A Correct and Useful Incremental Copying Garbage Collector

Martin Kero Johan Nordlander Per Lindgren
Luleå University of Technology, SE-97187 Luleå, Sweden
Martin.Kero@ltu.se

Abstract

Designing a garbage collector with real-time properties is a particularly difficult task, involving the construction of both an incremental run-time algorithm as well as methods enabling a priori reasoning about schedulability in two dimensions (time and memory usage in conjunction). In order to comply with such ambitious goals with any amount of formal rigor, a comprehensive understanding of the actual algorithm used is of course a fundamental requirement. In this paper we present a formal model of an incremental copying garbage collector, where each atomic increment is modeled as a transition between states of a heap process. Soundness of the algorithm is shown by proving that the garbage collecting heap process is weakly bisimilar to a non-collecting heap with infinite storage space. In addition, we show that our collector is both terminating and useful, in the sense that it actually recovers the unreachable parts of any given heap in a finite number of steps.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Reliability, Verification

Keywords Incremental Copying Garbage Collection, Labeled Transition System

1. Introduction

As embedded real-time software systems becomes increasingly complex, the need for more sophisticated run-time system features becomes a necessity. The ability to use shared dynamic data in conjunction with concurrent execution makes automatic memory management a requirement. However, designing automatic memory management for real-time systems is an inherently difficult task. In this paper we present what we believe is a necessary first part: a formal model of an incremental copying garbage collector (Section 2). Our collector is defined in terms of very small atomic increments, and we use the process calculus techniques of a labeled transition system and bisimulation to model mutator interaction and prove correctness (Sections 3 and 4). We furthermore show that our garbage collector is useful, in the sense that it actually recovers unreachable portions of a heap (Section 5).

Apart from taking the first step towards the overall goal of designing a formally proven correct real-time garbage collection system (which in the end should include both a correct algorithm and statically analyzable properties such as schedulability and memory usage, etc.), the main motivation behind this paper is to show that it is possible to reason about the garbage collector and the mutator(s) as process terms. This has, to the best of our knowledge, not been done before. Another significant motivation is that formal correctness of incremental copying garbage collection has not been shown until very recently [10]. A third motivation is to demonstrate that it is also possible to formally reason about the usefulness of an incremental copying garbage collector.

2. The Garbage Collector

Before we go into details of our model we would like to spend some time describing the algorithm more informally. Even though the way we present the algorithm is rather non-conventional, the algorithm itself is quiet well-known. We use Cheney's in-place breadth-first traversal of gray objects [7], and we deploy a read barrier similar to that of Brooks [5]; i.e., reads to old copies in white space are forwarded to their corresponding new copies in the gray/black heap. Furthermore, we use a write barrier in the style of Steele [14, 15], where the tri-color invariant is upheld by reverting black objects to gray upon mutation. The driving force behind these choices is that we strive to amortize as much of the garbage collection work into scheduled garbage collection time instead of taking the cost when a real-time task allocates, reads, or writes to the heap. These choices may need to be reconsidered depending on the mutator and the mission of the application. However, the algorithm and proofs can easily be reworked for this purpose.

We will now continue by describing the garbage collector in more detail. Let x, y, z range over heap addresses, and

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-893-0/07/0010...$5.00

let n range over integers. Let u, v range over values and be either a heap address or an integer. Let U and V range over sequences of such values.

A heap node can be either a sequence of values (enclosed by angle brackets $\langle V \rangle$) or a single forwarding address (denoted $\bullet x$). A heap H is a finite mapping from addresses to nodes, as captured by the following grammar.

$$\begin{array}{llcl} \text{(heap)} & H & ::= & \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\} \\ \text{(node)} & o & ::= & \langle V \rangle \mid \bullet\, x \\ \text{(value)} & v & ::= & x \mid n \end{array}$$

The domain $dom(H)$ of a heap $H = \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\}$ is the set $\{x_1, \ldots, x_n\}$. A heap look-up is defined as $H(x) = o$ if $x \mapsto o \in H$. We will write U, V to denote the concatenation of the value sequences U and V. Along the same line, we will write H, G for the concatenation of heaps H and G, provided their domains are disjoint.

For garbage collection purposes, a heap can be described as a triple of subheaps separated by heap borders ($\mid$). A heap border has the same meaning as the regular concatenation operator for heaps, but it also provides necessary bookkeeping information. The three subheaps captures the white, gray, and black part of the heap as in the *tricolor abstraction* [8].

The algorithm is based on a labeled transition system (LTS), where garbage collection transitions are so called internal (τ) transitions. Each individual τ transition constitutes an atomic increment by the garbage collector. In Figure 1 and 2, all possible internal transitions are shown. Determinism between different internal transitions are achieved by pattern matching, as each configuration matches only one single clause. The clauses are furthermore divided into two groups, which we call *scan* and *copy* transitions. A garbage collection cycle is a sequence of such transitions beginning with a START transition and ending with a DONE transition.

$$H_0 \xrightarrow{\text{START}} H_1 \longrightarrow \ldots \longrightarrow H_{n-1} \xrightarrow{\text{DONE}} H_n$$

Notice that an active garbage collection cycle is identified by a non-empty white subheap.

In contrast, the external triggered transitions, such as mutations and allocations are labeled, denoted by $H \xrightarrow{l} H'$. We will look into these in more detail in the next section. We use a single root pointer r to capture the root-set. Even though a real system most likely will contain more than one root, this can easily be captured in our model by adjusting the content of the node pointed to by r. I.e., the actual root-set is the content of the node labelled r.

At the beginning of a cycle the heap has the form $\emptyset \mid G, r \mapsto \langle V \rangle \mid \emptyset$, and initiating a garbage collection cycle (START) invalidates the whole heap except for the root node. That is, all nodes but r are made white by placing them to the left of the white-gray heap border. The algorithm then proceeds by scanning gray nodes (SCANSTART) and takes proper actions when embedded addresses are encountered. This is accomplished by a *scan pointer* that traverses the gray nodes. The scan pointer is denoted by the symbol $\downarrow$ and has similar function and purpose as the heap borders, i.e. regular concatenation as well as bookkeeping information. When a whole node has been scanned it is promoted from gray to black (SCANDONE). When there are no more gray nodes to scan the garbage collector is finished (DONE).

During scanning of a gray node, encountering an address may result in one of three possible actions. If the address found is not in the white heap, the algorithm just goes on to examine the next gray node field (SCANADDR). If the address is in the white heap, and the corresponding node is a forwarding node, the forwarding address replaces the encountered address (FORWARD). If, on the other hand, the node found is a regular white node, copying is initiated (COPYSTART). This is done by allocating a new empty node in the gray heap and *locking* the scan pointer, which we denote by the alternative concatenation symbol symbol $\uparrow_z$ (where the index is the address of the new empty node). The white node is then copied word by word (COPYWORD) until the whole node has been copied (COPYDONE). At this point, the address of the newly allocated node replaces the old encountered address, and the original white node is converted into a forwarding node.

In addition to these scenarios, two other transitions are also defined: SCANRESTART and COPYRESTART. These transitions are taken when mutations occur during garbage collection, and will be described in more detail in the next sections.

3. The Mutator

In order to capture the behavior of an interacting mutator we begin by defining a recursive function *read* that is based on the notion of an abstract *path* beginning in a *root*.

DEFINITION 3.1.

$$\textit{(path)} \quad p, q ::= \varepsilon \mid i : p \quad \textit{where } i \textit{ is an index}$$

DEFINITION 3.2. *For some heap H and root x,*

$$\begin{array}{ll} read(H, x, i : p) = read(H, V[i], p) & \textit{if } H(x) = \langle V \rangle \\ read(H, n, \varepsilon) = n & \\ read(H, x, \varepsilon) = x & \textit{if } H(x) \neq \bullet y \\ read(H, x, p) = read(H, y, p) & \textit{if } H(x) = \bullet y \end{array}$$

The mutator activities can now be defined as a set of labeled transitions $H \xrightarrow{l} H'$ (see Figure 3) where the label l captures the different means a mutator can interact with a heap.

$$\text{(label)} \quad l ::= r(p = n) \mid w(p = n) \mid w(p = q) \mid a(p)$$

START	$\emptyset \mid G, r \mapsto \langle V \rangle \mid \emptyset$	$\longrightarrow$	$G \mid r \mapsto \langle V \rangle \mid \emptyset$	
SCANSTART	$W \mid G, x^{\dagger} \mapsto \langle V \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle \downarrow V \rangle \mid B$	$W \neq \emptyset$, †may be dirty
SCANINT	$W \mid G, x \mapsto \langle V \downarrow n, V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle V, n \downarrow V' \rangle \mid B$	$W \neq \emptyset$
SCANADDR	$W \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle V, y \downarrow V' \rangle \mid B$	$W \neq \emptyset, y \notin dom(W)$
SCANRESTART	$W \mid G, \dot{x} \mapsto \langle V \downarrow V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle \downarrow V, V' \rangle \mid B$	$W \neq \emptyset$
SCANDONE	$W \mid G, x \mapsto \langle V \downarrow \rangle \mid B$	$\longrightarrow$	$W \mid G \mid x \mapsto \langle V \rangle, B$	$W \neq \emptyset$
DONE	$W \mid \emptyset \mid B$	$\longrightarrow$	$\emptyset \mid B \mid \emptyset$	

Figure 1. Scan transitions.

FORWARD	$W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V, z \downarrow V' \rangle \mid B$	
COPYSTART	$W, y^{\dagger} \mapsto \langle U \rangle, W' \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \langle U \rangle, W' \mid z \mapsto \langle\rangle, G, x \mapsto \langle V \uparrow_z y, V' \rangle \mid B$	†may be dirty; z is fresh
COPYWORD	$W, y \mapsto \langle U, u, U' \rangle, W' \mid G, z \mapsto \langle U \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \langle U, u, U' \rangle, W' \mid G, z \mapsto \langle U, u \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B$	†may be dirty
COPYRESTART	$W, \dot{y} \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle U' \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle\rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B$	†may be dirty
COPYDONE	$W, y \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle U \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \bullet z, W' \mid G, z \mapsto \langle U \rangle, G', x^{\dagger} \mapsto \langle V, z \downarrow V' \rangle \mid B$	†may be dirty

Figure 2. Copy transitions.

Here $r(p = n)$ means reading the integer n through the path p, $w(p = n)$ means writing the integer n at the end of path p, $w(p = q)$ means writing the value found at the end of path q at the end of path p, and $a(p)$ means allocate a fresh node and write the address of it at the end of path p. The behavior of $\xrightarrow{l}$ transitions is shown in Figure 3.

Two important details of Figure 3 require special mentioning. Firstly, clauses MUT/ALLOCMUT mark the mutaded node as *dirty*, indicated by a dot over the node's address ($\dot{x}$). Secondly, if a black node is mutated (MUTB / ALLOCMUTB), the node is reverted back to gray.

4. Correctness of the Garbage Collector

The tri-color invariant [8] is captured by a property we denote as *well-formedness*.

DEFINITION 4.1. ***(Well-formedness).*** *A heap $H = W|G|B$ is* well-formed *iff*

(1) $$B(x) = \langle V \rangle \textit{ implies } V \cap dom(W) = \emptyset$$
$$\wedge$$
(2) $$H(x) = \bullet y \textit{ implies } (x \in dom(W) \wedge y \notin dom(W))$$
$$\wedge$$
(3) $$r \in dom(G, B)$$
$$\wedge$$
(4) $$H = W \mid G, x \mapsto \langle V \downarrow U \rangle \mid B \textit{ implies } V \cap dom(W) = \emptyset$$
$$\wedge$$
(5) $$H = W \mid G, x \mapsto \langle V \uparrow_z y, U \rangle \mid B \textit{ implies } W(y) = \langle V, V' \rangle \textit{ and } G(z) = \langle V \rangle$$

Informally, a heap is well-formed if and only if no black nodes contain addresses to any white nodes (1), a forwarding

MUTW
$$\begin{array}{c} H = W, x \mapsto \langle U \rangle, W' \mid G \mid B \\ \xrightarrow{w(p:i=q)} \\ H' = W, \dot{x} \mapsto \langle U[i] := y \rangle, W' \mid G \mid B \end{array}$$
if $read(H, r, p) = x$ and $read(H, r, q) = y$

MUTG
$$\begin{array}{c} H = W \mid G, x \mapsto \langle U \rangle, G' \mid B \\ \xrightarrow{w(p:i=q)} \\ H' = W \mid G, \dot{x} \mapsto \langle U[i] := y \rangle, G' \mid B \end{array}$$
if $read(H, r, p) = x$ and $read(H, r, q) = y$

MUTB
$$\begin{array}{c} H = W \mid G \mid B, x \mapsto \langle U \rangle, B' \\ \xrightarrow{w(p:i=q)} \\ H' = W \mid \dot{x} \mapsto \langle U[i] = y \rangle, G \mid B, B' \end{array}$$
if $read(H, r, p) = x$ and $read(H, r, q) = y$

WRITE
$$H, x \mapsto \langle U \rangle, H' \xrightarrow{w(p:i=n)} H, x \mapsto \langle U[i] := n \rangle, H'$$
if $read(H, r, p) = x$

READ
$$H \xrightarrow{r(p=n)} H$$
if $read(H, r, p) = n$

ALLOCMUTW
$$\begin{array}{c} H = W, x \mapsto \langle U \rangle, W' \mid G \mid B \\ \xrightarrow{a(p:i)} \\ H' = W, \dot{x} \mapsto \langle U[i] := z \rangle, W' \mid z \mapsto \langle\rangle, G \mid B \end{array}$$
if $read(H, r, p) = x$

ALLOCMUTG
$$\begin{array}{c} H = W \mid G, x \mapsto \langle U \rangle, G' \mid B \\ \xrightarrow{a(p:i)} \\ H' = W \mid z \mapsto \langle\rangle, G, \dot{x} \mapsto \langle U[i] := z \rangle, G' \mid B \end{array}$$
if $read(H, r, p) = x$

ALLOCMUTB
$$\begin{array}{c} H = W \mid G \mid B, x \mapsto \langle U \rangle, B' \\ \xrightarrow{a(p:i)} \\ H' = W \mid \dot{x} \mapsto \langle U[i] := z \rangle, z \mapsto \langle\rangle, G \mid B, B' \end{array}$$
if $read(H, r, p) = x$

Figure 3. Mutator transitions.

node can only be white and its forwarding address cannot be to a white node (2), the root node r is either in G or B (3), and while scanning a node in the gray heap nothing left of the scan pointer can be pointers to the white heap (4).

Lemma 4.1 states that a garbage collection transition of the heap preserves well-formedness.

LEMMA 4.1. *If H is well-formed and $H \longrightarrow H'$ then H' is also well-formed.*

Proof By case study on the clauses defining $\longrightarrow$. For convenience, let $H = W|G|B$ and $H' = W'|G'|B'$. Observe that since H is well-formed, the root node is in G or B. Furthermore, since only transition START removes nodes from $G \cup B$, and there makes an explicit exception for r, (3) is upheld throughout.

START:

Since $B' = \emptyset$, (1) holds vacuously. Since H is well-formed and $W = \emptyset$, it cannot include any forwarding nodes. Thus, since no new forwarding node is introduced by the transition, H' does not include any forwarding nodes, i.e. (2) holds vacuously. (4) and (5) also hold vacously.

SCANSTART, SCANINT, SCANADDR, SCANRESTART, FORWARD, COPYSTART, COPYWORD, COPYRESTART, COPYDONE:

Since neither B nor W is modified, and no forwarding node is created, both (1) and (2) are upheld by the transition. Since none of the transitions extend the sequence of values left of the scan pointer with pointers to W, (4) is also upheld. Furthermore, since COPYSTART requires that $y \in dom(W)$ and z is a fresh node allocated in G, (5) is upheld. For COPYWORD and COPYRESTART, it is immediate from the structure of y and z that (5) is upheld.

SCANDONE:

From Definition 4.1 follows that $V \cap dom(W) = \emptyset$. Thus (1) holds for H'. SCANDONE does not create any forwarding nodes and no new node is introduced to W (i.e. $W = W'$). Thus, (2) is upheld by the transition. (4) and (5) hold vacously.

DONE:

Since H is well-formed, we know that B does not contain any forwarding nodes. Thus, both (1) and (2) hold vacuously.

MUTW,MUTG,READ,ALLOCMUTW,ALLOCMUTG:

Since B is not modified (i.e. $B = B'$) and $dom(W) = dom(W')$ (1) is upheld by the transitions. Since no forwarding nodes are introduced and, again, $dom(W) = dom(W')$, (2) is upheld by the transitions. Since the mutated node is made dirty, (4) and (5) hold vacously.

MUTB,ALLOCMUTB:

Since $B' \subset B$, i.e. no new node is introduced, and $dom(W) = dom(W')$, (1) is upheld by the transition. Since no new forwarding node is introduced and, again, $dom(W) = dom(W')$, (2) is also upheld by the transition. (4) and (5) hold vacously. ∎

Lemma 4.2 captures the property of determinism.

LEMMA 4.2. *If H is well-formed then the structure of H matches the pattern of exactly one τ transition.*

Proof By case study on all possible structures of a well-formed H.

$H = W \mid \emptyset \mid B$ where $W \neq \emptyset, B \neq \emptyset$:

Only one clause matches this pattern (DONE), and it will succeed no matter what internal structure W and B have.

$H = W \mid G \mid \emptyset$ where $W \neq \emptyset, G \neq \emptyset$,

$H = W \mid G \mid B$ where $W \neq \emptyset, G \neq \emptyset, B \neq \emptyset$:

Since G is not empty, there is at least one node in it. Thus, G must match the pattern $G = G', x \mapsto o$. We now need to look at the possible structures of $x \mapsto o$.

From the definition of a node, o is either a regular node ($\langle V \rangle$) or a forwarding address ($\bullet z$). However, from Definition 4.1 (well-formedness) follows that $x \mapsto o$ cannot be a forwarding node. We now need to look at the possible structures of $\langle V \rangle$. We have three cases.

$x \mapsto \langle V \rangle$:

Matches SCANSTART.

$\dot{x} \mapsto \langle V \downarrow V' \rangle$:

Matches SCANRESTART.

$x \mapsto \langle V \downarrow V' \rangle$:

$V' = \emptyset$:

Matches SCANDONE.

$V' = v, V''$:

$v = n$:

Matches SCANINT.

$v = y, y \notin dom(W)$:

Matches SCANADDR.

$v = y, y \mapsto \langle U \rangle \in W$:

Matches COPYSTART.

$v = y, y \mapsto \bullet z \in W$:

Matches FORWARD.

$x \mapsto \langle V \uparrow_z V' \rangle$:

From definition 4.1 follows that $V' = y, V''$, $y \mapsto \langle U \rangle \in W$, and $z \mapsto \langle U' \rangle \in G$. Furthermore, it follows that $H(z) \subseteq H(y)$ if y is not dirty. Thus we have the following cases.

$\dot{y} \mapsto \langle U \rangle$:

Matches COPYRESTART.

$y \mapsto \langle U \rangle, z \mapsto \langle U' \rangle, U' \subset U$:

Matches COPYWORD.

$y \mapsto \langle U \rangle, z \mapsto \langle U' \rangle, U' = U$:

Matches COPYDONE.

$H = \emptyset \mid G \mid \emptyset$ where $G \neq \emptyset$:

Since H is well-formed it includes at least the root node. I.e. $G = G', r \mapsto \langle V \rangle$. START will match whatever structure G' has.

$H = W \mid \emptyset \mid \emptyset$:

Since H is required to be well-formed, $r \in dom(G, B)$ which means that this structure is not well-formed. ∎

4.1 Termination

Theorem 4.3 captures a very important property: termination. It states that for every well-formed heap that is within a garbage collection cycle (the START transition has been taken), a state where the DONE transition can be taken may be reached after a finite number of garbage collection steps.

THEOREM 4.3. ***(Termination)*** *For every well-formed $H = W|G|B$ ($W \neq \emptyset$) there is some $H' = W'|\emptyset|B'$ such that $H \longrightarrow_* H'$.*

Proof We first need to define a notion of *weight* for a heap. Since the algorithm is incremental down to single word copy actions and allows mutations in between, we need a fine-grained metric for weight in order to capture the small weight-losses in each increment. We do this by a triple metric, where the first element is the dominant weight factor while as the second and the third elements capture the progress of scanning and copying a single node, respectively. The weight of a heap is defined in Figure 4.

Heap weights are ordered lexicographically; i.e., the first element is the most significant and the third (last) element is the least significant. For example, suppose we have two heaps H and H', and $weight(H) = (a, b, c)$ and $weight(H') = (d, e, f)$. Then,

$$\begin{aligned}&(a,b,c) < (d,e,f)\\ &\quad \text{if } (a < d) \text{ or } (a == d \wedge b < e)\\ &\qquad \text{or } (a == d \wedge b == e \wedge c < f).\end{aligned}$$

The proof relies on the fact that each τ transition possible when $W \neq \emptyset$ and $G \neq \emptyset$ reduces the weight of the heap.

SCANSTART:

$weight(W \mid G, x \mapsto \langle V \rangle \mid B) > weight(W \mid G, x \mapsto \langle \downarrow V \rangle \mid B)$

$\Leftarrow$

$(w(W \mid G, x \mapsto \langle V \rangle \mid B), \infty, \infty) > (w(W \mid G, x \mapsto \langle \downarrow V \rangle \mid B), |V|, \infty)$

$\Leftarrow$

Let $a = w(W \mid G, x \mapsto \langle V \rangle \mid B)$, then,

$(a, \infty, \infty) > (a, |V|, \infty)$, which is true since $\infty > |V|$.

$$
\begin{array}{rcl}
\mathit{weight}(H) & \overset{def}{=} & (w(H), w'(H), w''(H)) \\
w(H) & \overset{def}{=} & \sum_{x \in \mathit{dom}'(H)} (w(H,x) + w(H, H(x))) \\
\mathit{dom}'(H) & \overset{def}{=} & \{x \mid x \in \mathit{dom}(H) \wedge \nexists y, V, V'.H(y) = \langle V \uparrow_x V' \rangle\} \\
w(H, \langle V \rangle) & \overset{def}{=} & \sum_{v \in V} w(H, v) \\
w(H, \bullet x) & \overset{def}{=} & 0 \\
w(H, n) & \overset{def}{=} & 1 \\
w(W|G|B, x) & \overset{def}{=} & \text{if } x \in \mathit{dom}(B) \text{ then } 2 \\
 & & \text{else if } x \in \mathit{dom}(G) \text{ then } 3 \\
 & & \text{else if } x \in \mathit{dom}(W) \text{ then } 4
\end{array}
$$

```
$w'(H) \overset{def}{=}$ if $\exists x, V, V'.H(x) = \langle V \downarrow V' \rangle \vee H(x) = \langle V \uparrow V' \rangle$ then
    if $\dot{x}$ then
        $\infty$
    else
        $|V'|$
else
    $\infty$

$w''(H) \overset{def}{=}$ if $\exists x, V, V', y, z.H(x) = \langle V \uparrow_z y, V' \rangle$ then
    if $\dot{y}$ then
        $\infty$
    else
        $|H(y)| - |H(z)|$
else
    $\infty$
```

Figure 4. Weight of a heap

SCANINT,SCANADDR:

$\mathit{weight}(W \mid G, x \mapsto \langle V \downarrow v, V' \rangle \mid B) > \mathit{weight}(W \mid G, x \mapsto \langle V, v \downarrow V' \rangle \mid B)$
$\Leftarrow$
$(w(W \mid G, x \mapsto \langle V \downarrow v, V' \rangle \mid B), |n, V'|, \infty) > (w(W \mid G, x \mapsto \langle V, v \downarrow V' \rangle \mid B), |V'|, \infty)$
$\Leftarrow$
Let $a = w(W \mid G, x \mapsto \langle V, v, V' \rangle \mid B)$, then, $(a, |v, V'|, \infty) > (a, |V'|, \infty)$, which is true since $|v, V'| > |V'|$.

SCANRESTART:

$\mathit{weight}(W \mid G, \dot{x} \mapsto \langle V \downarrow V' \rangle \mid B) > \mathit{weight}(W \mid G, x \mapsto \langle \downarrow V, V' \rangle \mid B)$
$\Leftarrow$
$(w(W \mid G, \dot{x} \mapsto \langle V \downarrow V' \rangle \mid B), \infty, \infty) > (w(W \mid G, x \mapsto \langle \downarrow V, V' \rangle \mid B), |V, V'|, \infty)$
$\Leftarrow$
Let $a = w(W \mid G, x \mapsto \langle V, V' \rangle \mid B)$, then, $(a, \infty, \infty) > (a, |V, V'|, \infty)$, which is true since $\infty > |V, V'|$.

SCANDONE:
$weight(W \mid G, x \mapsto \langle V \downarrow\rangle \mid B) > weight(W \mid G \mid x \mapsto \langle V\rangle, B)$
$\Leftarrow$
$(w(W \mid G, x \mapsto \langle V \downarrow\rangle \mid B), 0, \infty) > (w(W \mid G \mid x \mapsto \langle V\rangle, B), \infty, \infty)$
$\Leftarrow$
Let $a = w(W \mid G \mid B)$, $H = W \mid G, x \mapsto \langle V \downarrow\rangle \mid B$, and $H' = W \mid G \mid x \mapsto \langle V\rangle, B$, then,
$(a + w(x \in H) + w(\langle V\rangle), 0, \infty) > (a + w(x \in H') + w(\langle V\rangle), \infty, \infty)$, which is true since $w(x \in H) > w(x \in H')$.

FORWARD:
$weight(W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V \downarrow y, V'\rangle \mid B) > weight(W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V, z \downarrow V'\rangle \mid B)$
$\Leftarrow$
Let $a = w(W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V, V'\rangle \mid B)$ then,
$(a + w(y), |y, V'|, \infty) > (a + w(z), |V|, \infty)$
From Definition 4.1 follows that $z \notin dom(W, W')$. Thus, $(a + w(y), |y, V'|, \infty) > (a + w(z), |V|, \infty)$ is true since $w(y) > w(z)$.

COPYSTART:
$weight(W, y \mapsto \langle U\rangle, W' \mid G, x \mapsto \langle V \downarrow y, V'\rangle \mid B) > weight(W, y \mapsto \langle U\rangle, W' \mid z \mapsto \langle\rangle, G, x \mapsto \langle V \uparrow_z y, V'\rangle \mid B)$
$\Leftarrow$
Let $a = w(W, y \mapsto \langle U\rangle, W' \mid G, x \mapsto \langle V, y, V'\rangle \mid B)$, then
$(a, |y, V'|, \infty) > (a + 0, |y, V'|, |U| - 0)$ which is true since $\infty > |U|$.

COPYWORD:
$weight(W, y \mapsto \langle U, u, U'\rangle, W' \mid G, z \mapsto \langle U\rangle, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B) > weight(W, y \mapsto \langle U, u, U'\rangle, W' \mid G, z \mapsto \langle U, u\rangle, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B)$
$\Leftarrow$
Let $a = w(W, y \mapsto \langle U, u, U'\rangle, W' \mid G, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B)$, then
$(a, |y, V'|, |U, u, U'| - |U|) > (a, |y, V'|, |U, u, U'| - |U, u|)$
$\Leftarrow$
$(a, |y, V'|, |u, U'|) > (a, |y, V'|, |U'|)$ which is true since $|u, U'| > |U'|$.

COPYRESTART:
$weight(W, \dot{y} \mapsto \langle U\rangle, W' \mid G, z \mapsto \langle U'\rangle, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B) > weight(W, y \mapsto \langle U\rangle, W' \mid G, z \mapsto \langle\rangle, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B)$
$\Leftarrow$
Let $a = w(W, y \mapsto \langle U\rangle, W' \mid G, z \mapsto \langle\rangle, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B)$ then,
$(a, |y, V'|, \infty) > (a, |y, V'|, |U|)$ which is true since $\infty > |U|$.

COPYDONE:
$weight(W, y \mapsto \langle U\rangle, W' \mid G, z \mapsto \langle U\rangle, G', x \mapsto \langle V \uparrow_z y, V'\rangle \mid B) > weight(W, y \mapsto \bullet z, W' \mid G, z \mapsto \langle U\rangle, G', x \mapsto \langle V, z \downarrow V'\rangle \mid B)$
$\Leftarrow$
Let $a = w(W, W' \mid G, z \mapsto \langle U\rangle, G', x \mapsto \langle V, V'\rangle \mid B)$ then,
$(a + 2 * w(y) + w(\langle U\rangle), |y, V'|, 0) > (a + w(y) + w(z) + w(\langle U\rangle), |V'|, \infty)$ which is true since $w(y) > w(z)$. ▮

4.2 Preservation

In this section, we show that our garbage collector preserves the *meaning* of the heap. In order to do that we need to define what it means for two heaps to be equivalent. The first attempt is to define a read-equivalence; that is, two heaps are equivalent if and only if a non-address read at a legal path gives the same result in both heaps. However, since a heap may contain shared paths, we may get identical read-behavior in two structurally different heaps that might be distinguished after further mutation. In order to capture equivalence even after mutation we need to assure that, for two heaps to be equivalent, they are both read-equivalent and contain exactly the same shared paths.

We begin by defining the predicate $join$, which is a boolean function that takes a heap, two starting addresses, and two paths as parameters, and returns True if the two paths starting at each address reach the same node. We also define a variant for the special case when the two paths both start at the same address.

DEFINITION 4.2.

$$\begin{aligned} join(H, x, y, p, q) &\overset{def}{=} read(H, x, p) = z \\ &\quad\ \wedge read(H, y, q) = z \\ join(H, x, p, q) &\overset{def}{=} join(H, x, x, p, q) \end{aligned}$$

We can now define the equivalence relation between heaps.

DEFINITION 4.3. *Two heaps H and H' are structurally equivalent, written $H \equiv H'$, if and only if,*
$\forall p \,.\, read(H, r, p) = n \iff read(H', r, p) = n$
and $(\forall q \,.\, join(H, r, p, q) \iff join(H', r, p, q))$

This equivalence relation is preserved by mutation; in fact it is preserved by all possible mutator activities, as captured by the following lemma.

LEMMA 4.4. *If $H \equiv H'$ and $H \xrightarrow{l} \hat{H}$, then $H' \xrightarrow{l} \hat{H}'$ and $\hat{H} \equiv \hat{H}'$.*

Proof By case study on $H \xrightarrow{l} \hat{H}$. We only show the case for MUTW; all other cases are either similar or trivial.

$$\begin{array}{rcr} H = & W, x \mapsto \langle U\rangle, W' \mid G \mid B & read(H,r,p) = x \\ & \xrightarrow{w(p:i=q)} & read(H,r,q) = y \\ \hat{H} = & W, \dot{x} \mapsto \langle U[i] := y\rangle, W' \mid G \mid B & \end{array}$$

From the definition of $H \equiv H'$ (Def. 4.3) follows that:

$join(H,r,p,p) = True$ (i.e. x) $\iff join(H',r,p,p) = True$ (let it be x'). (I)
$join(H,r,q,q) = True$ (i.e. y) $\iff join(H',r,q,q) = True$ (let it be y'). (II) $join(H,r,p:i,p:i) \iff join(H',r,p:i,p:i)$ (III)
$read(H,r,p:i) = n \iff read(H',r,p:i) = n$ (IV)
$\forall p', q'$.

$$\begin{array}{rcll} read(H,x,p') = n & \iff & read(H',x',p') = n & \text{(V)} \\ read(H,y,p') = n & \iff & read(H',y',p') = n & \text{(VI)} \\ join(H,x,r,p',q') & \iff & join(H',x',r,p',q') & \text{(VII)} \\ join(H,y,r,p',q') & \iff & join(H',y',r,p',q') & \text{(VIII)} \\ join(H,x,p',q') & \iff & join(H',x',p',q') & \text{(IX)} \\ join(H,y,p',q') & \iff & join(H',y',p',q') & \text{(X)} \\ join(H,x,y,p',q') & \iff & join(H',x',y',p',q') & \text{(XI)} \end{array}$$

From (I), (II), (III), and (IV) follows that $H' \xrightarrow{w(p:i=q)} \hat{H}'$.
We now have to show that $\hat{H} \equiv \hat{H}'$.
From the definition of $\equiv$ follows that:
$\forall p'$.

$$\begin{array}{rcll} read(\hat{H},r,p') = n & \iff & read(\hat{H}',r,p') = n & (1) \\ & \wedge & & \\ (\forall q' \,.\, join(\hat{H},r,p',q') & \iff & join(\hat{H}',r,p',q') & (2) \end{array}$$

We proceed by induction on the length of p'.

Base case: $p' = \varepsilon$
(1) holds vacuously.
(2) by induction on the length of q'
Base case: $q' = \varepsilon$
$join(\hat{H},r,\varepsilon,\varepsilon) = join(\hat{H}',r,\varepsilon,\varepsilon) = True$ (i.e. both for r)
Otherwise (i.e. $q' \neq \varepsilon$):
Let $q' = q_1' : j : q_2'$ and $read(\hat{H},r,q_1' : j) = z$ $(read(\hat{H}',r,q_1' : j) = z')$
Induction hypothesis:
$join(\hat{H},r,\varepsilon,q_1') \iff join(\hat{H}',r,\varepsilon,q_1')$
$join(\hat{H},r,z,\varepsilon,q_2') \iff join(\hat{H},r,z,\varepsilon,q_2')$
Case study on q_1':
$read(\hat{H},r,q_1') = x$ $(read(\hat{H}',r,q_1') = x')$:
$j = i$:From the induction hypothesis and (IX) follows that
$join(\hat{H},r,\varepsilon,q') \iff join(\hat{H},r,\varepsilon,q')$
$j \neq i$:From the induction hypothesis and (VII) follows that
$join(\hat{H},r,\varepsilon,q') \iff join(\hat{H},r,\varepsilon,q')$
$read(\hat{H},r,q_1') \neq x$: $(read(\hat{H}',r,q_1') \neq x')$
It follows directly from the induction hypothesis that
$join(\hat{H},r,\varepsilon,q') \iff join(\hat{H},r,\varepsilon,q')$

Otherwise $(p' \neq \varepsilon)$:
Let $p' = p_1' : j : p_2'$ and $read(\hat{H},r,p_1' : j) = z$ $(read(\hat{H}',r,p_1' : j) = z')$
Induction hypothesis:
$read(\hat{H},r,p_1') = n \iff read(\hat{H}',r,p_1') = n \wedge$
$(\forall q'.join(\hat{H},r,p_1',q') \iff join(\hat{H}',r,p_1',q')) \wedge$
$read(\hat{H},z,p_2') = n \iff read(\hat{H}',z',p_2') = n \wedge$
$(\forall q'.join(\hat{H},z,r,p_2',q') \iff join(\hat{H}',z',r,p_2',q'))$
Case study on p_1':
$read(\hat{H},r,p_1') = x$ $(read(\hat{H}',r,p_1') = x')$:
$j = i$:
(1):
From the induction hypothesis and (VI) follows that
$read(\hat{H},r,p') = n \iff read(\hat{H}',r,p') = n$
(2):
By induction on the length of q'
Base case: $q' = \varepsilon$
Similar to (2) in base case since $join(\hat{H},r,p',\varepsilon) = join(\hat{H},r,\varepsilon,p')$
Otherwise (i.e. $q' \neq \varepsilon$):
Let $q' = q_1' : k : q_2'$ and $read(\hat{H},r,q_1' : k) = \hat{x}$ $(read(\hat{H}',r,q_1' : k) = \hat{x}')$
Induction hypothesis:
$join(\hat{H},y,r,p_2',q_1') \iff join(\hat{H}',y',r,p_2',q_1')$
$join(\hat{H},y',\hat{x},p_2',q_2') \iff join(\hat{H}',y',\hat{x}',p_2',q_2')$
Case study on q_1'
$read(\hat{H},r,q_1') = x$: $(read(\hat{H}',r,q_1') = x')$
$k = i$:
From the induction hypotheses and (X) follows that
$join(\hat{H},r,p',q') \iff join(\hat{H}',r,p',q')$
$k \neq i$:
From the induction hypotheses and (XI) follows that
$join(\hat{H},r,p',q') \iff join(\hat{H}',r,p',q')$
$read(\hat{H},r,q_1') \neq x$: $(read(\hat{H}',r,q_1') \neq x')$
It follows from the induction hypotheses and (IX) that
$join(\hat{H},r,p',q') \iff join(\hat{H}',r,p',q')$

$j \neq i$:

(1):

From the induction hypothesis and (V) follows that

$read(\hat{H}, r, p') = n \iff read(\hat{H}', r, p') = n$

(2):

Similar to (2) in previous case (i.e. $j = i$).

$read(\hat{H}, r, p'_1) \neq x$: $(read(\hat{H}', r, p'_1) \neq x')$

(1):

From the induction hypothesis follows that

$read(\hat{H}, r, p') = n \iff read(\hat{H}', r, p') = n$

(2):

Similar to (2) in previous cases.

∎

4.2.1 Bisimulating Garbage Collection.

In order to show that our garbage collector preserves the meaning of a heap, we adopt the notion of *weak bisimulation* between processes [11]. In our context, a process is simply a heap that may take one $\xrightarrow{\tau}$ transition or one of all possible $\xrightarrow{l}$ transitions. We define two processes, one with garbage collection and one without.

$$P_{GC}(H) \overset{def}{=} \sum_{H \xrightarrow{l} H'} l.P(H') \quad + \quad \sum_{H \xrightarrow{\tau} H''} \tau.P(H'')$$

$$P(H) \overset{def}{=} \sum_{H \xrightarrow{l} H'} l.P(H')$$

$$\mathcal{P} \overset{def}{=} \{P(H) \mid H \text{ is well-formed}\} \cup \{P_{GC}(H) \mid H \text{ is well-formed}\}$$

A heap process is simply the sum of all l transitions and (for the garbage collecting case) the τ transitions possible for a certain heap H. Note that we use same notation for transitions made by a heap and a heap process. This should however not lead to any confusion since the actual meaning of a transition is still the same.

In order to introduce bisimilarity we need to recapitulate some standard notions from the process calculus literature [11].

DEFINITION 4.4. *The relations $\Rightarrow$ and $\overset{l}{\Rightarrow}$ are defined as follows:*

$\Rightarrow \overset{def}{=} \xrightarrow{\tau}{}^{*}$

$\overset{l}{\Rightarrow} \overset{def}{=} \Rightarrow \xrightarrow{l} \Rightarrow$

DEFINITION 4.5. *Let $\mathcal{S}$ be a binary relation over $\mathcal{P}$, then $\mathcal{S}$ is a weak simulation if and only if, whenever $P\mathcal{S}Q$,*

if $P \xrightarrow{\tau} P'$ then there exists $Q' \in \mathcal{P}$ such that $Q \Rightarrow Q'$ and $P'\mathcal{S}Q'$.

if $P \xrightarrow{l} P'$ then there exists $Q' \in \mathcal{P}$ such that $Q \overset{l}{\Rightarrow} Q'$ and $P'\mathcal{S}Q'$.

A binary relation $\mathcal{S}$ over $\mathcal{P}$ is said to be a *weak bisimulation* if both $\mathcal{S}$ and its converse are weak simulations. P and Q are *weakly bisimilar*, *weakly equivalent*, or *observation equivalent* (written $P \approx Q$) if there exists a weak bisimulation $\mathcal{S}$ such that $P\mathcal{S}Q$.

We now want to show that our two definitions of heap processes are weakly equivalent. This is captured by the following theorem.

THEOREM 4.5. *If H and H' is well-formed, and $H \equiv H'$, then $P_{GC}(H) \approx P(H')$*

Proof We will show that the following is a weak bisimulation:

$$\mathcal{S} \overset{def}{=} \{(P_{GC}(H), P(H')) \mid H, H' \text{ well-formed} \wedge H \equiv H'\}$$

Since H and H' are well-formed, we show that each transition $\xrightarrow{\tau}$ or $\xrightarrow{l}$ of $P_{GC}(H)$ corresponds a matching transition $\Rightarrow$ or $\overset{l}{\Rightarrow}$, respectively, of $P(H')$ (1), and vice versa (2). We can make the proposition more precise. We already know that $\longrightarrow$ preserves well-formedness (Lemma 4.1). For the two directions we have:

(1) If $H \equiv H'$ and $P_{GC}(H) \xrightarrow{\tau} P_{GC}(H'')$ then $P(H') \Rightarrow P(H')$ (i.e. $\Rightarrow$ is empty). Thus, what we need to show is that if $H \xrightarrow{\tau} H'$ then $H \equiv H'$. On the other hand, if $H \equiv H'$ and $P_{GC}(H) \xrightarrow{l} P_{GC}(\hat{H})$, then $P(H') \overset{l}{\Rightarrow} P(\hat{H}')$. I.e., what we will show is that, if $H \equiv H'$ and $H \xrightarrow{l} \hat{H}$ then $H' \xrightarrow{l} \hat{H}'$ and $\hat{H} \equiv \hat{H}'$.

(2) Since $P(H)$ cannot take τ transitions we only need to show that, if $H \equiv H'$ and $P(H) \xrightarrow{l} P(\hat{H})$ then $P_{GC}(H') \overset{l}{\Rightarrow} P_{GC}(\hat{H}')$ and $\hat{H} \equiv \hat{H}'$. This is similar to what we have in case (1).

Thus, if H and H' are well-formed and $H \equiv H'$:

(i) if $H \xrightarrow{\tau} H''$ then $H'' \equiv H$.

(ii)if $H \xrightarrow{l} \hat{H}$ then $H' \xrightarrow{l} \hat{H}'$ and $\hat{H} \equiv \hat{H}'$. However, from Lemma 4.4 follows that this is true for all possible $\xrightarrow{l}$ transitions.

What remains is to show (i). We proceed by case study on the clauses defining $\xrightarrow{\tau}$.

START,SCANSTART,SCANINT,SCANINT,SCANRESTART, SCANDONE:

Since new paths neither are created nor removed, (i) is true.

DONE:

Since H is well-formed, we know that B does not contain any addresses to W. Thus, since the root is in B, there does not exist any path such that one may reach W. Hence, paths are neither created nor removed, i.e. (i) is true.

FORWARD:

Since $y \mapsto \bullet z$ we have $\forall p \;.\; read(H,y,p) = read(H,z,p)$. Thus, no path is created (nor removed), i.e. (i) is true.

COPYSTART,COPYWORD,COPYRESTART:

Since $\nexists p \;.\; read(H,r,p) = z$, paths are neither created nor removed, i.e. (i) is true.

COPYDONE:

Let $read(H,r,p) = x$, $read(H,r,p') = y$, and $i = |V|$. Then, $join(H,r,i : p,p')$ is true. For H', if $read(H',r,p') = y$ and $read(H',r,i:p) = z$ then since $H'(y) = \bullet z$, $join(H',r,i : p,p')$ is true. Since for any path p'' we have $read(H',y,p'') = read(H',z,p'')$ and $H(y) = H(z) = H'(z)$, read equivalence is upheld by the transition. Thus, (i) is true. ∎

The termination and soundness proofs of the garbage collector are now complete.

5. Usefulness

In this section we show that our algorithm actually performs garbage collection. Definition 5.1 defines, for a root address r, the live part of a heap. We consider a node as live if it is reachable from the root, or if there exists a node containing the locked scan pointer with the address of the node as its index.

DEFINITION 5.1. ***(Live heap).*** *The live portion $L(H)$ of a heap H is defined as:*

$$\begin{aligned} L(H) = \{y \mid\ & \exists p \;.\; read(H,r,p) = y \vee \\ & \exists x, V, V' \;.\; H(x) = \langle V \uparrow_y V' \rangle\} \end{aligned}$$

As the dual of the live portion of the heap we also define the dead portion.

DEFINITION 5.2. ***(Dead heap).*** *The dead portion $D(H)$ of a heap H is defined as:*

$$D(H) = dom(H) - L(H)$$

The first and crucial step towards proving usefulness of the garbage collector is to show that a dead node can never become live again.

LEMMA 5.1. *If $H \longrightarrow^* H'$ then $D(H) \cap L(H') = \emptyset$.*

Proof By induction on the sequence $H \longrightarrow^* H'$.
Base case: $H \longrightarrow^* H$. From Definition 5.1 and 5.2 follows that $L(H) \cap D(H) = \emptyset$. Furthermore, looking at the last transition in the sequence ($\hat{H} \longrightarrow H'$), it is clear that for all cases $L(H') \cap D(\hat{H}) = \emptyset$. ∎

We can now proceed to the concluding theorem (Theorem 5.2), which states that if a node has been copied (i.e. there exists a forwarding node to the new copy), the address of its corresponding forwarding node is not in the dead part of the original heap (i.e. when the garbage collector started).

We begin by defining the set of forwarding nodes (Definition 5.3).

DEFINITION 5.3. ***(Forwarding nodes).*** $F(H) \overset{def}{=} \{x \mid H(x) = \bullet y\}$

THEOREM 5.2. ***(Usefulness).*** *If $H_0 \longrightarrow^* H_n$ forms a GC cycle, then*
$\forall H_i, 0 \le i \le n \;.\; F(H_i) \cap D(H_0) = \emptyset$

Proof By induction on the sequence $H_0 \longrightarrow^* H_i$.
Base case: $F(H_0) \cap D(H_0)$, since $F(H_0) = \emptyset$, $F(H_0) \cap D(H_0) = \emptyset$.
Induction hypothesis: $F(H_{i-1}) \cap D(H_0) = \emptyset$
We proceed by case study on the last transition in the sequence, i.e. $H_{i-1} \longrightarrow H_i$.

COPYDONE:

F is extended with y in this transition, so we need to show that $y \notin D(H_0)$. Since y is reachable from x in H_{i-1} we proceed by looking at x. x can either be a new allocation or $\exists x' \;.\; H_{i-1}(x') = \bullet x$.

For the first case:

Since y has been introduced to x by a mutation it follows from Lemma 5.1 that $y \notin D(H_0)$.

For the latter case:

From the induction hypothesis follows that $x' \notin D(H_0)$. Since y is reachable from x' in H_{i-1}, it follows from Lemma 5.1 that $y \notin D(H_0)$.

For all other cases:

Since $F(H_i) = F(H_{i-1})$ (or $F(H_i) = \emptyset$ for DONE) it follows from the induction hypothesis that $F(H_i) \cap D(H_0) = \emptyset$. ∎

From Theorem 5.2 follows that if a node is dead when the garbage collector starts, it will not get copied. Thus, we can conclude the usefulness property in Corollary 5.3. We first define what we mean by the size of a heap.

DEFINITION 5.4. ***(Size of a heap).*** *The size of a heap is defined as follows:*

$$size(H) = \sum_{o \in rng(H)} |o|$$

where $|o|$ is the number of values in o

(zero for a forwarding node).

Let $live(H) \overset{def}{=} \{x \mapsto o \mid x \in L(H) \text{ and } x \mapsto o \in H\}$.

COROLLARY 5.3. *Suppose $H \longrightarrow^* H'$ is a GC cycle then,*
$size(H') \le size(live(H) \cup \textit{new allocations})$

6. Related Work

Recent work by McCreight et al. introduces a general framework for reasoning about garbage collecting algorithms in Coq, and also includes a mechanized proof of correctness for Baker's copying garbage collector [10]. Central to this approach is a logical specification of the abstract properties a garbage collected heap must expose to its mutator client, a notion somewhat similar to the transition labels we use to model mutator-collector interaction. In contrast to our work, they require the actual algorithm to be expressed in a low level assembly language, and their garbage collector steps must furthermore be externally invoked in a strictly sequential fashion.

Another mechanized study can be found in [13], where Russinoff presents a correctness proof in nqthm of Ben-Ari's incremental mark and sweep collector [2]. He shows that it is safe (i.e. nothing but garbage is collected) and that it will eventually collect all garbage.

In [3], Birkedal et al. prove correctness of Cheney's classical stop-and-copy collector [7] using separation logic with the extension of local reasoning. This elegantly enables them to reason about both the specification and the proof in a manageable way. At the end, they express a promising future of this approach which would enable one to reason about more complex algorithms such as garbage collectors of a different type than stop-and-copy. However, this track of future work has, to the best of our knowledge, not been presented yet.

In [12], Morrisett et al. present a garbage collection calculus and specifies a garbage collection rule based on *free variables*, which models tracing garbage collectors. They present two implementations of it, one that corresponds to a copying collector and one to a generational one. They furthermore show that Milner-style type inference can be used to show that, even though a node is reachable, it can still be garbage, semantically. In [9], Hosoya and Yonezawa extend the idea of using type reconstruction to garbage collect, and they present a garbage collection algorithm based on dynamic type inference. In contrast to tracing garbage collection, where only unreachable garbage is collected, their scheme collects nodes that are semantically garbage.

A huge volume of work on informal descriptions of real-time garbage collection has been presented. One of the earliest incremental copying garbage collectors presented is that of Baker [1]. It is an extension of Cheney's collector into an incremental collector. He utilizes a read-barrier which disallows accesses to fromspace (the white heap). I.e. when the mutator tries to access the white heap, the barrier enforces either a forwarding (if the node has been copied already) or the node to be copied. In the same year (1978) Dijkstra et al. presented the ideas behind the tri-color invariant [8], which has been extensively used by others for reasoning (mostly informally though) about correctness of incremental copying garbage collection. In [5], Brooks presents a variation of Baker's collector. Among other differences, one is that the mutator always follows the indirection via the forwarding pointer, and if the node has not been copied (or the mutator accesses a node in tospace) the forwarding pointer points to the node itself. Instead of checking if the node is in fromspace upon every access, the mutator is always redirected.

7. Conclusion and Future Work

We have presented an incremental copying garbage collector model based on a set of atomic transitions (increments), and we have shown that it is both deterministic and terminating. We have also proved that a mutator-oriented model based on a process calculus with labeled transitions behaves the same with and without the internal garbage collector transitions (soundness). Finally, we have shown that our garbage collector actually does recover unreachable portions of the heap, a property we call *usefulness*.

The next step for future work is to deploy the collector in an actual system. The plan is to do this for a reactive real-time programming language called Timber [6].

Various extensions to the garbage collector is another track for future work. First of all, since the life-time of heap nodes may vary, it is of great interest that the garbage collector is extended with the possibility to handle more than two semispaces, i.e. extending it to a generational collector. The important issues to deal with then is how inter-generational addresses should be handled. The two extremes are: (1) monitoring them at all times (e.g. by a write barrier), and (2) traversing the whole heap but only copying the nodes in the generation that is currently being garbage collected. Another important issue is that of how nodes are selected different generations. This could be done dynamically by a so called promotion scheme, or, if sufficient analysis can be provided, by statically determining to which generation certain allocations should go.

Since our collector only models allocation in the gray heap, short-lived nodes that are allocated during garbage collection will not be collected in the current cycle. If sufficient analysis can be provided, some of the allocations may be done in the white heap instead. I.e., if we know that a certain allocation will not survive after the reaction, we can safely allocate it in the white space. The reason why nodes are by default allocated in the gray heap is because the cost of copying an extra node is higher than just having to scan it. However, to reach a more close to optimal solution, some allocations should be done in the white heap and some in the gray heap. *Region based memory management*, presented by Tofte and Talpin in [16] and an algorithm for deriving those regions, presented by Birkedal and Tofte in [4], seems like a promising technique. The basic idea behind region-based memory management is to find scopes in which sets of dynamically created nodes live and die. We believe that an isolated reaction in a Timber system can be used as a scope, and the nodes that live and die within such a scope can be bundled into a region which we simply may put in the white heap. However, more in-depth study of the inner workings of region based memory management and the corresponding inference algorithm is required in order to assess the actual possibilities in the context of Timber.

Acknowledgements

The authors would like to thank Thomas Hallgren and the anonymous referees for valuable feedback and suggestions.

References

[1] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

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

[3] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of the 31-st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 220–231. ACM Press, January 2004.

[4] Lars Birkedal and Mads Tofte. A Constraint-Based Region Inference Algorithm. *Theoretical Computer Science*, 258:299–392, 2001.

[5] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, 1984.

[6] Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The Semantic Layers of Timber. In *The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. C Springer-Verlag.*, 2003.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

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

[9] Haruo Hosoya and Akinori Yonezawa. Garbage Collection via Dynamic Type Inference - A Formal Treatment. *Lecture Notes in Computer Science*, 1473:215–239, Jan 1998.

[10] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A General Framework for Certifying Garbage Collectors and Their Mutators. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, pages 468–479, San Diego, CA, USA, June 2007.

[11] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[12] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, New York, NY, USA, 1995. ACM Press.

[13] David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[14] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[15] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

[16] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 1997.