sweep* step, and is set to $-\infty$ and $-\infty$ before and after the *Sweep* step, respectively. Mutators test *swept* in **create** and **update** actions to avoid generating floating garbage during the *Sweep* step.

var $dirty \in BOOL$

is used to ensure that the *Scan* step only terminates when all reachable objects have been marked. It is set to **false** by the collector each time it starts a new scan, and reset to **true** by a mutator that detects the scan has missed a gray object, forcing the collector to repeat the scan.

var $scanned = -\infty \in ADDR \uplus \{-\infty\}$

tracks the progress of the *Scan* step, and is reset to $-\infty$ between scans. Mutators test *scanned* before resetting *dirty* to avoid causing spurious scans.

4.2 The mutator actions

The only local variables of a mutator are its multiset of heap pointers, and free memory pool.

```
const  $MaxPool > 0 \in NAT$
var    $pool$     $= \emptyset \in$ multiset of $ADDR$
       $roots$    $\in$ multiset of $ADDR$
init   $roots \cap free = \emptyset$
```

There are two mutator marking actions. *MarkGray* is used before the *Scan* step, otherwise *MarkAndWarn* is used to ensure that a concurrent scan does not miss the marked object (more precisely, to ensure that the collector traces the object at least once during the *Scan* step).

```
$MarkGray(x) \equiv$
    if $color[x] = White$ then $color[x] \leftarrow Gray$
$MarkAndWarn(x) \equiv$
    if $color[x] \neq Black$ then
        $MarkGray(x)$
        if $x \leq scanned$ then $dirty \leftarrow$ true
```

The code for mutator m should execute the *Cooperate* procedure at reasonably close intervals; the overhead is minimal except for the root marking that occurs once per cycle. We could also allow the mutator to relocate a pointer in a new heap object and mark this object rather than the one referenced by the pointer, i.e., the marking can be performed by a local copying collection cycle as in [10].

```
$Cooperate \equiv$
    if $status_m \neq status_C$ then
        if $status_m = Sync_2$ then
            foreach $x \in roots$ do $MarkGray(x)$
        $status_m \leftarrow status_C$
```

Memory reservation is the only action requiring a critical section, introduced here by the **await**...**do**... construct. The **pick** $x \in S$ construct chooses and removes one copy of a random element $x \in S$.

```
$Reserve \equiv$
    await $free \neq \emptyset$ do
        repeat
            pick $x \in free$ do $pool \leftarrow pool \oplus \{x\}$
        until $free = \emptyset \lor |pool| = MaxPool$
```

The *Create* procedure chooses the color of the new object based on the progress of the cycle to minimize floating garbage. The race with the *Sweep* step is resolved with the *Gray* color; we could also defer the decision to the next allocation or handshake. Note how the sentinel values of *swept* simplify the logic.

```
$Create \equiv$
    pick $x \in pool$ do
        $color[x] \leftarrow Black$
        if $status_m \neq Async \lor x < swept$ then
            $color[x] \leftarrow White$
        else if $x = swept$ then
            $color[x] \leftarrow Gray$
        return $x$
```

Although there is no overhead on the fill operation, the mutator must completely fill the fields of an object before marking or using the object, and in any case before marking its roots.

Although the *Update* operation carries the most overhead, it all occurs up front, before the **store** proper. The proof shows that during the *Async* phase, the marking overhead effectively cuts out the field from the collector's tracing space. Hence a mutator repeatedly modifying the same field only needs to incur the **update** once per collector cycle.

```
$Update(x, i, y) \equiv$
    if $status_m \neq Async$ then
        $MarkGray(heap[x, i])$
        $MarkGray(y)$
    else if $swept = -\infty$ then
        $MarkAndWarn(heap[x, i])$
    $heap[x, i] \leftarrow y$
```

4.3 The collector

We assume the size of the collector *cache* is bounded.

```
const  $MaxCache > 0 \in NAT$
var    $cache$    $= \emptyset \in$ multiset of $ADDR$
       $phase$    $= Async \in STATUS$
```

Trace is just the standard *Black* tracing procedure, with the recursion stack made explicit by *cache*, and with overflow handled by the *Gray* color. The comparison with *scanned* is tighter than in *MarkAndWarn*, because tracing is not concurrent with scanning.

```
$Trace(x) \equiv$
    $MarkBlack(x)$
    while $cache \neq \emptyset$ do pick $y \in cache$ do
        foreach $i \in INDEX$ do $MarkBlack(heap[y, i])$
$MarkBlack(x) \equiv$
    if $color[x] \neq Black$ then
        if $|cache| < MaxCache$ then
            $color[x] \leftarrow Black$
            $cache \leftarrow cache \oplus \{x\}$
        else
            $color[x] \leftarrow Gray$
            if $x < scanned$ then $dirty \leftarrow$ true
```

Because handshakes are completely asynchronous for the mutators, they require some waiting on the part of collector. Apart from access to the free list, handshakes are the only synchronization overhead on the collector.

```
$Handshake(s) \equiv$
    $status_C \leftarrow s$
    foreach $m \in PID$ do
        await $status_m \neq phase$ do skip
    $phase \leftarrow s$
```

The collector cycle (Figure 7) is a straightforward implementation of the diagrams of Figure 6. Note that both *scanned* and *swept* always point at the object under scrutiny or just before it.

```
Clear:  $Handshake(Sync_1)$
Mark:   $swept \leftarrow -\infty$
        cobegin
            $Handshake(Sync_2)$
            $Handshake(Async)$
        and
            foreach $x \in Globals$ do $Trace(x)$
Scan:   repeat
            $dirty \leftarrow$ false
            $scanned \leftarrow 0$
            while $scanned < End$ do
                if $color[scanned] = Grey$ then
                    $Trace(scanned)$
                $scanned \leftarrow scanned + 1$
            $scanned \leftarrow -\infty$
        until not $dirty$
Sweep:  $swept \leftarrow 0$
        while $swept < End$ do
            if $color[swept] \in \{Black, Gray\}$ then
                $color[swept] \leftarrow White$
            else if $color[swept] = White$ then
                $color[x] \leftarrow Blue$
                await true do $free \leftarrow free \oplus \{x\}$
            $swept \leftarrow swept + 1$
        $swept \leftarrow +\infty$
```

Figure 7: **The collector cycle**

5 Extensions

In this section, we describe how our algorithm can be extended to deal with more realistic heap and process management. These extensions are absolutely needed for a useful implementation, and they interfere in non-trivial ways with correctness proofs. The model in the appendix and the proof cover all extensions discussed here.

5.1 Process management

Because the collector must wait on all threads to complete a handshake, managing process creation and termination mainly poses liveness problems. We must make sure that the collector only has to wait on a finite number of processes to complete a cycle, by imposing that mutators call *Cooperate* before they launch a new process for the first time and always give their own status to new processes: processes with the "wrong" status will not beget offspring until they answer the collector.

We must also ensure that the collector only tests a finite number of processes. We can do this by maintaining a list of active processes. The contention for the list can be resolved by using double indirection (handles) for each link, each process inserting its recent offspring after itself on each *status* change, and letting the collector remove dead processes.

5.2 Heap management

Realistic heap management involves dealing with variable-sized objects, system allocation, and fragmentation.

We stick to the traditional implementation of variable-sized object, a header word containing the object size and color followed by the pointer fields, in order not to interfere with debugging. This fixes the direction of scans and sweeps—bottom-up—but does not otherwise affect the algorithm.

There is a mild clash between this header convention and the system allocation conventions, which usually grow memory from the top up. This can be solved by setting a top *limit* at the beginning of the *Scan* step. Since $swept = -\infty$ at that time, only floating garbage can be **create**d above *limit*. The *Clear* step must then start by unmarking all objects created above *limit* during the previous *Scan* and *Sweep* steps.

Block splitting (using only part of a free memory block to **create** an object) is delicate because it interferes with the comparisons with *swept*; the correct solution is to **create** the object at the top of the block, and to do the equality test with the block pointer rather than the object pointer.

Finally, the collector should be able to merge adjacent free blocks. In a sequential system this is done by rebuilding the free list during the *Sweep* step. Doing this in a concurrent system creates contention with the mutators, which can be reduced by letting the collector reclaim large segments of the free list for rebuilding. Since those segments must be white, we have the option of keeping the free list in white, and marking the free objects blue when they are reserved by a mutator.

6 The proof

A "proof" of a garbage collection algorithm is never a proof of an actual implementation of that algorithm; it is a proof of some mathematical model that conveys the essential ideas of the algorithm. More often this model is chosen in order to make the proof as short and elegant as possible [9, 4, 7], so it is very high-level and abstract. This yields elegant papers, but also carries a price: it is not clear how to fit the vast amount of details of an actual implementation in the small, cleverly crafted invariants of the published proof. This is somewhat unsettling for an asynchronous shared-memory garbage collector, where intricate synchronization problems invariably creep in the implementation of high-level concepts.

We purport to provide a proof that does provide for all the details of an actual implementation, including all the extensions discussed above, but that remains abstract enough to be tractable. The key step here is the choice of the model. By giving dataflow description of the algorithm we make the communication pattern between the various internal states explicit. Writing down the safety invariants—the most critical part of a garbage collection algorithm proof—then becomes a simple, if tedious, matter of combining and relating the values of the various variables. During the course of this work most of the problems with the algorithm were identified during the construction of the model; only a few more appeared during the safety proof. The liveness proof is straightforward.

6.1 The model

The mathematical model on which this proof is based is listed in Appendix A. The formalism we chose is a cross between **algol**, UNITY [7], and TLA [16]. Superficially our format resembles most UNITY: a set of concurrent atomic assignments, with only weak fairness constraints. The code of the individual atomic assignments uses an **algol**-like syntax.

(1) $step \in STEP \wedge phase \in \{Async, Sync_1, Sync_2\}$

(2) $status_C = phase = Async$
$\vee\ (status_C = Async \wedge phase = Sync_2 \wedge step = Mark)$
$\vee\ (status_C = Sync_1 \wedge phase \neq Sync_2 \wedge step = Clear)$
$\vee\ (status_C = Sync_2 \wedge phase \neq Async \wedge step = Mark)$

(3) $\forall m, status_m \in \{status_C, phase, Dead, Free, Quick\}$

(4) $\forall m, answering_m \ \Rightarrow\ status_m = phase \neq status_C$

(5) $\forall m, marking_m \ \Rightarrow\ status_m = Sync_2 \neq status_C$

(6) $\forall m, status_m \in \{Dead, Free, Quick\} \ \Rightarrow\ pc_m = Halt$

(7) $\{p \mid status_p = Quick\} = \bigoplus_m \{child_m \mid pc_m = Launch\}$

Figure 8: **Handshake invariants**

Mathematically, however, our formalism is really sugared TLA, because this offers the best approach to proving independently that implementations of the collector or processes match the model. From TLA we inherit the local variables of subprocesses and the selective use of fairness constraints—only statements containing a $\overset{\mathcal{WF}}{\Longrightarrow}$ are subject to a weak fairness constraint.[1]

We express the algorithm in a dataflow rather than an imperative style, in keeping with the TLA view that the easiest thing to abstract away from in a program is the programming language syntax. Instead of having a single imperative variable x whose exact meaning at any given time depends on a *pc* variable that could take several dozen values, we use a handful of dataflow variables, each of which holds the set of values of x at a given processing state. We can use a single set to factor away common processing, much like we use procedures in an imperative setting: for example the *mark* variable of the mutator model corresponds to the *MarkGray* procedure.

This dataflow style allows us to considerably reduce the number of *pc* values. For example the collector is almost completely parallelized; only the four steps remain. Besides the obvious gain in compactness, this also makes our model more general, indicating how the collector could be parallelized. In fact we have attempted to make the model as general as possible, e.g., we use a *rover* pointer to allow the collector to start tracing any *Gray* object at any time.

We take a rather high-level view of the free list mangement. Operations on the free list are viewed simply as atomic multiset operations; their implementation on terms of semaphores and linked list operation is standard and does not interfere with the rest of the algorithm. We model the free list with two sets, *free* and *alloc*, of *White* and *Blue* objects, respectively. The "memory" action that transfers objects from *free* to *alloc* can be assigned either to the collector, for a *Blue* free list, or to the mutators, for a *White* list, as hinted in section 5.

We are even more cavalier with process management. The process list is implicit; two extra values, *Free* and *Quick*, indicate processes *not* on the list. While the implementation outlined in section 5 is a little tricky, it does not interact with the rest of the algorithm, so there is little point in introducing more detail.

(8) $whiten \oplus claim \subseteq [0, ptr) \cap [0, swept]$

(9) $step = Sweep \wedge ptr < limit \ \Rightarrow\ swept \leq ptr$

(10) $step \neq Sweep \ \Rightarrow\ swept \in \{-\infty, +\infty\}$

(11) $step = Clear \wedge phase = Async \ \Rightarrow swept = +\infty$

(12) $status_C \neq Async \ \Rightarrow\ ptr = limit \wedge whiten = \emptyset$

(13) $step \in \{Mark, Scan\} \ \Rightarrow\ whiten = claim = \emptyset$

(14) $step = Mark \wedge status_C = Async \ \Rightarrow swept = -\infty$

(15) $step = Scan \ \Rightarrow swept = -\infty$

(16) $step = Scan \ \Rightarrow\ scanned \leq ptr \wedge scanned < limit$

(17) $step = Scan \wedge reset \wedge scanned = -\infty \ \Rightarrow\ ptr = limit$

(18) $step \neq Scan \ \Rightarrow\ reset$

(19) $step \in \{Sweep, Clear\} \ \Rightarrow\ blacken = trace = \emptyset$

(20) $step \in \{Sweep, Clear\} \ \Rightarrow\ cache = fields = \emptyset$

(21) $step \in \{Sweep, Clear\} \ \Rightarrow\ rover = 0$

Figure 9: **Collector invariants**

On the other hand, we have a detailed model of the heap layout, because the fragmentation procedures interfere quite subtly with the *Sweep* step. Each *heap* location contains either a pointer or an object header containing its *size* and *color*; objects start with a header and are adjacent in the *heap*.

All actions in our model, except free list and process management actions, are asynchronous in the sense that they make at most one read or write on a global variable or heap location, if one discounts reads of variables that are read-only for all other processes, such as *swept* for the collector, $process_m$ for a running mutator m, or the *size* of a non-garbage object.

Compliance with the latter constraint has introduced a few bumps and twists in the model. For example the collector uses an explicit register *ptr* to sweep the memory. However it is straightforward to show that the resulting behavior still conforms to the timing diagrams of Figure 6. Global invariants (1–7) (Figure 8) show that the handshakes go through, and invariants (8–21) (Figure 9), which are local to the collector, fill in the rest of the picture. Three notation details: m always ranges over *PID*, local mutator variables are subscripted (e.g., $marking_m$), and sets are considered a special case of multisets, so (7) asserts that $\{\{child_m\} \mid pc_m = Launch\}$ is a partition of the set of *Quick* processes.

6.2 Safety

Safety for a garbage collector generally reduces to "the collector does not free reachable objects". Here, because of our more detailed model, we must also show that "the object layout is not disrupted by the mutators or the collector". The first step towards this is to capture the "layout" and "reachable" concepts precisely, which we do in Figure 10.

Z, Y, and X are the "end", "field", and "field value" re-

[1] Free variables in an action are implicitly quantified existentially in the action, so for example the collector action with precondition $x \in cache$ must eventually be performed if *cache* is not empty infinitely often.

$$
\begin{aligned}
xZy &\triangleq x \in \mathit{ADDR} \wedge \mathit{heap}[x] \in \mathit{HEADER} \\
&\quad \wedge y = \mathit{heap}[x].\mathit{size} + x + 1 \\
xYy &\triangleq \exists z, xZz \wedge x < y < z \\
xXy &\triangleq \exists z, xYz \wedge y = \mathit{heap}[z] \\
O &\triangleq Z^*(0) \cap [0, \mathit{end}) \\
W &\triangleq \{x \in O \mid \mathit{heap}[x].\mathit{color} = \mathit{White}\} \\
G &\triangleq \{x \in O \mid \mathit{heap}[x].\mathit{color} = \mathit{Gray}\} \\
B &\triangleq \{x \in O \mid \mathit{heap}[x].\mathit{color} = \mathit{Black}\} \\
C_m &\triangleq \{\mathit{new}_m \mid \mathit{pc}_m \in \mathit{CREATE} \setminus \{\mathit{Split}\}\} \\
N &\triangleq \{\mathit{new}_m \mid \exists m, \mathit{pc}_m \in \mathit{CREATE} \setminus \{\mathit{Split}, \mathit{Fill}\}\} \\
S_m &\triangleq \{\mathit{old}_m \mid \mathit{pc}_m = \mathit{Split}\} \\
F_m &\triangleq \{\mathit{heap}[y] \mid \mathit{pc}_m \in \mathit{CREATE} \wedge \mathit{new}_m\ Y\ y \notin \mathit{fill}_m\} \\
A_m &\triangleq \widehat{\mathit{roots}}_m \cup \widehat{\mathit{mark}}_m \cup F_m \\
&\quad \cup \{x \mid (x = \mathit{new}_m \vee x = \mathit{old}_m \vee xY\mathit{field}_m) \\
&\qquad\quad \wedge \mathit{pc}_m \in \mathit{UPDATE}\} \\
&\quad \cup \{x \in \mathit{args}_m \mid \mathit{status}_m \neq \mathit{Quick}\} \\
&\quad \cup \{x \in \mathit{args}_p \mid \mathit{pc}_m = \mathit{Launch} \wedge p = \mathit{child}_m\} \\
U &\triangleq X^*\left((G \cup B) \setminus \bigcup\nolimits_m C_m\right) \cup \bigcup\nolimits_m X^*(A_m) \\
V &\triangleq (W \cup G \cup B) \setminus \widehat{\mathit{free}} \\
J &\triangleq V \setminus \bigcup\nolimits_m (X^*(A_m) \cup C_m) \\
K &\triangleq \{\mathit{field}_m \mid \exists m, \mathit{pc}_m {\in} \mathit{UPDATE} \wedge \mathit{status}_m \neq \mathit{Sync}_1 \\
&\qquad\qquad \wedge (\mathit{pc}_m = \mathit{Store} \vee \mathit{old}_m \neq \mathit{heap}[\mathit{field}_m])\} \\
xTy &\triangleq \exists z \notin K, xYz \wedge y = \mathit{heap}[z] \notin B \\
R_C &\triangleq \{x \in G \mid \mathit{step} \neq \mathit{Scan} \vee x \geq \mathit{ptr} \vee \mathit{reset} \vee \mathit{dirty}\} \\
&\quad \cup \mathit{blacken} \cup \widehat{\mathit{cache}} \cup (\mathit{heap}[\widehat{\mathit{fields}} \setminus K] \cup \mathit{trace}) \setminus B \\
R_m &\triangleq (\widehat{\mathit{mark}}_m \cup \{x \in F_m \mid \mathit{marking}_m\}) \setminus B \\
M &\triangleq T^*\left(R_C \cup \bigcup\nolimits_m R_m\right) \cup (G \cup B) \setminus \bigcup\nolimits_m C_m
\end{aligned}
$$

Figure 10: **Auxiliary definitions**

(22) $U \oplus \bigoplus_m C_m \subseteq V \setminus \widehat{\mathit{claim}}$

(23) $\mathit{free} \oplus \mathit{claim} \subseteq W$

(24) $\mathit{alloc} \oplus \bigoplus_m (\mathit{pool}_m \oplus S_m) = O \setminus (W \cup G \cup B)$

(25) $\forall m, \mathit{fill}_m \subseteq \{x \in Y(\mathit{new}_m) \mid \mathit{pc}_m \in \mathit{CREATE}\}$

(26) $\forall m, \mathit{pc}_m \in \mathit{UPDATE} \;\Rightarrow\; \mathit{field}_m \in Y(O)$

(27) $\mathit{whiten} \subseteq V \wedge \widehat{\mathit{fields}} \subseteq Y(O) \wedge R_C \subseteq U \cup G$

(28) $\forall m, \mathit{status}_m \neq \mathit{Async} \vee \mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\}$
$\quad\Rightarrow\; C_m \subseteq W \cup B \wedge \mathit{pc}_m \neq \mathit{GrayNew}$

(29) $\forall m, \mathit{pc}_m = \mathit{Split} \;\Rightarrow\; \mathit{heap}[\mathit{new}_m].\mathit{color} = \mathit{Black}$

(30) $\{\mathit{ptr}, \mathit{limit}, \mathit{rover}\} \subseteq O \cup \{\mathit{end}\}$

(31) $\mathit{ptr} \leq \mathit{limit} \wedge \mathit{sublimit} \leq \mathit{limit}$

(32) $\mathit{end} \in Z^*(0)$

(33) $\forall m, \mathit{pc}_m = \mathit{Split} \;\Rightarrow\; \mathit{old}_m\ Y\ \mathit{new}_m$

(34) $\forall m, \mathit{pc}_m = \mathit{Split} \;\Rightarrow\; Z(\mathit{old}_m) = Z(\mathit{new}_m)$

(35) $\mathit{step} = \mathit{Sweep} \;\Rightarrow\; \left(U \cup \bigcup_m C_m\right) \cap W \subseteq [0, \mathit{ptr})$

(36) $\forall m, \mathit{pc}_m = \mathit{ClearNew} \wedge \mathit{step} = \mathit{Sweep} \;\Rightarrow \mathit{new}_m < \mathit{ptr}$

(37) $\mathit{step} = \mathit{Sweep} \;\Rightarrow\; \widehat{\mathit{free}} \cap [\mathit{ptr}, \mathit{sublimit}) = \emptyset$

(38) $\mathit{step} = \mathit{Sweep} \;\Rightarrow\; B \subseteq \mathit{whiten} \cup N \cup [\mathit{ptr}, \mathit{end})$

(39) $\forall m, \mathit{pc}_m = \mathit{TestSweep} \wedge \mathit{step} = \mathit{Sweep} \wedge \mathit{old}_m < \mathit{swept}$
$\quad\Rightarrow\; \mathit{new}_m \notin [\mathit{swept}, \mathit{ptr}) \setminus (W \cup \mathit{whiten})$

(40) $\forall x \in O, \mathit{step} = \mathit{Sweep} \wedge \mathit{swept} < x < \mathit{ptr} \;\Rightarrow\; x \in V$

(41) $\mathit{step} = \mathit{Clear} \;\Rightarrow\; B \subseteq \mathit{whiten} \cup N \cup [\mathit{ptr}, \mathit{limit})$

(42) $\mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\} \;\Rightarrow\; \mathit{heap}[K] \subseteq M = X^*(M)$

(43) $\widehat{\mathit{cache}} \subseteq G \cup B \wedge \mathit{blacken} \subseteq W \cup G$

(44) $\forall m, \mathit{pc}_m = \mathit{Store} \wedge \mathit{status}_m \neq \mathit{Async} \wedge \neg\mathit{marking}_m$
$\quad\Rightarrow\; \mathit{new}_m \in B \cup G \cup R_m$

(45) $\forall m, \mathit{pc}_m \in \{\mathit{TestScan}, \mathit{SetDirty}\} \wedge \mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\}$
$\quad\Rightarrow\; \mathit{old}_m \in G \cup B$

(46) $\forall m, \mathit{pc}_m \in \mathit{UPDATE} \setminus \{\mathit{Store}\} \;\Rightarrow\; \mathit{status}_m = \mathit{Async}$

(47) $\forall m, \mathit{status}_m {\in} \{\mathit{Dead}, \mathit{Free}, \mathit{Quick}\} \;\Rightarrow\; A_m = \mathit{pool}_m = \emptyset$

(48) $\forall m, \mathit{pc}_m {=} \mathit{Halt} \;\Rightarrow\; \mathit{args}_m = \emptyset$

(49) $\forall m, \mathit{marking}_m \vee \mathit{status}_m {=} \mathit{Async} \wedge \mathit{step} {\in} \{\mathit{Mark}, \mathit{Scan}\}$
$\quad\Rightarrow\; X^*(A_m) \subseteq M$

(50) $\forall m, \mathit{marking}_m \wedge \mathit{pc}_m = \mathit{Fill} \;\Rightarrow\; \mathit{new}_m \in W$

(51) $\forall m, \mathit{status}_m = \mathit{Async} \wedge \mathit{step} \in \{\mathit{Mark}, \mathit{Scan}\}$
$\quad\Rightarrow \mathit{mark}_m = \emptyset \wedge C_m \subseteq B \wedge \mathit{pc}_m \neq \mathit{ClearNew}$

(52) $\mathit{step} = \mathit{Scan} \;\Rightarrow\; R_C \cup U \setminus B \subseteq [0, \mathit{limit})$

(53) $\forall m, \mathit{pc}_m = \mathit{GrayOld} \wedge \mathit{step} = \mathit{Scan} \;\Rightarrow\; \mathit{old}_m < \mathit{limit}$

(54) $\forall x \in O \setminus B, \mathit{step} = \mathit{Scan} \wedge x < \mathit{ptr} < \mathit{limit}$
$\quad\Rightarrow\; x < \mathit{scanned}$

Figure 11: **Safety invariants**

lations, respectively, so X^* is the reachability relation. O is the set of objects, W, G, B its color subsets. A_m is the set of objects immediately accessible by mutator m. It includes not only $\widehat{\mathit{roots}_m}$ (the set underlying the multiset roots_m), but also objects that are being marked or updated, and filled fields F_m of an object being created. The latter (C_m) is not part of A_m. U is the set of objects under use; it includes all objects reachable from the "registers" A_m, or from shaded objects not under creation (which may be used by the collector). V is the set of valid objects, and $O \setminus V$ is the set of available memory blocks. J is the set of garbage objects: valid objects that are not reachable from any mutator.

The main safety invariant is (22) which implies that used objects only contain pointers to used objects, that they do not appear on the *free* or *claim* lists, and that there are no pointers to objects under creation (since the inclusion of a multiset union in a set implies that the union is disjoint). However all the invariants in Figure 4 depend on each other to some extent (except 46–47), and must be proved simultaneously.

(23–25) asserts that all free memory, blue or white, is well accounted for. (25–31) ensure that mutator and collector pointer variables have proper values; (28) ensures that the mutators do not create unfilled gray objects when the collector is tracing. (32) asserts that the object layout is consistent on all the used portion of the heap, and is needed to establish (30–31); (33–34) ensure that fragmentation preserves (32).

(35–37) ensure that only garbage is reclaimed by the sweep step. (38–41) ensure that the sweep and clear steps do not leave black objects behind, except newly created objects that will be cleared (N). (40) is the key property of the split-off-top policy: the sweep cannot leap over free memory block headers. This ensures (39), which in turn ensures that new objects are created with the right color.

The remaining invariants ensure that the mark and scan steps shade all objects, so that (35) is established at the beginning of the next sweep. These invariants are all based on the formula for M, the set of objects that would be traced by the collector if the mutators cleared all their registers. M contains all shaded objects, plus all objects reachable by the trace relation T from the trace roots R_C and R_m. The trace relation is like the reachability relation, except it ignores black fields and fields that are being updated and whose values are unreliable (K).

The main marking invariant is (49); it is the equivalent of the "reachable objects are reachable from a gray" invariant of the basic algorithm. In turn, (49) crucially depends on (42), which asserts that M is closed under reachability; it is the equivalent of the "no black points to a white" invariant of the basic algorithm.

Part of the proof of (49) is that M is non-decreasing after $\mathit{phase} = \mathit{Sync}_2$. The purpose of the **update** overhead is to ensure that the inevitable increase of K does not lead to a decrease of M, i.e., it ensures that old_m will always be traced. In addition, K also contains the fields for which old_m is the "wrong" value, and the overhead is misspent. The $\mathit{status} \neq \mathit{Async}$ overhead covers that case by ensuring that the value deposited in a field that remains in K because of a concurrent assignment is always traced.

Finally, note that the invariants remain valid if we add to K all the fields that have been updated since the start of the mark step. Once a field has been "cut off" from the collector, it remains so. This implies that *Async* processes need to incur the **update** overhead at most once per cycle per updated field.

6.3 Liveness

The liveness part of the proof is much more standard. We need to establish that "all garbage is eventually collected", i.e., $x \in J \leadsto x \notin V$. It is straightforward to show that all "quick" garbage in $J \cup W \cup [0, \mathit{limit})$ at the beginning of a sweep step is collected by that step, and that the rest of the garbage is whitened and thus becomes quick garbage during the sweep and clear steps, and remains quick garbage during the mark and scan steps. Therefore, we only need to show progress of the collector cycle, and this is trivial except for the handshakes and the scan step.

For the handshakes, first note that each active mutator must eventually change its status after the collector has changed his, either by doing an **exit**, or several **cooperate** actions. If m never does an **exit**, then $\Box[\mathit{pc}_m \neq \mathit{Halt}$ so m eventually sets $\mathit{answering}_m$. m can only reset $\mathit{answering}_m$ by completing the *cooperate*, which it must eventually do since all other actions are blocked. In addition, only the mutators with $\mathit{pc} \neq \mathit{Halt}$ at the start of the handshake can spawn processes with $\mathit{status} = \mathit{phase}$; once these mutators have responded the set of mutators with $\mathit{status} = \mathit{phase}$ decreases, hence the handshake completes.

Let us assume the scan step never terminates, $\Box[\mathit{step} = \mathit{Scan}]$. Note that no used white objects can be created during the scan step, so that $W \cap (U \cup M)$ is eventually constant. After all pending **update**s complete, no mutator will ever set the *color* of an object to *Gray*. It follows from this and (43) that $\mathit{cache} \oplus G$ must decrease, so the normal emptying of the cache must eventually stop. At this point the cache must be empty, since the overflow action cannot empty the cache. Therefore we must have $\mathit{cache} = \mathit{blacken} = \emptyset$ from this point on, and G is constant. Eventually we must also have $\Box[\mathit{fields} = \mathit{trace} = \emptyset]$.

If $\Box[\neg \mathit{reset}]$ at this point, then eventually $\Box[\mathit{ptr} = \mathit{limit}]$, so $\Diamond[\mathit{dirty}]$, otherwise the scan step would end, and by (17) $\Diamond[\mathit{reset}]$, a contradiction. So $\Diamond[\mathit{reset}]$, and thus $\Diamond[\mathit{scanned} = -\infty]$. From this point on we must have $G \cap [0, \mathit{scanned}] = \emptyset$, since *blacken* must remain empty. Thus once all pending **update**s have completed, no mutator can set *dirty*. As above, eventually we have $\mathit{reset} \wedge \mathit{scanned} = -\infty$, and by (18) $\mathit{ptr} = \mathit{limit}$, whence eventually $\neg\mathit{reset} \wedge \neg\mathit{dirty}$. This implies $\Box[\neg\mathit{dirty} \wedge \neg\mathit{reset}]$, which implies a contradiction by the above.

References

[1] Appel, A. W. *Compiling with continuations.* Cambridge University Press, 1992.

[2] Appel, A. W., Ellis, J. R., and Li, K. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Notices 23*, 7 (1988), 11–23.

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

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

[5] Boehm, H. J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. *SIGPLAN Notices 26*, 6 (1991), 157–164.

[6] Brooks, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Lisp and Functional Programming 1984* (1984), ACM Press, pp. 256–262.

[7] Chandy, K. M., and Misra, J. *Parallel Program Design.* Addison-Wesley, 1988.

[8] Cypress. *BiCMOS/CMOS data book.* Cypress Semiconductor, 1991.

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

[10] Doligez, D., and Leroy, X. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Principles of Programming Languages 1993* (1993), ACM Press, pp. 113–123.

[11] Herlihy, M., and Moss, J. E. B. Non-blocking garbage collection for multiprocessors. Technical report CRL 90/9, DEC Cambridge Research Lab., 1990.

[12] Hibino, Y. A practical garbage collection algorithm and its implementation. In *7th Annual International Symposium on Computer Architecture* (1980), ACM Press, pp. 113–120.

[13] Hickey, T., and Cohen, J. Performance analysis of on-the-fly garbage collection. *Commun. ACM 27*, 11 (1984), 1143–1154.

[14] Kung, H. T., and Song, S. W. An efficient parallel garbage collection system and its correctness proof. In *Foundations of Computer Science 1977* (1977), IEEE Computer Society Press, pp. 120–131.

[15] Lamport, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. IEEE Conf. Parallel Processing* (1976), pp. 50–54.

[16] Lamport, L. The temporal logic of actions. Research report 79, DEC Systems Research Center, 1991.

[17] Leroy, X., and Mauny, M. The Caml Light system, version 0.5 — documentation and user's guide. Technical report L-5, INRIA, 1992.

[18] Nettles, S., O'Toole, J., Pierce, D., and Haines, N. Replication-based incremental copying collection. In *International Workshop in Memory Management 1992* (1992), vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 357–364.

[19] Newman, I. A., Stallard, R. P., and Woodward, M. C. A hybrid multiple processor garbage collection algorithm. *The Computer Journal 30*, 2 (1987), 119–127.

[20] North, S. C., and Reppy, J. H. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture 1987* (1987), vol. 242 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 113–133.

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

A The full algorithm model

Global declarations

$$
\begin{array}{lll}
\textbf{type} & \mathit{ADDR} & \triangleq \mathit{NAT} \\
 & \mathit{SIZE} & \triangleq \mathit{NAT} \\
 & \mathit{COLOR} & \triangleq \{\mathit{Blue}, \mathit{White}, \mathit{Gray}, \mathit{Black}\} \\
 & \mathit{HEADER} & \triangleq \textbf{record} \begin{cases} \mathit{color} \in \mathit{COLOR} \\ \mathit{size} \;\; \in \mathit{SIZE} \end{cases} \\
 & \mathit{WORD} & \triangleq \mathit{ADDR} \uplus \mathit{HEADER} \\
 & \mathit{PID} & \\
 & \mathit{STATUS} & \triangleq \{\mathit{Async}, \mathit{Sync}_1, \mathit{Sync}_2, \mathit{Dead}, \mathit{Free}, \mathit{Quick}\}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{var} & \mathit{heap} & \in \textbf{array } [\mathit{ADDR}] \textbf{ of } \mathit{WORD} \\
 & \mathit{end} & \in \mathit{ADDR} \\
 & \mathit{dirty} & \in \mathit{BOOL} \\
 & \mathit{free}, \mathit{alloc} & \in \textbf{multiset of } \mathit{ADDR} \\
 & \mathit{status}_C & \in \mathit{STATUS} \\
 & \mathit{swept} & \in \mathit{ADDR} \uplus \{-\infty, +\infty\} \\
 & \mathit{scanned} & \in \mathit{ADDR} \uplus \{-\infty\}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall m \in \mathit{PID},} \\
\quad \mathit{status}_m & \in \mathit{STATUS} \\
\quad \mathit{args}_m & \in \textbf{multiset of } \mathit{ADDR}
\end{array}
$$

Global initialization

$$
\begin{array}{ll}
\textbf{init} & \mathit{end} = 0 \\
 & \mathit{free} = \mathit{alloc} = \emptyset \\
 & \mathit{status}_C = \mathit{Async} \\
 & \forall m \in \mathit{PID}, \mathit{status}_m \in \{\mathit{Free}, \mathit{Async}\} \\
 & \{m \in \mathit{PID} \mid \mathit{status}_m \neq \mathit{Free}\} \text{ is finite} \\
 & \forall m \in \mathit{PID}, \mathit{args}_m = \emptyset
\end{array}
$$

Memory

```
$\langle\ s \in \mathit{SIZE}$
    $\Longrightarrow \mathit{heap}[\mathit{end}] \leftarrow \textbf{record} \begin{cases} \mathit{color} \mapsto \mathit{Blue} \\ \mathit{size} \mapsto s \end{cases}$
        $\mathit{end} \leftarrow \mathit{end} + s + 1$
        $\mathit{alloc} \leftarrow \mathit{alloc} \oplus \{\mathit{end}\}\ \rangle$
$\langle\ x \in \mathit{free}$
    $\Longrightarrow \mathit{heap}[x].\mathit{color} \leftarrow \mathit{Blue}$
        $\mathit{free} \leftarrow \mathit{free} \ominus \{x\}$
        $\mathit{alloc} \leftarrow \mathit{alloc} \oplus \{x\}\ \rangle$
$\langle\ x \in \mathit{free}$
    $\Longrightarrow \mathit{free} \leftarrow \mathit{free} \ominus \{x\}\ \rangle$
$\langle\ x \in \mathit{alloc}$
    $\Longrightarrow \mathit{alloc} \leftarrow \mathit{alloc} \ominus \{x\}\ \rangle$
        $\mathit{heap}[x].\mathit{color} \leftarrow \mathit{White}$
```

```
Mutator $m$

type $\mathit{CREATE} \triangleq \{\mathit{Split}, \mathit{TestSweep}, \mathit{ClearNew}, \mathit{GrayNew}, \mathit{Fill}\}$
     $\mathit{UPDATE} \triangleq \{\mathit{TestOld}, \mathit{GrayOld}, \mathit{TestScan}, \mathit{SetDirty}, \mathit{Store}\}$
     $\mathit{LABEL} \triangleq \mathit{CREATE} \uplus \mathit{UPDATE} \uplus \{\mathit{Halt}, \mathit{Work}, \mathit{Launch}\}$
var $\mathit{pc} = \mathit{Halt} \in \mathit{LABEL}$
    $\mathit{roots} = \mathit{pool} = \mathit{mark} = \mathit{fill} = \emptyset \in \textbf{multiset of } \mathit{ADDR}$
    $\mathit{answering} = \mathit{marking} = \textbf{false} \in \mathit{BOOL}$
    $\mathit{child} \in \mathit{PID}$
    $\mathit{old}, \mathit{new}, \mathit{field} \in \mathit{ADDR}$
startup
$\langle\ \mathit{pc} = \mathit{Halt}\ \wedge\ \mathit{status}_m \in \{\mathit{Async}, \mathit{Sync}_1, \mathit{Sync}_2\}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{roots} \leftarrow \mathit{args}_m$
        $\mathit{args}_m \leftarrow \emptyset$
        if $\mathit{status}_m \neq \mathit{status}_C$ then $\mathit{answering} \leftarrow \textbf{true}$
        $\mathit{pc} \leftarrow \mathit{Work}\ \rangle$
launch
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ \neg\mathit{answering}\ \wedge\ p \in \mathit{PID}\ \wedge\ \mathit{status}_p = \mathit{Free}$
    $\Longrightarrow\ \mathit{child} \leftarrow p$
        $\mathit{status}_p \leftarrow \mathit{Quick}$
        $\mathit{pc} \leftarrow \mathit{Launch}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Launch}\ \wedge\ x \in \mathit{roots}\ \wedge\ p = \mathit{child}$
    $\Longrightarrow\ \mathit{args}_p \leftarrow \mathit{args}_p \oplus \{x\}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Launch}\ \wedge\ p = \mathit{child}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{status}_p \leftarrow \begin{cases} \mathit{Async} & \textbf{if } \mathit{marking} \\ \mathit{status}_m & \textbf{otherwise} \end{cases}$
        $\mathit{pc} \leftarrow \mathit{Work}\ \rangle$
exit
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ \mathit{mark} = \mathit{pool} = \emptyset$
    $\Longrightarrow\ \mathit{status}_m \leftarrow \mathit{Dead}$
        $\mathit{answering} \leftarrow \mathit{marking} \leftarrow \textbf{false}$
        $\mathit{roots} \leftarrow \emptyset$
        $\mathit{pc} \leftarrow \mathit{Halt}\ \rangle$
cooperate
$\langle\ \mathit{pc} \neq \mathit{Halt}\ \wedge\ \mathit{status}_m \neq \mathit{status}_C$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{answering} \leftarrow \textbf{true}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ \mathit{answering}\ \wedge\ \neg\mathit{marking}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{answering} \leftarrow \textbf{false}$
        if $\mathit{status}_m = \mathit{Sync}_2$ then
            $\mathit{mark} \leftarrow \mathit{mark} \oplus \mathit{roots}$
            $\mathit{marking} \leftarrow \textbf{true}$
        $\mathit{status}_m \leftarrow \begin{cases} \mathit{Sync}_1 & \textbf{if } \mathit{status}_m = \mathit{Async} \\ \mathit{Sync}_2 & \textbf{otherwise} \end{cases}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ \mathit{marking}\ \wedge\ \mathit{mark} = \emptyset$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{answering} \leftarrow \mathit{marking} \leftarrow \textbf{false}$
        $\mathit{status}_m \leftarrow \mathit{Async}\ \rangle$
mark
$\langle\ x \in \mathit{mark}\ \wedge\ \mathit{heap}[x].\mathit{color} \neq \mathit{White}$
    $\Longrightarrow\ \mathit{mark} \leftarrow \mathit{mark} \ominus \{x\}\ \rangle$
$\langle\ x \in \mathit{mark}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[x].\mathit{color} \leftarrow \mathit{Gray}$
        $\mathit{mark} \leftarrow \mathit{mark} \ominus \{x\}\ \rangle$
move
$\langle\ x \in \mathit{roots}$
    $\Longrightarrow\ \mathit{roots} \leftarrow \mathit{roots} \oplus \{x\}\ \rangle$
$\langle\ x \in \mathit{roots}$
    $\Longrightarrow\ \mathit{roots} \leftarrow \mathit{roots} \ominus \{x\}\ \rangle$
load
$\langle\ x \in \mathit{roots}\ \wedge\ x < z \leq x + \mathit{heap}[x].\mathit{size}$
    $\Longrightarrow\ \mathit{roots} \leftarrow \mathit{roots} \oplus \{\mathit{heap}[z]\}\ \rangle$
reserve
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ x \in \mathit{alloc}$
    $\Longrightarrow\ \mathit{alloc} \leftarrow \mathit{alloc} \ominus \{x\}$
        $\mathit{pool} \leftarrow \mathit{pool} \oplus \{x\}\ \rangle$
$\langle\ x \in \mathit{pool}$
    $\Longrightarrow\ \mathit{pool} \leftarrow \mathit{pool} \ominus \{x\}$
        $\mathit{alloc} \leftarrow \mathit{alloc} \oplus \{x\}\ \rangle$
create
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ \neg\mathit{answering}\ \wedge\ s \in \mathit{SIZE}$
  $\wedge\ x \in \mathit{pool}\ \wedge\ \mathit{heap}[x].\mathit{size} >= s$
    $\Longrightarrow\ \mathit{pool} \leftarrow \mathit{pool} \ominus \{x\}$
        $\mathit{old} \leftarrow x$
        $\mathit{new} \leftarrow x + \mathit{heap}[x].\mathit{size} - s$
        $\mathit{fill} \leftarrow \{\mathit{new} + 1, \ldots, \mathit{new} + s\}$
        $\mathit{heap}[\mathit{new}] \leftarrow \textbf{record} \begin{cases} \mathit{color} \mapsto \mathit{Black} \\ \mathit{size} \mapsto s \end{cases}$
        $\mathit{pc} \leftarrow \begin{cases} \mathit{Split} & \textbf{if } \mathit{new} > \mathit{old} \\ \mathit{TestSweep} & \textbf{otherwise} \end{cases}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Split}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[\mathit{old}].\mathit{size} \leftarrow \mathit{new} - 1 - \mathit{old}$
        $\mathit{pool} \leftarrow \mathit{pool} \oplus \{\mathit{old}\}$
        $\mathit{pc} \leftarrow \mathit{TestSweep}\ \rangle$
$\langle\ \mathit{pc} = \mathit{TestSweep}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{pc} \leftarrow \begin{cases} \mathit{ClearNew} & \textbf{if } \mathit{status}_m \neq \mathit{Async} \\ & \quad \vee\ \mathit{swept} > \mathit{new} \\ \mathit{GrayNew} & \textbf{if } \mathit{old} \leq \mathit{swept} \leq \mathit{new} \\ \mathit{Fill} & \textbf{otherwise} \end{cases}\ \rangle$
$\langle\ \mathit{pc} = \mathit{ClearNew}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[\mathit{new}].\mathit{color} \leftarrow \mathit{White}$
        $\mathit{pc} \leftarrow \mathit{Fill}\ \rangle$
$\langle\ \mathit{pc} = \mathit{GrayNew}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[\mathit{new}].\mathit{color} \leftarrow \mathit{Gray}$
        $\mathit{pc} \leftarrow \mathit{Fill}\ \rangle$
fill
$\langle\ y \in \mathit{roots}\ \wedge\ z \in \mathit{fill}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[z] \leftarrow y$
        if $\mathit{marking}$ then $\mathit{mark} \leftarrow \mathit{mark} \ominus \{y\}$
        $\mathit{fill} \leftarrow \mathit{fill} \ominus \{z\}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Fill}\ \wedge\ \mathit{fill} = \emptyset$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{roots} \leftarrow \mathit{roots} \oplus \{\mathit{new}\}$
        if $\mathit{marking}$ then $\mathit{mark} \leftarrow \mathit{mark} \oplus \{\mathit{new}\}$
        $\mathit{pc} \leftarrow \mathit{Work}\ \rangle$
update
$\langle\ \mathit{pc} = \mathit{Work}\ \wedge\ \neg\mathit{answering}$
  $\wedge\ x, y \in \mathit{roots}\ \wedge\ x < z \leq x + \mathit{heap}[x].\mathit{size}$
    $\Longrightarrow\ \mathit{new} \leftarrow y$
        $\mathit{field} \leftarrow z$
        $\mathit{old} \leftarrow \mathit{heap}[z]$
        if $\mathit{status}_m \neq \mathit{Async} \wedge \neg\mathit{marking}$ then
            $\mathit{mark} \leftarrow \mathit{mark} \oplus \{\mathit{new}\}$
        if $\mathit{status}_m = \mathit{Sync}_2$ then
            $\mathit{mark} \leftarrow \mathit{mark} \oplus \{\mathit{old}\}$
        $\mathit{pc} \leftarrow \begin{cases} \mathit{TestOld} & \textbf{if } \mathit{status}_m = \mathit{Async} \\ \mathit{Store} & \textbf{otherwise} \end{cases}\ \rangle$
$\langle\ \mathit{pc} \in \mathit{UPDATE}\ \wedge\ \mathit{swept} > -\infty$
    $\Longrightarrow\ \mathit{pc} \leftarrow \mathit{Store}\ \rangle$
$\langle\ \mathit{pc} = \mathit{TestOld}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{pc} \leftarrow \begin{cases} \mathit{GrayOld} & \textbf{if } \mathit{heap}[\mathit{old}].\mathit{color} = \mathit{White} \\ \mathit{TestScan} & \textbf{if } \mathit{heap}[\mathit{old}].\mathit{color} = \mathit{Gray} \\ \mathit{Store} & \textbf{otherwise} \end{cases}\ \rangle$
$\langle\ \mathit{pc} = \mathit{GrayOld}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[\mathit{old}].\mathit{color} \leftarrow \mathit{Gray}$
        $\mathit{pc} \leftarrow \mathit{TestScan}\ \rangle$
$\langle\ \mathit{pc} = \mathit{TestScan}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{pc} \leftarrow \begin{cases} \mathit{SetDirty} & \textbf{if } \mathit{scanned} \geq \mathit{old} \\ \mathit{Store} & \textbf{otherwise} \end{cases}\ \rangle$
$\langle\ \mathit{pc} = \mathit{SetDirty}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{dirty} \leftarrow \textbf{true}$
        $\mathit{pc} \leftarrow \mathit{Store}\ \rangle$
$\langle\ \mathit{pc} = \mathit{Store}$
    $\overset{\mathcal{WF}}{\Longrightarrow}\ \mathit{heap}[\mathit{field}] \leftarrow \mathit{new}$
        $\mathit{pc} \leftarrow \mathit{Work}\ \rangle$
```

Collector

```
type $STEP \triangleq \{Sweep, Clear, Mark, Scan\}$
var $step = Sweep \in STEP$
    $phase = Async \in STATUS$
    $ptr = limit = sublimit = rover = 0 \in ADDR$
    $reset = \mathbf{true} \in BOOL$
    $whiten = blacken = trace = \emptyset \in \mathbf{set\ of}\ ADDR$
    $claim = cache = fields = \emptyset \in \mathbf{multiset\ of}\ ADDR$
```

sweep

```
$\langle\ step = Sweep\ \wedge\ swept = ptr < sublimit$
    $\overset{\mathcal{WF}}{\Longrightarrow}$ if $heap[ptr].color \in \{Gray, Black\}$ then
            $whiten \leftarrow whiten \cup \{ptr\}$
        else if $heap[ptr].color = White$ then
            $claim \leftarrow claim \oplus \{ptr\}$
        $ptr \leftarrow ptr + heap[ptr].size + 1\ \rangle$
$\langle\ step = Sweep\ \wedge\ swept < ptr < sublimit$
    $\overset{\mathcal{WF}}{\Longrightarrow} swept \leftarrow ptr\ \rangle$
$\langle\ step = Sweep\ \wedge\ sublimit \leq ptr < x \leq limit$
    $\overset{\mathcal{WF}}{\Longrightarrow} free \leftarrow free \ominus \{ptr, \ldots, x-1\}$
        $sublimit \leftarrow x\ \rangle$
$\langle\ step = Sweep\ \wedge\ ptr = limit$
    $\overset{\mathcal{WF}}{\Longrightarrow} swept \leftarrow +\infty\ \rangle$
$\langle\ step = Sweep\ \wedge\ swept = +\infty$
    $\overset{\mathcal{WF}}{\Longrightarrow} limit \leftarrow end$
        $step \leftarrow Clear\ \rangle$
```

clear

```
$\langle\ step = Clear\ \wedge\ ptr < limit$
    $\overset{\mathcal{WF}}{\Longrightarrow}$ if $heap[ptr].color \in \{Gray, Black\}$ then
            $whiten \leftarrow whiten \cup \{ptr\}$
        $ptr \leftarrow ptr + heap[ptr].size + 1\ \rangle$
$\langle\ x \in whiten$
    $\overset{\mathcal{WF}}{\Longrightarrow} whiten \leftarrow whiten \backslash \{x\}$
        $heap[x].color \leftarrow White\ \rangle$
$\langle\ step = Clear\ \wedge\ ptr = limit\ \wedge\ whiten = \emptyset$
    $\overset{\mathcal{WF}}{\Longrightarrow} status_C \leftarrow Sync_1\ \rangle$
$\langle\ status_C = phase = Sync_1\ \wedge\ claim = \emptyset$
    $\overset{\mathcal{WF}}{\Longrightarrow} status_C \leftarrow Sync_2$
        $step \leftarrow Mark\ \rangle$
```

claim

```
$\langle\ x \in claim\ \wedge\ y = x + heap[x].size + 1 \in claim$
    $\Longrightarrow claim \leftarrow claim \ominus \{y\}$
        $heap[x].size \leftarrow size + heap[y].size + 1\ \rangle$
$\langle\ x \in claim$
    $\overset{\mathcal{WF}}{\Longrightarrow} claim \leftarrow claim \ominus \{x\}$
        $free \leftarrow free \oplus \{x\}\ \rangle$
```

handshake

```
$\langle\ status_C \neq phase\ \wedge\ \forall m \in PID, status_m \neq phase$
    $\overset{\mathcal{WF}}{\Longrightarrow} phase \leftarrow status_C\ \rangle$
$\langle\ status_m = Dead$
    $\overset{\mathcal{WF}}{\Longrightarrow} status_m \leftarrow Free\ \rangle$
```

globals

```
$\langle\ step \in \{Mark, Scan\}$
    $\Longrightarrow rover \leftarrow 0\ \rangle$
$\langle\ step \in \{Mark, Scan\}\ \wedge\ rover < end$
    $\Longrightarrow rover \leftarrow rover + heap[rover].size + 1\ \rangle$
$\langle\ step \in \{Mark, Scan\}\ \wedge\ rover < end$
  $\wedge\ heap[rover].color = Gray$
    $\Longrightarrow blacken \leftarrow blacken \cup \{rover\}\ \rangle$
```

trace

```
$\langle\ x \in blacken$
    $\overset{\mathcal{WF}}{\Longrightarrow} heap[x].color \leftarrow Black$
        $blacken \leftarrow blacken \backslash \{x\}$
        $cache \leftarrow cache \oplus \{x\}\ \rangle$
$\langle\ x \in cache\ \wedge\ cache \neq \{x\}$
    $\Longrightarrow cache \leftarrow cache \ominus \{x\}$
        $heap[x].color \leftarrow Gray$
        if $x < ptr$ then $reset \leftarrow \mathbf{true}\ \rangle$
$\langle\ x \in cache$
    $\overset{\mathcal{WF}}{\Longrightarrow} cache \leftarrow cache \ominus \{x\}$
        $fields \leftarrow fields \oplus \{x+1, \ldots, x + heap[x].size\}$
$\langle\ x \in fields$
    $\overset{\mathcal{WF}}{\Longrightarrow} fields \leftarrow fields \ominus \{x\}$
        $trace \leftarrow trace \cup \{heap[x]\}\ \rangle$
$\langle\ x \in trace$
    $\overset{\mathcal{WF}}{\Longrightarrow} trace \leftarrow trace \backslash \{x\}$
        if $heap[x].color \in \{White, Gray\}$ then
            $blacken \leftarrow blacken \cup \{x\}\ \rangle$
```

mark

```
$\langle\ phase \neq Async$
    $\overset{\mathcal{WF}}{\Longrightarrow} swept \leftarrow -\infty\ \rangle$
$\langle\ status_C = phase = Sync_2\ \wedge\ swept = -\infty$
    $\overset{\mathcal{WF}}{\Longrightarrow} status_C \leftarrow Async\ \rangle$
$\langle\ step = Mark\ \wedge\ phase = Async\ \wedge\ scanned = -\infty$
    $\overset{\mathcal{WF}}{\Longrightarrow} ptr \leftarrow limit \leftarrow end$
        $step \leftarrow Scan\ \rangle$
```

reset

```
$\langle\ reset$
    $\overset{\mathcal{WF}}{\Longrightarrow}$ if $step = Scan$ then $ptr \leftarrow limit$
        $scanned \leftarrow -\infty\ \rangle$
$\langle\ step = Scan\ \wedge\ reset\ \wedge\ scanned = -\infty$
    $\overset{\mathcal{WF}}{\Longrightarrow} ptr \leftarrow 0$
        $reset \leftarrow dirty \leftarrow \mathbf{false}\ \rangle$
$\langle\ step = Scan\ \wedge\ scanned < ptr\ \wedge\ dirty$
    $\overset{\mathcal{WF}}{\Longrightarrow} reset \leftarrow \mathbf{true}\ \rangle$
```

scan

```
$\langle\ step = Scan\ \wedge\ scanned < ptr < limit$
    $\overset{\mathcal{WF}}{\Longrightarrow} scanned \leftarrow ptr\ \rangle$
$\langle\ step = Scan\ \wedge\ scanned = ptr < limit$
    $\overset{\mathcal{WF}}{\Longrightarrow}$ if $heap[ptr].color = Gray$ then
            $blacken \leftarrow blacken \cup \{ptr\}$
        $ptr \leftarrow ptr + heap[ptr].size + 1\ \rangle$
$\langle\ step = Scan\ \wedge\ ptr = limit\ \wedge\ \neg reset\ \wedge\ \neg dirty$
  $\wedge\ cache = fields = \emptyset\ \wedge\ blacken = trace = \emptyset$
    $\overset{\mathcal{WF}}{\Longrightarrow} reset \leftarrow \mathbf{true}$
        $rover \leftarrow ptr \leftarrow sublimit \leftarrow 0$
        $step \leftarrow Sweep\ \rangle$
```