2007:56

LICENTIATE THESIS

Garbage Collecting Reactive Real-Time Systems

Martin Kero

Luleå University of Technology
Department of Computer Science and Electrical Engineering
EISLAB

2007:56 | ISSN: 1402-1757 | ISRN: LTU-LIC -- 07⁄56 -- SE

Garbage Collecting Reactive Real-Time Systems

Martin Kero

EISLAB
Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

Supervisors:

Dr. Johan Nordlander
Dr. Per Lindgren

European Union
Structural Funds

To Malin, Eddie, and Jeremiah

Abstract

As real-time systems become more complex, the need for more sophisticated runtime kernel features arises. One such feature that substantially lessens the burden of the programmer is automatic memory management, or *garbage collection*. However, incorporating garbage collection in a real-time kernel is not an easy task. One needs to guarantee, not only that sufficient memory will be reclaimed in order to avoid out of memory errors, but also that the timing properties of the systems real-time tasks are unaffected.

The first step towards such a garbage collector is to define the algorithm in a manageable way. It has to be made incremental in such way that induced pause times are small and bounded (preferably constant). The algorithm should not only be correct, but also provably *useful*. That is, in order to guarantee that sufficient memory is reclaimed each time the garbage collector is invoked, one need to define some measure of usefulness. Furthermore, the garbage collector must also be guaranteed to be schedulable in the system. That is, even though the collector is correct and proved useful, it still has to be able to do its work within the system.

In this thesis, we present a model of an incremental copying garbage collector based on *process terms* in a *labeled transition system*. Each kind of garbage collector step is captured as an internal transition and each kind of external heap access (read, write, and allocate) is captured as a labeled transition. We prove correctness and usefulness of the algorithm.

We also deploy the garbage collector in a real-time system, to wit, the runtime kernel of *Timber*. Timber is a strongly typed, object-oriented, purely reactive, real-time programming language based on *reactive objects*. We show how properties of the language can be used in order to accomplish very efficient and predictable garbage collection.

Contents

Acknowledgements

I remember my senior year as a master student. I was involved in a project course where one of the topics was to implement a garbage collector for Timber. At that time, Timber was just an obscure functional programming language to me. This was the first time I met my current supervisor, Johan Nordlander, who also happens to be the founder of Timber. I remember that my only interest was to get my master's degree and, as fast as possible, begin my career in the industry as a software engineer. However, due to my supervisors Johan Nordlander and Per Lindgren, a growing interest for programming language research emerged. Now I am writing the thesis for my licentiate degree, thinking about those days. Being in this position was not even on the map for me merely three and a half years ago. I am thankful to them for guiding me onto this path. I am also thankful for the tremendous patience they have shown for the, sometimes, very slow pace of my work. One thing that I certainly have learned from them is: *better being correct than being fast*.

Thanks are also due to my fellow colleagues at EISLAB, especially Peter Jonsson, Viktor Leijon, and Andrey Kruglyak. They have been the foundation of the stimulating research environment and with whom I have spent many many hours discussing topics related to our work.

Most importantly though, I would like to thank my wife Malin for her great patience and immovable support; and my boys, Eddie and Jeremiah, for being THE inspiration in my life.

Parts of this thesis are based on previously published research papers. Chapter 3 is based on [24]. Chapter 4 is partly based on [23] and [29].

Chapter 1

Introduction

The task of manually managing dynamic memory gets more difficult as programs become more complex. A correctly designed and implemented garbage collector lessens this burden significantly. Designers and programmers of real-time systems avoid using garbage collectors due to the unpredictable affects they have on the timing behavior of the system. For real-time systems with hard timing requirements, garbage collection has simply not been an option. However, as these systems are designed for more and more complex missions, the burden of manual memory management finally becomes unbearable.

This thesis is dedicated to investigate the challenges and difficulties of making automatic dynamic memory management, or garbage collection, possible for real-time systems. In particular, we present a formal model of an incremental copying garbage collector algorithm. We present full proofs of correctness in terms of termination and preservation. We furthermore implement the algorithm and deploy it in the run-time kernel of Timber, a programming language for reactive real-time systems. Finally, we show how Timber enables both efficient and predictable real-time garbage collection.

1.1 Real-time systems

The definition of a *real-time system* is generally very fuzzy. In order to sufficiently cover the intuition, a complete definition would most probably end up covering most interactive computer systems present today. What we can say, though, is that the difference between a real-time system and a non-real-time system is that real-time systems have explicit timing requirements as a vital part of its specification. Of course, one could argue that most regular non-real-time systems have some limit for what a *reasonable* response time is. However, in that very formulation lies the difference, to wit, the key word *reasonable*. What is a reasonable response time for a certain system without explicit timing requirements? It depends. In contrast, what is a reasonable response time of a real-time system specified to respond within 50 microseconds? Within 50 microseconds, of course.

Even though different people probably gives different answers to the question of what the definition of a real-time system is, we will stick to this definition in rest of this thesis. That is, *a real-time system is a system that has explicit timing requirements as a vital part of its specification.* To make things more complicated, real-time systems are usually divided into two different categories. Firstly, we have the *hard* real-time systems that are considered broken if they fail meeting a deadline. I.e., timing failures are equally bad as computational errors. On the other hand, we have *soft* real-time systems for which the *value* of the response degrades with its lateness. This difference is illustrated in Figure 1.1.

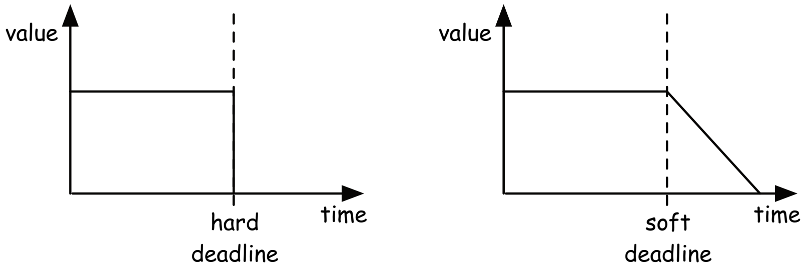

Figure 1.1: Difference between hard and soft real-time systems. Value of the response as a function of time.

We will in this thesis mostly consider hard real-time systems, and whenever we use the term real-time system, we implicitly mean hard real-time systems. Whenever we discuss aspects within a soft real-time system we will explicitly say so.

1.1.1 Modeling real-time systems

In this section we will give a brief overview of the common modeling paradigm used for real-time systems. A real-time systems consists of a set of *tasks* and each task – either independently or codependently with other tasks – has a *mission.* An instance of a task is called a *job.* A task can be classified in three different categories: periodic, sporadic, or aperiodic. Periodic tasks release jobs at a certain pace, whereas sporadic tasks have a minimum inter-arrival time of their job releases. Aperiodic tasks have no minimum inter-arrival time specified, which means that they may arrive arbitrary often. For reasonable causes, realistic task sets never include aperiodic tasks. Attached to each task is its relative *deadline.* An absolute deadline for each job is then calculated according to the release time of that particular job. Commonly, these relative deadlines are translated into static priorities, which simplifies the runtime scheduler. I.e., static priorities are easily stored and compared in runtime.

To capture the worst-case behavior of the system, the task set is simplified in such way that one assumes sporadic tasks to arrive at their worst-case pace. That means, each sporadic task *is* a periodic task with the period set to its minimum inter-arrival time. The execution pattern of a typical real-time system is shown in Figure 1.2. In order to enable a priori schedulability analysis, the worst-case execution times (WCET) of all tasks must be known. This is the main reason to why garbage collection for real-time systems is notoriously difficult. The problem is that a garbage collector commonly affects the WCET of tasks in an unpredictable manner.

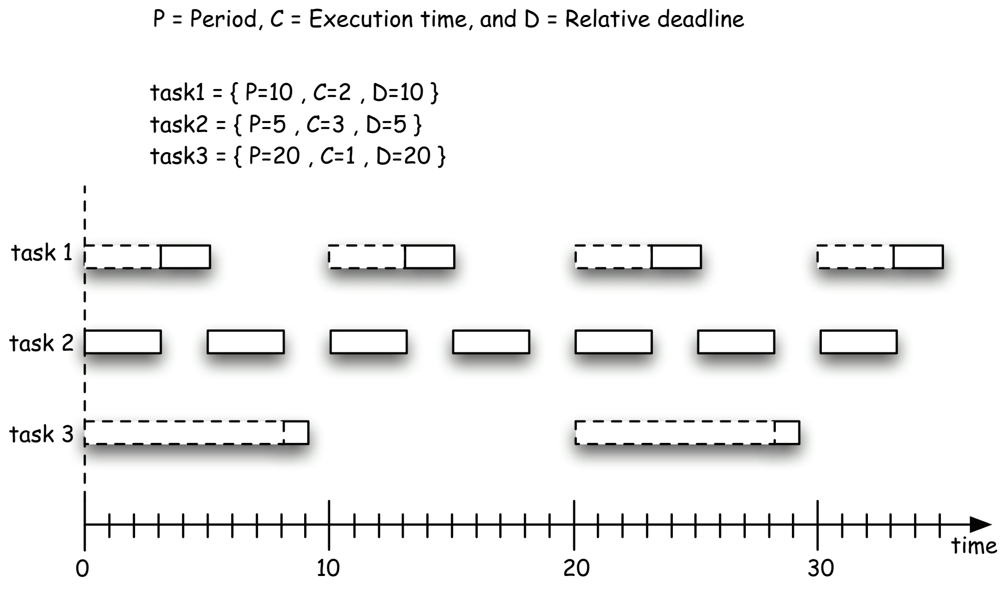
P = Period, C = Execution time, and D = Relative deadline
task1 = { P=10 , C=2 , D=10 }
task2 = { P=5 , C=3 , D=5 }
task3 = { P=20 , C=1 , D=20 }
task 1
task 2
task 3
0
10
20
30
time

Figure 1.2: A typical set up of a real-time system (dashed box indicates pending job).

The *busy periods* of a real-time system are the sequences of job executions without intermediate *idle time*. In the system shown in Figure 1.2, we have busy periods at time 0 to 9, 10 to 18, 20 to 29, and so on. Correspondingly, we have *idle periods* at time 9 to 10, 18 to 20, 29 to 30, and so on. This simple model of a real-time system is then commonly extended to deal with code with critical sections, shared I/O devices, etc. Several scheduling policies for real-time systems exists but it is not really within the scope of this thesis to survey them. Readers interested in how real-time systems are modeled and how schedulability is determined are recommended, among many other resources, to consult the book Real-Time Systems, by Krishna and Shin [27].

1.2 Automatic memory management – garbage collection

One of the most tedious and error-prone tasks of programming fairly complex systems is that of managing dynamic memory. As dynamic multi-threading and object-orientation craves for highly dynamic data storage capabilities, the task of reclaiming and reusing old storage space efficiently is often very detached from the primary purpose of the system. It would be very convenient if such ”low-level” details of where and how dynamic storage is reclaimed could be dealt with automatically behind the scenes. The idea of integrating such a facility in the run-time system of a language's program environment has been around for a long time. Nowadays, with high-level object-oriented languages such as Java and C#, this idea has been spread to the main stream programming community.

1.2.1 Garbage collecting real-time systems

Bringing the idea of garbage collection from main stream programming into the real-time paradigm is the main idea behind this thesis.

Garbage collecting real-time systems has been proved to be a difficult task. The problem is multi-faceted. First of all, in order to make bounded pause times possible one needs an *incremental* garbage collector algorithm. That is, it must be possible to let real-time tasks to interleave with the garbage collector in a bounded and fine-grained manner. In addition, since real-time systems commonly run on platforms with scarce resources, it must also reclaim memory at a sufficient pace. This requires not only a fine-grained incremental garbage collection algorithm, but also a comprehensive knowledge about its inner workings as well as its interaction with the application. Moreover, the memory needs of the application must also be well understood. Thus, real-time garbage collection is certainly more than just an incremental algorithm. However, in order to assess an algorithm and its scheduling possibilities, one needs a thorough understanding of the complexities of its inner workings. Given that, one needs to construct a framework for schedulability analysis of programs that includes the garbage collector.

1.3 Contributions of this thesis

The approach presented in this thesis is to schedule garbage collection only in idle periods. This requires a heap size that is sufficiently large to cope with the worst-case busy period of the system. A priori analysis of memory usage is thus dependent upon the results of schedulability analysis of the system. Nonetheless, this allows schedulability analysis to be performed first, without requiring knowledge about the behavior of the garbage collector or the memory usage of the system. However, in order to a priori determine the memory requirements of the system, one needs a robust model of how the garbage collected memory system behaves in the presence of parallel mutator processes.

In this thesis we present such a model, where we use an incremental copying garbage collector. The model is based on process terms in a *labeled transition system* (LTS), where garbage collector increments are defined as internal transitions and the mutators' interface to the heap is captured by labeled transitions. We reason about the basic properties of the model. We show that the garbage collector eventually terminates and that the meaning of the heap is preserved at all times. These properties constitutes *correctness* of the model. We also show that the model enables a priori memory requirement analysis by reasoning about *usefulness* of the garbage collector in terms of reduced memory occupancy.

We implement the garbage collector and we deploy it in the run-time kernel of Timber, an object-oriented programming language for reactive real-time systems that complies with the mutator interface of our model. We show that hard real-time tasks in Timber do not require any execution time overhead due to garbage collection. I.e., the costs of synchronization (read and write barriers) commonly found in incrementally garbage collected memory systems are zero.

1.4 Outline

The outline of the thesis is as follows:

- The thesis continues with a background study of different garbage collection techniques (Chapter 2).
- Chapter 3 describes a formal model of an incremental copying garbage collector. The model is based on process terms in a *labeled transition system*, where internal transitions captures the garbage collector increments and mutator actions are labeled transitions. We show that our garbage collector eventually terminates and that the meaning of the heap is preserved in the model at all times. We also show that the garbage collector is *useful* in terms of reduced memory occupancy. This chapter is based on a previously published research paper [24].
- In the next chapter (Chapter 4), we survey a real-time system modeling paradigm. More precisely, we give a brief introduction to Timber, a strongly typed, object-oriented, reactive, real-time programming language. We discuss its core construct, namely the *reactive object*. Finally, we describe the run-time kernel of the language.
- In Chapter 5, we describe how our garbage collector can be implemented and deployed in the run-time kernel of Timber. We show how the semantics of the language enables especially efficient and predictable garbage collection. We emphasize the real-time properties of the language and how it affects the requirements on the garbage collector.
- This thesis ends with conclusions and pointers to future work (Chapter 6).

Chapter 2

Garbage Collection

The basic principle of automatic memory reclamation, or *garbage collection*, is the process of automatically reclaiming memory space holding dynamic data that will not be used by the program anymore. The reclaimed memory space is instead made available for future use by the program. In terms of correctness, the garbage collector must not reclaim memory that will be used, nor should it fail to reclaim memory that will not be used. Non-free memory space that will not be used in the future is referred to as *garbage*. When and how garbage is collected depends on the actual algorithm used. We will survey some of the basic techniques used for garbage collection and then look at what makes a garbage collector work for real-time systems.

There exists (at least) two different approaches to garbage collection. On one hand we have the *reference counting* garbage collectors, which means that every dynamically allocated node on the heap holds a counter that keeps track of how many references there exists to that particular node. In other words, at every destructive update and new allocation (with initialization) of heap nodes, the reference counters of the new (and old) immediate descendants must be updated. In addition, references from outside the heap (i.e. from run-time stacks, CPU registers, static/global references, etc.) must also be accounted for. The two biggest disadvantages of this approach are, firstly, the amount of overhead due to incrementing/decrementing reference counters tends to be large for many applications, and secondly, storing aliveness information locally in each node fails to collect cyclic garbage structures. The two biggest advantages of reference counting are that it is quite straightforward to implement and it is naturally incremental (i.e. the work can "easily" be spread out during the execution of the program without inferring too long pause times).

The second approach takes a global perspective on the aliveness property. It is based on a set of root references (run-time stack, CPU registers, static/global references, etc.), from which all live nodes on the heap are *reachable*. The reachability property is an over-approximation of the aliveness property, since a live node is always reachable, while a reachable node might be garbage due to the fact that the program might never use it anymore. Anyhow, the approach is based on traversing the graph of references reachable from the roots, distinguishing the reachable data from the garbage. This approach is called *tracing* garbage collection. How garbage is collected after the reachable nodes has been found depends on which tracing technique that is used. We will look into a few of these techniques in the following sections.

For readers interested in a more comprehensive description of different garbage collection algorithms and their intricate details are recommended to consult the book *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*, by Richard Jones and Rafael Lins [22]; as well as the garbage collection survey by Paul R. Wilson [52].

2.1 The mark-sweep garbage collector

One of the earliest tracing garbage collector is the *mark and sweep* garbage collector [31]. The basic idea of the algorithm is to mark all live (reachable) nodes on the heap by traversing the graph of references beginning in the roots. Then the heap is swept and everything that is unmarked is reclaimed. The advantages of this approach in comparison to reference counting are that it can detect and collect cyclic structures without any special precautions and it does not cause any overhead upon pointer manipulation. The downside is that it is naturally non-incremental; i.e., in order to collect any garbage, it has to run uninterrupted for a relatively long time.

2.1.1 The marking stack

The marking phase of mark-sweep collector is naturally recursive. In order to improve performance, mark-sweep collectors commonly utilize an auxiliary marking stack which holds pointers to nodes that are known to be alive but have not been visited yet. Using a stack for marking naturally leads to a depth-first traversal. The procedure can be described as follows. At the beginning of a garbage collection cycle, the marking stack holds only the root pointer (i.e. it is marked). The procedure then repeatedly pops off the pointer at the top the stack, scans the node for unmarked children and pushes them onto the stack. When the stack is empty, the marking phase is finished.

One issue that is commonly encountered when implementing the marking stack is that of minimizing the depth of the stack to reduce the risk of overflow. As an example of stack depth minimizing actions, we can mention the Boehm-Demers-Weiser mark-sweeping conservative garbage collector for C and C++ [8]. Their marking stack holds pairs of pointers, pointing at the start and the end of the object. In order to reduce the risk of overflow, they examine the object on the top of the stack before they push its descendants onto it. If the object is small (less than 128 words), all its descendants are pushed onto the stack. If the object is big (more than 128 words) only descendants within the first 128 words are pushed onto the stack (marked) and the stack entry is updated so as to correspond to what is left to examine. In order to handle, the less likely, but still possible, stack overflows, nodes are simply not pushed onto the stack whenever the stack is about to overflow (nodes are simply dropped). When the stack eventually becomes empty, the old stack is replaced by a new one of twice the size of the old one. The collector then scans the heap for marked objects with unmarked children and continues the marking phase from there.

2.1.2 Pointer reversal

In this section we will look at a technique for storing marking stack information in-place, while traversing the graph of reachable objects on the heap. The technique of *pointer reversal* was independently developed by Schorr and Waite [42] and Deutsch [26]. The idea is to, while the marking process traverses down the graph, reverse the pointers it follows. I.e., the address of the parent is stored in one of the pointer fields of the object that is currently under examination. As the process retreats after visiting a branch, the pointer fields are restored to their original contents. Thus, the information used for marking is stored in-place rather than in an possibly unbounded marking stack. This was first developed for binary-branched objects [42, 26], but later also extended to variable sized objects, e.g. by Thorelli [48].

The biggest disadvantage of pointer reversal in terms of efficiency is that objects need to be visited many times in order to traverse the graph (at least $n + 1$ times, where n is the number of pointers in the object). On systems with more complex memory architectures, this might be extremely undesirable due to possibly increased number of cache misses and page faults. Due to this, pointer reversal is suggested to be used only as a last resort, e.g., after a detected stack overflow [42]. Still, some systems rely on this technique as their primary strategy [50, 55].

2.1.3 Bitmap marking

In order to determine if an object is marked, the most intuitive solution is to store a *mark bit* in a header field of the object. For many systems, this is not a problem. E.g., some systems store type information in a header field, and the mark bit can probably be squeezed in at some free position in this field without imposing larger overhead. However, for some systems this free space might be impossible to find. A solution is to store the mark bits in a separate bitmap table. The size of the table is inversely proportional to the size of the smallest object that can be allocated on the heap. In order to make it more sophisticated, separate bitmaps can be used for each different kind of object. Accesses can then typically be done via a hash table or search tree. The Boehm-Demers-Weiser collector is an example of where these separate bitmaps are used [8]. In addition, bitmaps can conveniently also be used for the sweep phase as we will see in the next subsection.

The disadvantage of using bitmaps is that mapping an address to a mark bit in the table is generally more expensive than if the information would be stored in the object. Some results indicates that it can cost as much as twelve times as much [58].

2.1.4 The sweep phase

Commonly, the main case made against mark-sweep garbage collection is that it has to sweep the whole heap in order to complete a garbage collection cycle. However, this can be done in parallel with the mutator execution since mark bits, as well as garbage (unmarked) objects, are inaccessible for the mutator. The most common way to interleave the mutator and the sweeper is to do a certain amount of sweeping at each allocation (e.g. [20]). In addition, this approach does not require any explicit *free-list* manipulations since garbage objects are recycled directly to the mutator instead of via a free-list buffer. However, if a bitmap is used, it is more efficient to sweep at least a few words of the bitmap at the same time in order to avoid the need of having to reload and save the bitmap (indexes and bit-masks) at every allocation.

2.2 Fragmentation

Both reference counting and mark sweep garbage collectors suffer from *fragmenting* the heap. This is caused when objects of different sizes are repeatedly allocated and reclaimed. Fragmentation can be a serious problem due to the fact that the free space is not contiguous, and a big enough allocation request may fail even though the total amount of free memory is sufficient. There is, at least, a couple of ways to deal with this problem.

2.2.1 Fixed block size(s)

In order to handle the fragmentation problem the memory can be divided into a set of blocks of fixed size(s). The simplest and easiest way is to use one block size, which is determined by the biggest possible allocation request the application may make. This will of course cause a lot of overhead if the allocation needs of the application vary a lot in size. There are other, more sophisticated, ways to do this, where differently sized blocks may be used. The ideas are based on splitting and coalescing memory blocks for a more accurate match between the block sizes and the actual requests of storage space. As examples we have *segregated free lists* [12], and *buddy systems* [25, 39]. However, even though these systems makes the worst case scenario more unlikely, they do not prohibit it.

Commonly, the problem of fragmentation is divided into two kinds: internal and external fragmentation. External fragmentation is what we normally refer to as just fragmentation; i.e., the fragments of free memory in between objects on the heap. The term internal fragmentation refers to the overhead induced by using fixed block sizes, where some objects might not fit exactly within one block and thus causing more memory to be allocated than needed. An example that segregates objects of different sizes is the two-level allocator of Boehm-Demers-Weiser collector [8]. The low level allocator allocates big chunks of memory (typically four kilobytes) through a standard operating system service (e.g. `malloc`). Each one of these bigger chunks contains only objects of the same size, and a free-list for each object size is maintained. If the free-list is empty for a certain size when a allocation request is made (and the garbage collector is unsuccessful in reclaiming any objects of that particular size), a new chunk for the particular object size is allocated at the lower level. I.e., the free-list for a certain object size may be threaded through several bigger chunks. If a chunk is detected as empty (which is cheaply detected in the Boehm-Demers-Weiser collector) it can be returned to the operating system (e.g. via `free`).

In the worst case scenario, the segregated free-lists of the Boehm-Demers-Weiser collector may cause extreme memory overheads if objects of the same size with long lifetimes are spread out on different chunks. In addition, when new allocations of that size are made, objects with possibly very different lifetime behavior may reside in the same chunk, causing the chunk to never become empty. This, in turn, means that the total amount of allocated memory (at the low level) may be several times more than the actual need. However, in the common case, objects that are active at the same time tends to share the same lifetime behavior. I.e., they are allocated closely, both temporarily and spatially, and also likely to be reclaimed at the same time [16].

2.2.2 Defragmentation

The more robust, but harder, way to deal with fragmentation is to actually defragment the memory by means of moving and compacting live data into a contiguous memory space, which ultimately leads to a contiguous free space. This is of course a more direct way to deal with fragmentation but it comes with a huge load of complications. Defragmenting a big memory may take a long time and allowing the application to run during this process requires a very precautionary synchronization scheme. In addition, uncooperative systems may not function correctly if objects are moved (e.g. C). For these systems, garbage collectors that move live data are simply not an option. In the next sections, we will look at two different approaches to defragmenting the heap.

2.3 The mark-compact garbage collector

The mark-compact garbage collector commonly traverses the live data on the heap many times. The process can be described as consisting of three phases; firstly the live data is marked (similarly to the mark-sweep collector), secondly the graph is compacted by relocating the objects, and thirdly pointers to relocated cells are updated. The way objects are ordered in the resulting heap may differ depending on the actual mark-compact algorithm that is used. There are two ways (three if one counts arbitrary ordering) in which objects commonly are ordered upon compaction. First we have linearizing compaction, which order objects according to the order they appear in the graph (e.g. depth first or breadth first). The second way to order objects is based on the order in which they were allocated; i.e., the live objects are slid together. We will look at two different techniques that are used by mark-compact garbage collectors.

2.3.1 The two-finger algorithms

The first technique we will look at is the two-finger algorithm [4]. It is based on two pointers; one of them points to the next free spot in memory (`free`) and the second to the next live object to be moved (`live`). They start at each end of the heap and work their way towards each other. `free` sweeps the heap for free spots while `live` sweeps from the other end for live objects. Objects discovered by `live` are moved to the empty spots found by `free`. A forwarding pointer is written in the old location of the object. The procedure is finished when the two pointers meet. Then the heap is swept again to update pointers that are pointing to old copies.

The downside with this technique is that a fixed block size is required. However, similarly to the Boehm-Demers-Weiser allocator, objects of different sizes can be segregated into different regions of the heap, and compacted separately.

2.3.2 The forwarding address algorithms

The forwarding address technique reverses the order in which objects are moved and pointers are updated. After the mark phase, new locations of all live objects are calculated and stored in a header field of each object. Then pointers are updated to the new calculated address and, finally, live objects are slid together.

The greatest advantage of this technique is that it can handle differently sized objects with no restriction. On the downside it has to make three passes over the heap, although the amount of work at each pass is quite small. In addition, it requires every object to have an extra field for the forwarding address, although it can be combined with the mark bit.

2.4 The copying garbage collector

The main idea behind the copying garbage collector is to, while traversing the graph, copy reachable objects into a large enough contiguous free space. When all reachable objects have been copied, the old space can then be freed altogether. The resulting heap is both defragmented and garbage collected. The basic steps can be described as follows. The heap is split in two parts; *tospace* is the active one and *fromspace* the inactive one (see example in Figure 2.1). The first thing the copying garbage collector does is to flip the labels so fromspace becomes tospace and vice versa. The objects pointed to by the root-set is then copied and forwarding pointers are installed in their old copies (Figure 2.2 and 2.3). While traversing the graph, if an object is found in fromspace and it is already copied (determined by the existence of a forwarding pointer) the pointer through which it was found is updated with the value of the forwarding address (Figure 2.4). Otherwise the object is copied (Figure 2.5). When the collector has finished traversing the graph the garbage collection cycle is done. The resulting heap is a defragmented and garbage collected version of the previous one (2.6).

The first copying collector, presented in 1963, was the Minsky collector for LISP [34]. It used secondary storage, i.e. a file on disk, as target space for copying. Today this approach would not be very efficient, due to the several orders of magnitude slower file operations in comparison to memory operations. Fenichel's and Yochelson's [15] collector divides the heap into two semispaces where only one of them is active at the same time.

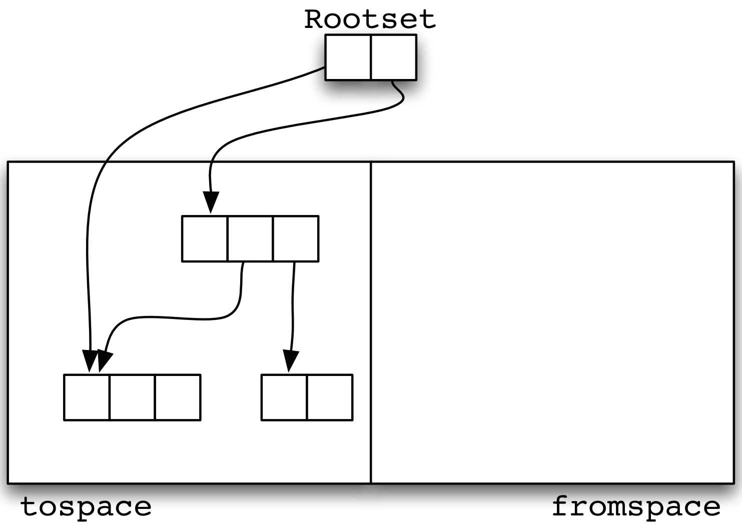

Figure 2.1: Example of a heap set up before a copying garbage collection cycle.

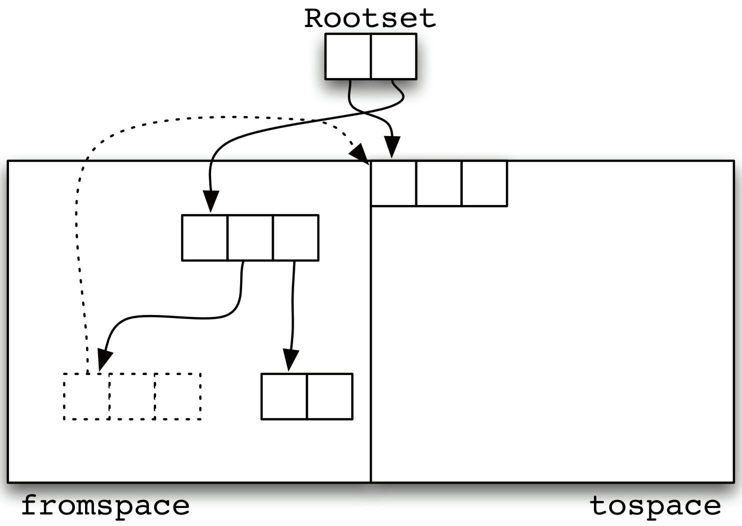

Figure 2.2: Begin by copying the first root.

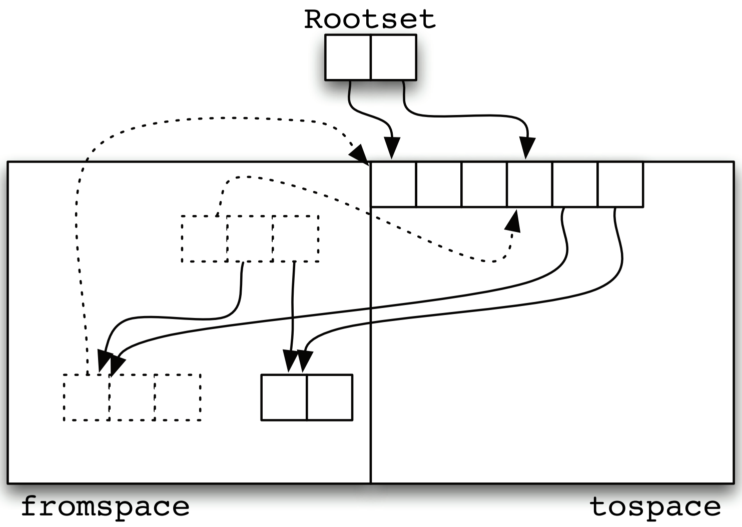

Figure 2.3: Continue with second one.

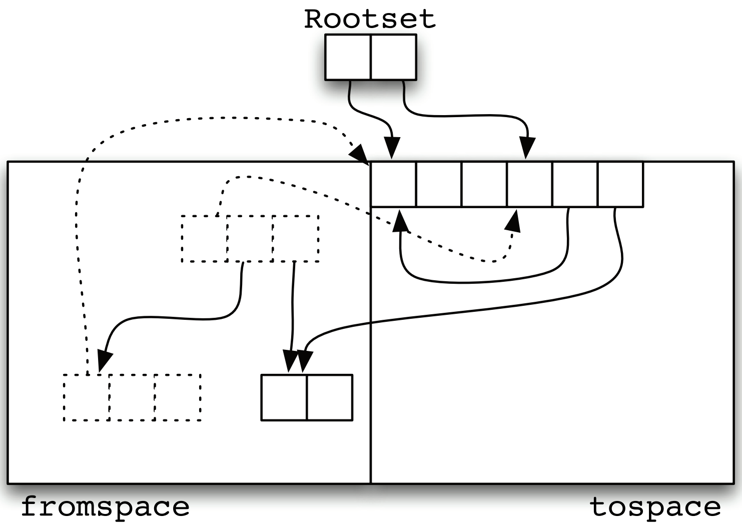

Figure 2.4: Update a pointer with the new (forwarding) address.

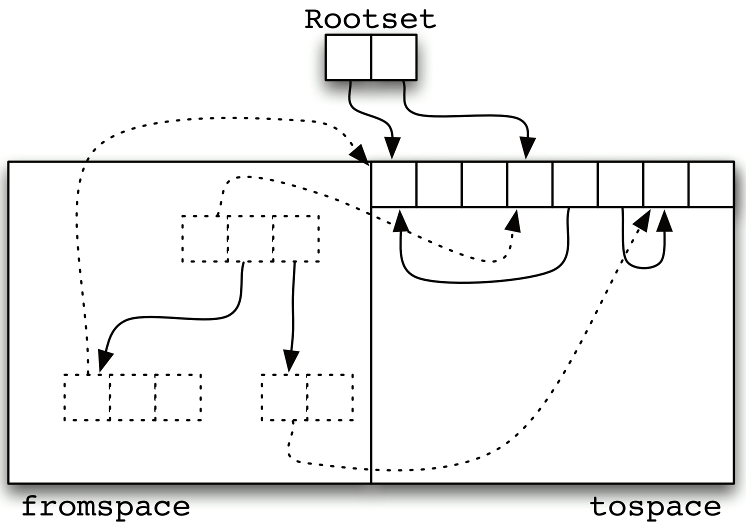

Figure 2.5: Copy encountered objects to tospace.

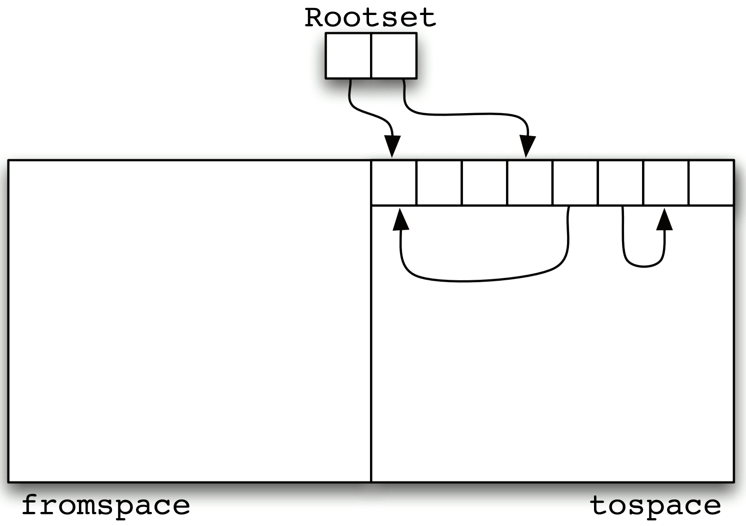

Figure 2.6: When the garbage collection cycle is finished, the resulting heap is a defragmented and garbage collected version of the previous one.

A collection cycle is performed by copying all live blocks from the active space (fromspace) into the inactive space (tospace). Then the labels are switched, i.e. the previously active space becomes inactive and vice versa. To avoid copying the same block more than once, a forwarding pointer is installed in the old copy.

2.4.1 Cheney's copying garbage collector

In [11], Cheney presents a non-recursive copying traversal algorithm using a scavenging scan pointer. It utilizes an in-place breadth first traversal in contrast to the recursive approach of Fenichel's and Yochelson's collector. The basic idea is to scan the copied blocks for references into fromspace, and if such a reference is encountered, copy it into tospace (see Figure 2.7). The process can be described as follows. The algorithm uses two pointers (`scan` and `top`). Initially, both of them points to the beginning of tospace. The algorithm then copies the roots to tospace, which moves `top` accordingly. These objects in between `scan` and `top` are then scanned (from left to right) for pointers into fromspace. When (if) such a pointer is found, the object is copied into tospace (pushing `top`). Thus, while `scan` chases `top` it will push `top` forward as long as there are more objects to copy. The algorithm is finished when `scan` catches up with `top` (i.e., the whole graph has been traversed and copied).

The biggest advantage of the Cheney copying collector is its simplicity. No mark bitmaps or free-lists need to be maintained, and allocation is simply done by pushing a pointer (`top`). Only live data has to be visited (i.e. no sweeping the heap). This is especially good for systems that produce a lot of garbage (e.g. functional programs).

The main drawback of a copying garbage collector is that it requires twice as much of memory than the program actually needs in order to be able to copy all live objects from fromspace into tospace.

In [1], Appel argues that copying garbage collection can be made arbitrary efficient if sufficient space is available. Since the heap is a push-only stack, Appel argues that heap allocation can be made faster than stack allocation. However, this requires the heap size to be several times larger than the average volume of live data.

In addition, the spatial locality issue is still a matter of discussion. However, in contrast to most other tracing garbage collectors, Cheney's copying collector arranges the graph of live objects in a breadth-first manner, in contrast to the forwarding address mark-compact algorithms (Section 2.3.2), which arrange objects in the same order they were allocated.

In [35], Moon presents a modified version of Cheney's collector, which is approximately depth-first. Instead of always moving `scan` towards `top`, Moon uses a third pointer, `scan_partial`, which scans the latest virtual memory page [35] (see Figure 2.8).

The downside of Moon's algorithm is that objects will be scanned twice. However, it is argued that scanning an object one extra time is cheap, since it cannot cause any more copying.

Wilson et al. remove the re-scanning of Moon's algorithm by modifying it into a two-level version of Cheney's original breadth-first traversal [53]. In addition to the two

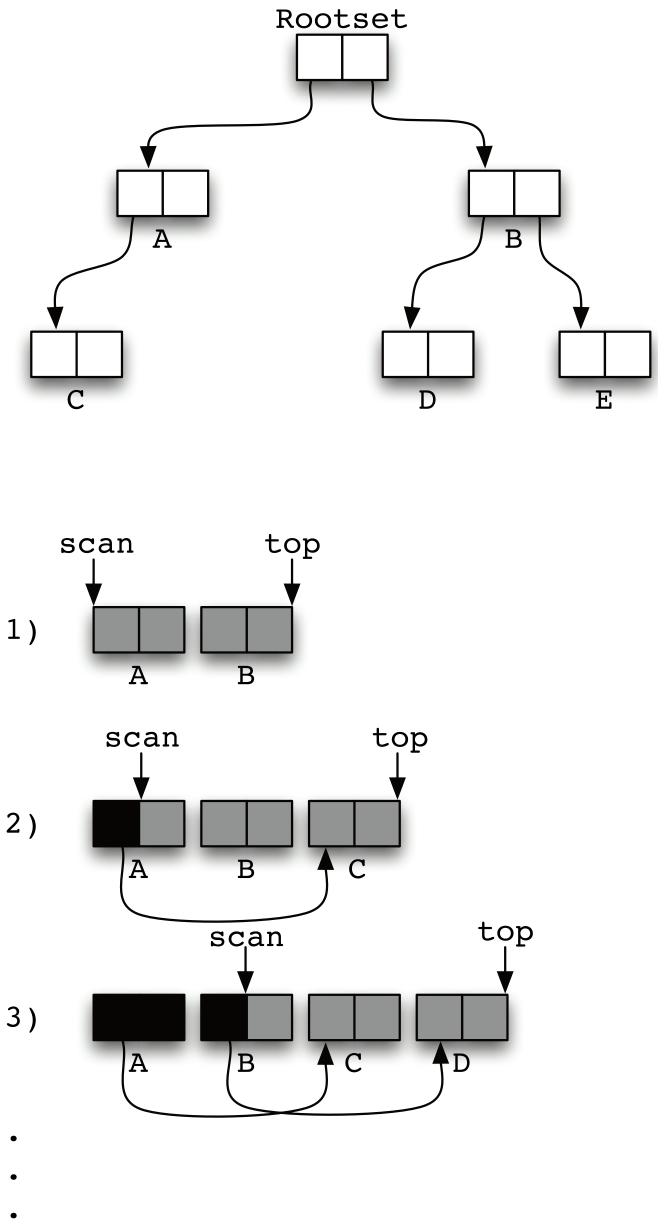

Figure 2.7: The Cheney algorithm's breadth-first traversal

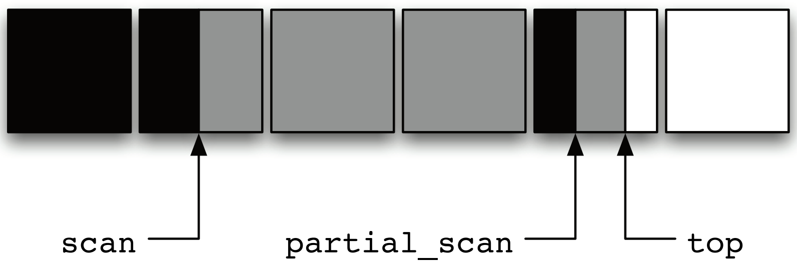

Figure 2.8: Moon's approximately depth-first traversal.

pointers, `scan` and `top`, each page has its own local two pointers. The major scan pointer points at the first incompletely scanned page in tospace. The minor scan pointer of that page scans the page until it reaches the end (or top) of it, then the major scan pointer is moved to the next page. If the minor scan pointer of the first incompletely scanned page causes an object to be copied it triggers a local scan of that page and continues until it is finished. Then the process continues from where it were before. Already scanned pages of tospace is easily detected and skipped by the major scan pointer, since each page has its own local scan pointer. That is, if the local scan pointer is at the end of the page it is already scanned.

2.5 Generational Garbage Collection

A significant drawback of a copying collector scheme is the possibility of copying long-lived blocks back and forth between the spaces when only short-lived blocks are collected. To remedy this problem one could use more than two spaces and separate the blocks between the spaces by their age. Due to the distinction in age, these spaces are commonly referred to as *generations*. Examples of early copying collectors using generations are the Lieberman and Hewitt collector [28] and the Ungar collector [51]. The main difficulty in designing a copying collector based on generations, which was identified even in the early days, is the detection and treatment of inter-generation pointers. To avoid traversing a generation that is not subject for garbage collection, these pointers need to be stored as part of the root set. E.g., the root set for garbage collecting generation A has to include all pointers from generation B to generation A. If these pointers are not known a priori, generation B must be traversed in order to find them.

2.5.1 Promotion schemes

The basic idea of a promotion scheme is to determine when objects should be promoted from a younger generation to an older one. Promoting objects too soon will cause the older generation to be filled with garbage and increase the need for garbage collecting it in a frequency closed to that of the younger one. On the other hand, promoting objects too late causes the generational collector to behave like a regular one. E.g., for a copying collector, long-lived objects would be copied back and forward several times.

The common way to record age is to count how many times an object has been copied. The question then is; how many times does an object need to be copied before it is considered old enough to be promoted? Ungar claims that increasing the threshold beyond two will likely only reduce the number of surviving objects very little [51]. In addition, according to Wilson and Moher [54], one commonly needs to increase the threshold by a factor of four or more in order to reduce the number of survivors by fifty percent.

Counting the number of times an object has been copied is not a pure measure of age. If the size of the younger generation is small, the number of times an object has been copied will result in a younger age than if the size is big. One of the driving forces behind generational (non-incremental though) garbage collection is that smaller pieces of the heap is collected, which in turn means shorter pause times. In addition, for generational copying garbage collectors, pause times are reduced even more since younger generations (with lower survivor rates) are garbage collected more often. Anyhow, increasing the size of the young generation will not only let objects grow older, but also induce longer pause times. Tuning this is a difficult and complex task, which for example is shown in [21].

A downside with generational garbage collectors is that age of objects needs to be recorded somehow. In [43], Shaw suggests that each generation should be divided into regions, called *buckets*. The young generation is divided into *newspace* and *agingspace*. New allocations are made in newspace. Every nth garbage collection cycle, objects in newspace are moved to agingspace and objects in agingspace are promoted to the next generation. Thus, no age is needed to be recorded in the object, instead the age is detected by where in the generation it resides.

Lieberman and Hewitt [28] realized that if the expected lifetime of an object is known when it is allocated, generational garbage collection would be much more efficient. The necessary bookkeeping in order to enable objects to grow old, as well as the work needed to *promote* an object to an older generation, both carry significant overhead.

2.5.2 Inter-generational pointers

Since generational garbage collectors only garbage collect parts of the heap each time (that is after all the main idea of it), pointers into the current part from other parts of the heap must be accounted for. I.e., some objects in the part that we garbage collect might only be alive through pointers from outside that part. We refer to this kind of pointers as inter-generational pointers. Inter-generational pointers have to be accurately recorded and maintained in order to garbage-collect each generation correctly. The simplest and probably most naive way would be to traverse (or even linearly scan) other generations at collection time. However, that would remove the basic idea of generations, which, all in all, aims at reducing the amount of objects to touch. Even though this technique has been shown to improve performance in comparison to the basic two-semispace copying collector [47, 43], there are more precise and delicate ways to deal with this kind of pointers. We will, for the sake of simplicity, from now on only consider pointers from older to younger generations, since most generational garbage collectors either collect only the younger generation (which happens often) or collect the whole heap (which happens more seldom). Thus, pointers from younger to older generations do not cause any problems.

2.5.3 The write-barrier

There are two ways inter-generational pointers are created; either through a destructive update of a pointer field in an object in the older generation, or the pointers simply reside in a young object that is promoted to an older generation. The second case is easily detected by the garbage collector. I.e., every time an object is promoted to an older generation the object is scanned for pointers to the younger generations.

The problem lies in the first case, the mutator updating pointer fields of objects in older generations. This is commonly soled by means of a *write-barrier*, which traps writes to heap objects, checks if the written value is a pointer to a younger generation, and if so, takes proper action. As examples of this write-barrier we can mention *entry tables* used by Lieberman and Hewitt [28] and *remembered sets* by Ungar [51]. The basic idea behind entry tables is that pointers from an older generation into a younger one has to go through an indirecting table that is stored in the younger generation. Thus, when the garbage collector runs it only has to use the entry table as part of the root set in order to capture the inter-generational pointers. The idea behind remembered sets is that when the mutator writes an old-to-young pointer, a pointer to the old object is stored in a remembered set. When the garbage collector runs, it scans the objects stored in the remembered set.

2.6 Incremental Garbage Collection

One of the major motivations behind generational garbage collection was to reduce the imposed pause times due to garbage collection. Since even generational garbage collectors occasionally will collect the whole heap, they still have the same worst-case scenario as their non-generational dittos. In order to reduce the worst-case pause time it was early discovered that it would be desirable to let mutators interleave the garbage collecting process in a pseudo-concurrent manner. However, this gives rise to a need for rigorous synchronization schemes between the memory management system and its mutators.

2.6.1 Coherence, consistency, and conservatism

In general, the coherence problem is basically due to multiple processes sharing mutable data. Preserving coherence in an incrementally garbage collected system generally boils down to actions to *notify* the collector about mutations made by the mutators. Incremental copying collectors generally suffer from an even worse coherence problem, as the mutators must also account for mutations (re-locations) made by the collector. This situation is commonly referred to as a *multiple readers, multiple writers* problem. However, the required protection is not that severe. The reason is that collector and mutators do not necessarily need to have a consistent view of the heap. For instance, the garbage collector may very well, for a limited period of time, treat some dead objects as being alive (which only results in some unnecessary occupied memory), as long as it never treats live objects as being dead (which could have severe consequences).

All garbage collectors are more or less *conservative*, in such way that some dead objects may be treated as alive for a limited time because the cost, at a certain moment, of determining if it really is dead is too high.

2.6.2 The tricolor invariant

Dijkstra et. al. [13] present the *tricolor marking* abstraction. This is useful in understanding how garbage collection can be made incremental. During a garbage collection cycle, an object in the heap can be in three different states. It can be undetected (possibly garbage), detected (alive), or processed. The tricoloring marking is based upon the idea of painting objects in different colors. An undetected object is colored white, detected objects are colored gray, and scanned objects are black. At the beginning of a collection cycle, all objects are white. When the collector traverses the graph, it will color white objects gray and gray objects black. The order in which this is done is dependent on the type of tracing collector that is used, but for an incremental copying collector, it is suitable to interleave the detection and copying in a *breadth-first* manner; that is, before a gray object is colored black, all its white descendants are colored gray. This will lead to a criterion that is the essence of the *tricolor invariant*; to wit, a black object can never point to a white object. It always has to be a gray object in between them. A violation of this invariant may cause an object that is alive to be reclaimed as garbage.

In Figure 2.9 and 2.10, we see an example of how mutations may cause violations of the tricolor invariant. Figure 2.9 shows the heap before the mutation, and in Figure 2.10 we see the heap after the mutation. The collector thinks that since A is already scanned (i.e., it is black) all its descendants have been detected. Thus, D will be collected even though it is still alive.

The solution is to either color D gray (Figure 2.11), or revert the color of A back to gray (Figure 2.12), when A is mutated. Either way, the tricolor invariant is preserved.

2.6.3 Conservatism and allocation

Another choice needed to be made is where new allocations should reside. One could allocate them as white, which means that new allocations are able to die and be collected within the same garbage collection cycle they were allocated. This is the least conservative allocation strategy. New allocations that survive, must be marked/copied and scanned. Allocating new object gray means that the collector does not need to mark/copy newly allocated objects but still needs to scan them. The last possibility

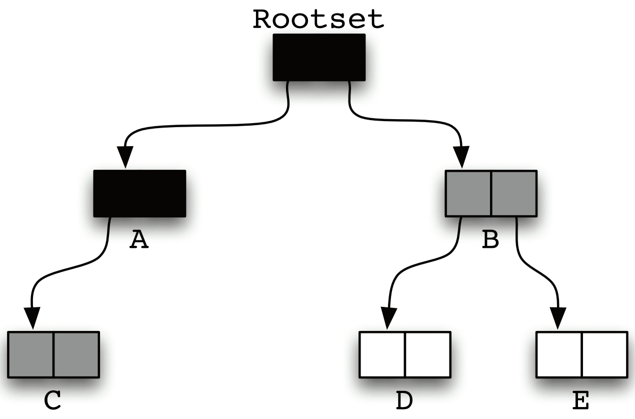

Figure 2.9: The tri-color invariant is upheld during garbage collection.

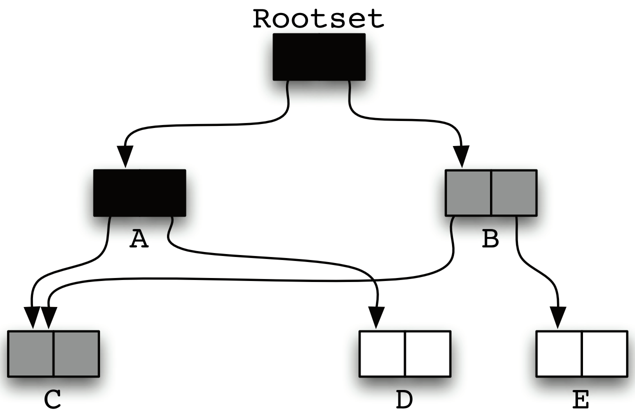

Figure 2.10: The mutator introduces a violation of the tri-color invariant.

is allocating new objects black. This means that allocations made during garbage collection does not impose any more work needed to be done by the collector. However, initialization of these objects must ensure no black to white pointers are created.

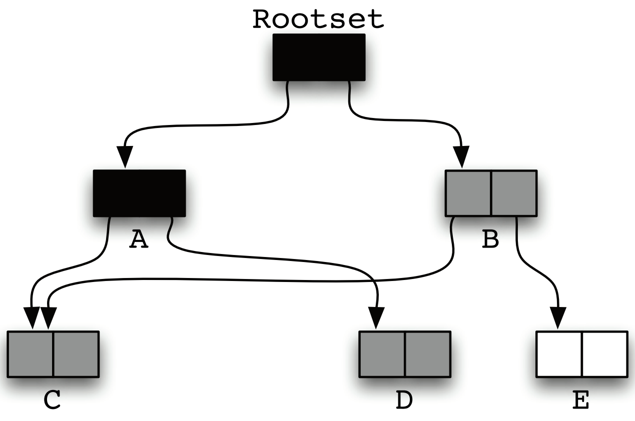

Figure 2.11: The tri-color invariant is preserved by coloring the white object (D) gray.

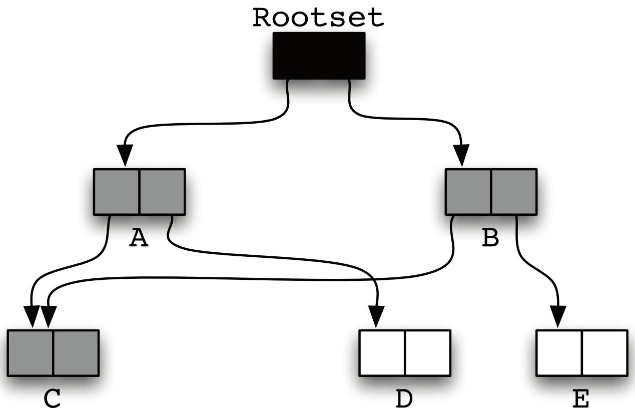

Figure 2.12: The tri-color invariant is preserved by coloring the black object (A) gray.

2.6.4 Barrier methods

Preserving the tricolor invariant is done through so called *barriers*. We will look at these barrier methods in the context of an incremental copying garbage collector. We can prohibit the mutator from accessing fromspace (white) objects by using a *read barrier* that, whenever the mutator tries to access a white object, the collector interleaves to copy the object (color it gray), as shown in Figure 2.11. Alternatively, whenever the mutator writes a white pointer into a black object, the color of the mutated object is reverted back to gray.

The read barrier

Baker's collector [2] is an extension of Cheney's collector. It is made incremental with a *tospace invariant*; that is, the mutator can only *see* objects in tospace. That means, creating or dereferencing a pointer to a white object causes the object to be copied.

This is accomplished by a read barrier that checks if the mutator dereferences a pointer to a white object, and a write barrier that checks if the written value is a pointer to a white object. If the mutator dereferences a pointer to a white object, the object is copied first.

The intrinsic tricolor marking in Cheney's collector is shown in Figure 2.7, where everything behind the scan pointer is black and everything in front of it is gray. In Baker's algorithm, new allocations during garbage collection are made at the other end of tospace and considered black (see Figure 2.13).

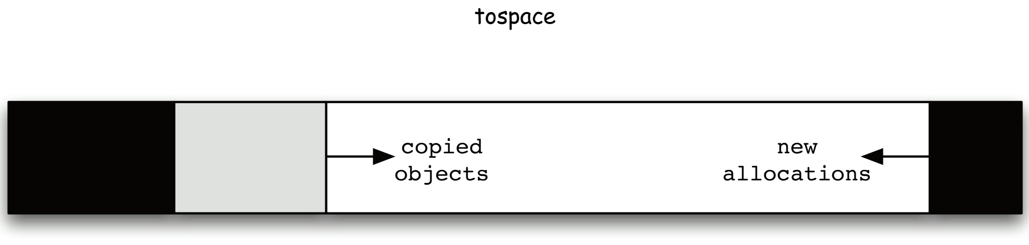

Figure 2.13: New allocations during garbage collection are made at the other end of tospace and considered black in Baker's incremental copying garbage collector.

In contrast to Baker's choice of allocating objects during garbage collection as black, one could allocate them as either gray or white. However, since Baker's collector is based on a tospace invariant, allocating new objects as white would not make any sense. Furthermore, since these newly allocated objects cannot contain pointers to white objects (i.e., the mutator is not allowed to see white objects and thus cannot write pointers to white objects into the new object), allocating new objects as gray would just cause unnecessary scanning.

Brooks [9] extends Baker's collector by always initializing the forwarding pointer field of a new allocation to point to the new object itself. Accesses can then always be indirected through the forwarding pointer by the read barrier; instead of checking if a forwarding pointer exists.

Although enforcing the tricolor invariant through a read barrier has its advantages, the main drawback of read barriers is its inefficiency. Zorn's evaluation of different barrier methods shows that read barriers are expensive on conventional machines, mainly because reads tend to occur very frequently in comparison to writes [57].

The write barrier

There is one way an incremental garbage collector can behave correctly without preserving the tricolor invariant. Yuasa's *snapshot-at-the-beginning* incremental mark-sweep garbage collector does not prohibit black to white pointers [56]. Instead, the write barrier traps pointer updates and marks the object pointed to previously if the object is white. In effect, it preserves the aliveness of the heap as it was when the garbage collector started (hence the name snapshot-at-the-beginning). New allocations during garbage collection are made non-white. Thus, Yuasa's collector will not fail to mark all live objects. However, on the other hand, Yuasa's collector is very conservative since white objects that die during garbage collection will not be collected before the next cycle.

A more common technique for ensuring that all live objects will be marked/copied is the *incremental update* methods, which are not as conservative as Yuasa's algorithm. They all preserve the tricolor invariant (no black to white pointers) through the write-barrier which, depending on the algorithm, colors one of the objects involved gray.

The garbage collector presented by Dijkstra et al. [13] is an example of the more conservative incremental update techniques. It colors a white object gray regardless of the color of its new parent. E.g., when a white to white pointer is written, the pointee is colored gray. A less conservative technique is the one due to Steele [44, 45], which reverts the color of a black object back to gray upon mutation.

2.6.5 Distinguishing mutable and immutable data

Doligez and Leroy [14], as well as Huelsbergen and Larus [19], identify the advantage of distinguishing mutable and immutable objects from a garbage collectors perspective. This is especially an advantage for a copying collector, due to its consistency issues. Basically, in terms of consistency, mutable and immutable data differ in such way that access to two different copies of the same immutable data structure does not result in incoherence, whereas mutator access to multiple copies of a mutable data structure does.

2.7 Real-Time garbage collection

A *real-time garbage collector* is a garbage collector that (1) does not cause any real-time task to miss its deadline at the same time as it (2) reclaims garbage memory in a sufficient pace. Both of these requirements originate in the schedulability problem; i.e., is the system schedulable or not? However, in order to make such assessments, one needs to concretize what there is to schedule. Commonly, scheduling garbage collection is often reasoned abut in terms of induced pause times. Since non-incremental garbage collectors induce too long pause times, they are not suitable for real-time systems. Reducing the worst-case pause time means reducing the length of the longest (worst-case) garbage collector increment. One thing is certain, the execution time of each increment of the garbage collector must be small and bounded (preferably constant).

2.7.1 Scheduling garbage collection

The problem of scheduling the garbage collector increments within a real-time system is two-dimensional. First of all, one needs be able to guarantee that the timing requirements of the real-time tasks are not violated. At the same time the garbage collector must also be scheduled in such way that sufficient free memory is available at all times. The ultimate question is then, is there a schedule where both of these requirements are fulfilled?

2.7.1.1 Scheduling policies

Finding an optimal scheduling policy (i.e. be able to schedule every system for which there exists a schedule) is probably practically impossible due to the NP nature of its corresponding decision problem. However, from a pragmatic point of view, it is still good if we may schedule a big subset of them.

In his thesis, Henriksson presents a policy for scheduling garbage collection in embedded systems with both soft and hard real-time tasks [17]. The main idea of his policy is that the garbage collector should never interfere with the hard real-time tasks. However, in order to sufficiently reclaim garbage produced by the hard real-time tasks, necessary garbage collection work due to their actions has to be scheduled with an higher priority than the soft real-time tasks. In Figure 2.14, a typical example of Henriksson's schedule is shown.

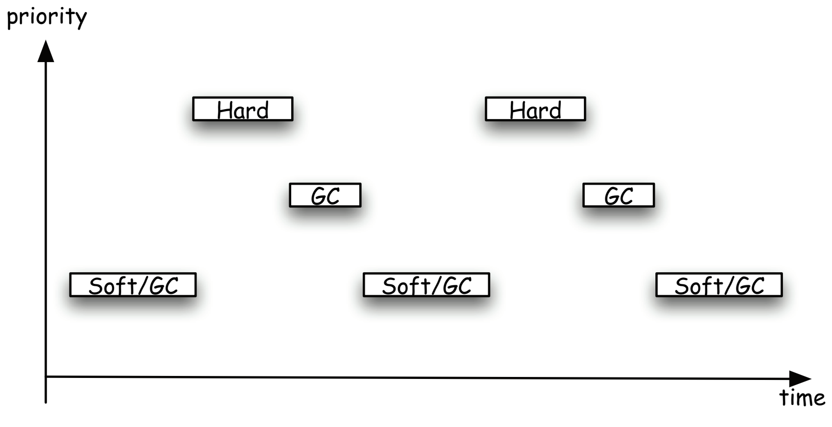

Figure 2.14: It typical schedule of a system based on Henriksson's scheduling policy.

2.7.2 Predicting garbage collection pause times

Before we go into details about how pause times due to garbage collection can be predicted, we need to decide how the garbage collector may be scheduled in the system. We assume a concurrent garbage collector with sporadic behavior. That means, even though the garbage collector has a lower priority than the hard real-time tasks, it may be running when a hard real-time job is released.

There are two kinds of induced pause times due to garbage collection. The first kind is due to executing a garbage collection increment. That is, if a real-time job is released while a garbage collector increment is being executed, the job has to be postponed until the increment has finished. The WCET of real-time tasks must include the longest WCET of the garbage collector increments. The second kind is due to the read barrier and the write barrier. The WCET of heap reads and writes has to include the WCET of the corresponding barrier methods.

2.7.3 Guaranteeing sufficient memory reclamation

Guaranteeing that enough memory is reclaimed at all times is a more difficult task than predicting the pause times due to garbage collection. Let us first look at a system with only hard real-time tasks[1]. We assume that the only pause time the garbage collector may induce is as defined in the previous subsection. In other words, the garbage collector increments may only be started when no hard real-time tasks are pending. The question is then: is there enough idle time and is it distributed in such way that garbage memory can be reclaimed sufficiently fast? In order to assess this, we need to know the memory consumption behavior of each task.

Real-time garbage collection is more than just an algorithm; it includes schedulability, memory requirements, etc.. An incremental garbage collection algorithm is just a good starting point. The main requirements on the algorithm is typically as follows. Bounded pause times are achieved by predictable worst-case execution time of all increments. Worst-case memory usage overhead is derivable from the algorithm. And finally, the most important aspect: the collector must be possible to schedule in the real-time system. It should neither cause violations of timing constraints, nor should it cause the system to consume more memory than available.

2.7.4 Predictability and efficiency of copying garbage collection

If we would be able to determine the amount of live data at any given point in time, the copying garbage collector would be almost hundred percent predictable. The execution time of the collector is directly proportional to the amount of data that currently is alive. Due to the fact that the copying garbage collector is actually a defragmentation process and garbage collection is a side-effect, it would be unfair to evaluate its efficiency only in terms of collected garbage. The copying collector suffers from inefficiency when it

[1]It can be argued that it is always possible to suspend a soft real-time task in order to do sufficient garbage collection.

performs unnecessary defragmentation. In the worst case, it would copy half the heap (whole fromspace), accomplishing neither defragmentation nor garbage collection.

On the other hand, when the amount of live data is small and there is a lot of garbage produced, the collector would be very efficient. In the best case, it would collect half a heap worth of garbage almost without doing anything at all, at least not in terms of copying work.

2.8 Design choices

In his thesis, Ritzau presents a real-time reference counting garbage collector where cyclic structures have to be broken manually [40]. Even though reference counting is naturally incremental, the lack of native support for collecting circular data structures removes the technique from our list of choices. Furthermore, the behavior of non-compacting algorithms is generally very unpredictable in terms of free memory availability due to fragmentation. Sophisticated methods exist for solving this problem, but they commonly induce too much overhead on the allocator. So, we need an algorithm that is incremental, compacts the heap, and enables small and predictable allocation time. In addition, it should be able to run together with an arbitrary number of mutator threads; i.e., it should be possible to integrate into a typical real-time model as described in Section 1.1.1. Most importantly, the algorithm should enable decoupling of the WCET of the real-time tasks and the garbage collector. That is, assuming that enough free memory is available at all times, it should be possible to reason about the schedulability of the system without knowing anything about the garbage collector. Furthermore, adding garbage collection should not invalidate that analysis. In other words, the algorithm should enable incremental analyses without mutual dependencies.

Based on the above mentioned reasons, we have chosen an incremental copying garbage collector which are scheduled to only run in idle periods. In the next chapter we describe the memory model including the garbage collector, which enables reasoning about the garbage collector in the context of parallel mutator processes. In Chapter 5, we describe how scheduling garbage collection in idle periods opens up possibilities for incremental reasoning about the real-time capabilities of the whole system, including garbage collection.

Chapter 3

A Correct and Useful Incremental Copying Garbage Collector

This chapter is based on a previously published research paper [24].

In this chapter we present a formal model of an incremental copying garbage collector (Section 3.1). 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 3.2 and 3.3). We furthermore show that our garbage collector is useful, in the sense that it actually recovers unreachable portions of a heap (Section 3.4).

Apart from taking the first step towards the overall goal of designing a formally proved 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 contribution of this chapter is that we 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 contribution is that formal correctness of incremental copying garbage collection has not been shown until very recently [32]. A third contribution is that we demonstrate that it is also possible to formally reason about the usefulness of an incremental copying garbage collector.

3.1 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 [11], and we deploy a read barrier similar to that of Brooks [9]; 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 [44, 45], 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 defer 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 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* [13].

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 3.1 and 3.2, 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.

$$H_0 \xrightarrow{\textsc{Start}} H_1 \longrightarrow \ldots \longrightarrow H_{n-1} \xrightarrow{\textsc{Done}} H_n$$

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 labeled 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.

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 3.1: Scan transitions.

3.2 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 3.2.1

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

Definition 3.2.2 *For some heap H and root x,*

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

Forward	$\begin{array}{c} W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B \\ \longrightarrow \\ W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V, z \downarrow V' \rangle \mid B \end{array}$	
CopyStart	$\begin{array}{c} W, y^{\dagger} \mapsto \langle U \rangle, W' \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B \\ \longrightarrow \\ W, y \mapsto \langle U \rangle, W' \mid z \mapsto \langle\rangle, G, x \mapsto \langle V \uparrow_z y, V' \rangle \mid B \end{array}$	†may be dirty z is fresh
CopyWord	$\begin{array}{c} W, y \mapsto \langle U, u, U' \rangle, W' \mid G, z \mapsto \langle U \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B \\ \longrightarrow \\ W, y \mapsto \langle U, u, U' \rangle, W' \mid G, z \mapsto \langle U, u \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B \end{array}$	†may be dirty
CopyRestart	$\begin{array}{c} W, \dot{y} \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle U' \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B \\ \longrightarrow \\ W, y \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle\rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B \end{array}$	†may be dirty
CopyDone	$\begin{array}{c} W, y \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle U \rangle, G', x^{\dagger} \mapsto \langle V \uparrow_z y, V' \rangle \mid B \\ \longrightarrow \\ W, y \mapsto \bullet z, W' \mid G, z \mapsto \langle U \rangle, G', x^{\dagger} \mapsto \langle V, z \downarrow V' \rangle \mid B \end{array}$	†may be dirty

Figure 3.2: Copy transitions.

The mutator activities can now be defined as a set of labeled transitions $H \xrightarrow{l} H'$ (see Figure 3.3) where the label l captures the different means a mutator can interact with a heap.

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

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 3.3.

Two important details of Figure 3.3 require special mentioning. Firstly, clauses Mut/AllocMut mark the mutated 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.

MutW	$\begin{array}{c} H = W, x \mapsto \langle U \rangle, W' \mid G \mid B \\ \overset{w(p:i=q)}{\longrightarrow} \\ H' = W, \dot{x} \mapsto \langle U[i] := y \rangle, W' \mid G \mid B \end{array}$	if $read(H, r, p) = x$ and $read(H, r, q) = y$
MutG	$\begin{array}{c} H = W \mid G, x \mapsto \langle U \rangle, G' \mid B \\ \overset{w(p:i=q)}{\longrightarrow} \\ H' = W \mid G, \dot{x} \mapsto \langle U[i] := y \rangle, G' \mid B \end{array}$	if $read(H, r, p) = x$ and $read(H, r, q) = y$
MutB	$\begin{array}{c} H = W \mid G \mid B, x \mapsto \langle U \rangle, B' \\ \overset{w(p:i=q)}{\longrightarrow} \\ H' = W \mid \dot{x} \mapsto \langle U[i] = y \rangle, G \mid B, B' \end{array}$	if $read(H, r, p) = x$ and $read(H, r, q) = y$
Write	$H, x \mapsto \langle U \rangle, H' \overset{w(p:i=n)}{\longrightarrow} H, x \mapsto \langle U[i] := n \rangle, H'$	if $read(H, r, p) = x$
Read	$H \overset{r(p=n)}{\longrightarrow} H$	if $read(H, r, p) = n$
AllocMutW	$\begin{array}{c} H = W, x \mapsto \langle U \rangle, W' \mid G \mid B \\ \overset{a(p:i)}{\longrightarrow} \\ H' = W, \dot{x} \mapsto \langle U[i] := z \rangle, W' \mid z \mapsto \langle\rangle, G \mid B \end{array}$	if $read(H, r, p) = x$
AllocMutG	$\begin{array}{c} H = W \mid G, x \mapsto \langle U \rangle, G' \mid B \\ \overset{a(p:i)}{\longrightarrow} \\ H' = W \mid z \mapsto \langle\rangle, G, \dot{x} \mapsto \langle U[i] := z \rangle, G' \mid B \end{array}$	if $read(H, r, p) = x$
AllocMutB	$\begin{array}{c} H = W \mid G \mid B, x \mapsto \langle U \rangle, B' \\ \overset{a(p:i)}{\longrightarrow} \\ H' = W \mid \dot{x} \mapsto \langle U[i] := z \rangle, z \mapsto \langle\rangle, G \mid B, B' \end{array}$	if $read(H, r, p) = x$

Figure 3.3: Mutator transitions.

3.3 Correctness of the Garbage Collector

The tri-color invariant [13] is captured by a property we denote as *well-formedness*.

Definition 3.3.1 ***(Well-formedness).*** *A heap $H = W|G|B$ is* well-formed *iff*

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

Informally, a heap is well-formed if and only if no black nodes contain addresses to any white nodes (1), a forwarding 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 3.3.1 states that a garbage collection transition of the heap preserves well-formedness.

Lemma 3.3.1 *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 vacuously.

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 $y \in \textit{dom}(W)$ 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 3.3.1 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 vacuously.

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 vacuously.

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 vacuously. ∎

Lemma 3.3.2 captures the property of determinism.

Lemma 3.3.2 *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 3.3.1 (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 3.3.1 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. ∎

3.3.1 Termination

Theorem 3.3.3 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 3.3.3 ***(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 3.4.

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

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

```
$w'(H) \overset{def}{=}$  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 3.4: Weight of a heap

two heaps H and H', and $\mathit{weight}(H) = (a, b, c)$ and $\mathit{weight}(H') = (d, e, f)$. Then,

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

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 $\infty > |V|$.

ScanInt,ScanAddr:

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

$\Leftarrow$

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

$\Leftarrow$

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:

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

$\Leftarrow$

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

$\Leftarrow$

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 3.3.1 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 $w(y) > w(z)$. ∎

3.3.2 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 3.3.2

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

We can now define the equivalence relation between heaps.

Definition 3.3.3 *Two heaps H and H' are structurally equivalent, written $H \equiv H'$, if and only if,*
H and H' are both well-formed, and
$\forall p \,.\, \mathit{read}(H,r,p) = n \Longleftrightarrow \mathit{read}(H',r,p) = n$
and $(\forall q \,.\, \mathit{join}(H,r,p,q) \Longleftrightarrow \mathit{join}(H',r,p,q))$

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

Lemma 3.3.4 *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.

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

From the definition of $H \equiv H'$ (Def. 3.3.3) follows that:
$\mathit{join}(H,r,p,p) = \mathit{True}$ (i.e. x) $\Longleftrightarrow \mathit{join}(H',r,p,p) = \mathit{True}$ (let it be x'). (I)
$\mathit{join}(H,r,q,q) = \mathit{True}$ (i.e. y) $\Longleftrightarrow \mathit{join}(H',r,q,q) = \mathit{True}$ (let it be y').
(II) $\mathit{join}(H,r,p:i,p:i) \Longleftrightarrow \mathit{join}(H',r,p:i,p:i)$ (III)
$\mathit{read}(H,r,p:i) = n \Longleftrightarrow \mathit{read}(H',r,p:i) = n$ (IV)
$\forall p', q'.$

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

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:
$\forall p'$.

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

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:
$join(\hat{H}, r, \varepsilon, q'_1) \Longleftrightarrow join(\hat{H}', r, \varepsilon, q'_1)$
$join(\hat{H}, r, z, \varepsilon, q'_2) \Longleftrightarrow join(\hat{H}, r, z, \varepsilon, q'_2)$
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
$join(\hat{H}, r, \varepsilon, q') \Longleftrightarrow join(\hat{H}, r, \varepsilon, q')$
$j \neq i$:From the induction hypothesis and (VII) follows that
$join(\hat{H}, r, \varepsilon, q') \Longleftrightarrow join(\hat{H}, r, \varepsilon, q')$
$read(\hat{H}, r, q'_1) \neq x$: ($read(\hat{H}', r, q'_1) \neq x'$)
It follows directly from the induction hypothesis that
$join(\hat{H}, r, \varepsilon, q') \Longleftrightarrow join(\hat{H}, r, \varepsilon, q')$

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:
$read(\hat{H}, r, p'_1) = n \Longleftrightarrow read(\hat{H}', r, p'_1) = n \wedge$
$(\forall q'.join(\hat{H}, r, p'_1, q') \Longleftrightarrow join(\hat{H}', r, p'_1, q')) \wedge$
$read(\hat{H}, z, p'_2) = n \Longleftrightarrow read(\hat{H}', z', p'_2) = n \wedge$
$(\forall q'.join(\hat{H}, z, r, p'_2, q') \Longleftrightarrow join(\hat{H}', z', r, p'_2, q'))$
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
$read(\hat{H}, r, p') = n \Longleftrightarrow read(\hat{H}', r, p') = n$
(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.

∎

3.3.2.1 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 [33]. 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.

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

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

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

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 [33].

Definition 3.3.4 *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 3.3.5 *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 3.3.5 *If* $H \equiv H'$*, then* $P_{GC}(H) \approx P(H')$

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

$$\mathcal{S} \overset{def}{=} \{(P_{GC}(H), P(H')) \mid H \equiv H'\}$$

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 3.3.1). 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 3.3.4 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 $\forall p \,.\, \mathit{read}(H, y, p) = \mathit{read}(H, z, p)$. Thus, no path is created (nor removed), i.e. (i) is true.

CopyStart,CopyWord,CopyRestart:

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

CopyDone:

Let $\mathit{read}(H, r, p) = x$, $\mathit{read}(H, r, p') = y$, and $i = |V|$. Then, $\mathit{join}(H, r, i : p, p')$ is true. For H', if $\mathit{read}(H', r, p') = y$ and $\mathit{read}(H', r, i : p) = z$ then since $H'(y) = \bullet z$, $\mathit{join}(H', r, i : p, p')$ is true. Since for any path p'' we have $\mathit{read}(H', y, p'') = \mathit{read}(H', z, p'')$ 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.

3.4 Usefulness

In this section we show that our algorithm actually performs garbage collection. Definition 3.4.1 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 3.4.1 ***(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 3.4.2 ***(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 3.4.1 *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 3.4.1 and 3.4.2 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 3.4.2), 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 3.4.3).

Definition 3.4.3 ***(Forwarding nodes).*** $F(H) \overset{def}{=} \{x \mid H(x) = \bullet y\}$

Theorem 3.4.2 ***(Usefulness).*** *If $H_0 \longrightarrow^* H_n$ forms a GC cycle, then*
$\forall H_i, 0 \leq i \leq n \,.\, F(H_i) \cap D(H_0) = \emptyset$

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 3.4.1 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 3.4.1 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 3.4.2 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 3.4.3. We first define what we mean by the size of a heap.

Definition 3.4.4 ***(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 3.4.3 *Suppose $H \longrightarrow^* H'$ is a GC cycle then,*
$size(H') \leq size(live(H) \cup$ new allocations$)$

3.5 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 [32]. 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 [41], where Russinoff presents a correctness proof in nqthm of Ben-Ari's incremental mark and sweep collector [3]. He shows that it is safe (i.e. nothing but garbage is collected) and that it will eventually collect all garbage.

In [5], Birkedal et al. prove correctness of Cheney's classical stop-and-copy collector [11] 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 [36], 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 [18], 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 [2]. 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 [13], which has been extensively used by others for reasoning (mostly informally though) about correctness of incremental copying garbage collection. In [9], 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.

3.6 Summary

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*.

Chapter 4

Reactive Systems

This chapter is partly based on two previously published research papers [23] and [29].

The main idea of reactive programming is that the program should react to events in its environment. The traditional batch-oriented programming model, however, imposes an active master-slave relationship to the environment – the program *controls* the environment. Even though programming models have been evolving since the batch-oriented days, the view of the environment has persisted. Access to external devices is still made through block-and-wait-for-input procedures subtly hidden by regular subroutine calls. This may be appropriate (even today) when it comes to communication with a wide range of external devices dealing with data storage (hard disks, RAM, etc.). However, when these external ”devices” become more difficult to control, this view of the world becomes highly inappropriate. For instance, a program including a graphical user interface may need to react to mouse clicks, key presses, and network packets in an order determined solely by the environment. Or, an embedded system, which usually has an even wider range of input sources. These types of complex environments cannot be controlled in a simple way such as the case of more controllable external devices. They require a facility for handling (in contrast to controlling) *events* in a robust manner.

4.1 Reactive on top of active

Block-waiting-for-input procedures (e.g. `getChar()`) are common in almost all modern programming languages, and event-driven programs are commonly encoded on top of this model. This is typically done by a so called *event-loop*. Events are captured by a block-and-wait call to an event delivery service offered by the operating system and then distributed to its appropriate handler. A typical structure of the event-loop is shown in Figure 4.1.

In order to preserve aliveness of the system, only one block-and-wait call to the event delivery service is allowed for each concurrent activity. This is done by putting it at the top-level. Utmost care must be taken to ensure that no one of the event handlers makes blocking calls.

Concurrency in such systems are usually achieved by using threads. Designing a multi-threaded system explicitly is a tedious and error-prone task. Shared data must be protected by semaphores, mutexes, etc.. Utmost care must be taken to ensure that the system is free of race conditions and deadlocks, which usually are hidden very subtly.

4.2 Pure reactive programming

A purely reactive system can be defined as a system without block-and-wait calls. Instead of that, the system is designed to *react* to input from its environment. In other words, instead of actively ask for input through a block-and-wait procedure, the default state of the system is idle, ready to react to whatever it is defined to react on.

In contrast to an event-loop based structure as shown in Figure 4.1, a pure event-driven system may connect events directly to their corresponding handlers as shown in Figure 4.2. In order to make this structure more flexible, concurrency and mutual

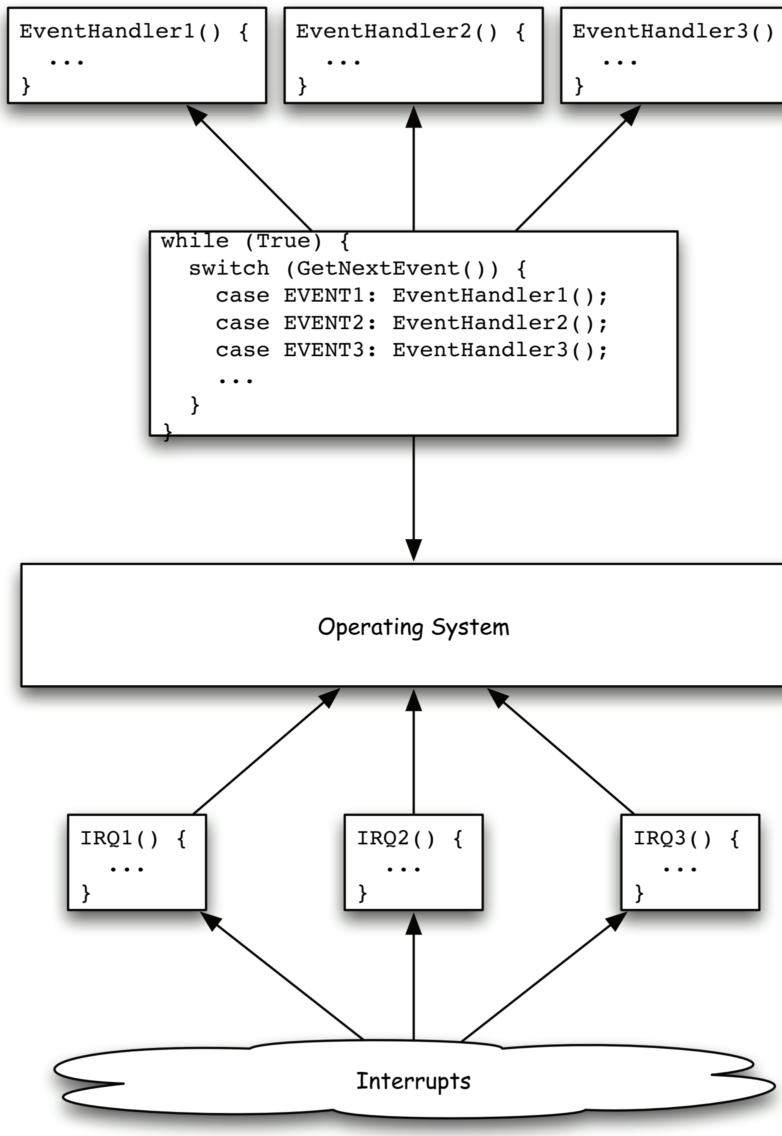

Figure 4.1: A typical event-loop structure.

exclusion could be made implicit through the semantics of a programming model for pure reactive systems.

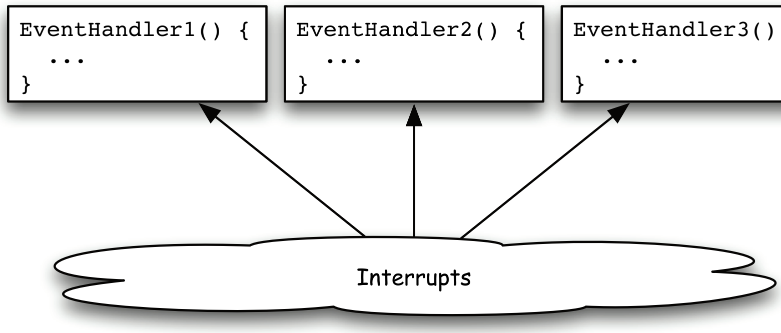

Figure 4.2: Connecting events directly to their corresponding handlers.

In the next section, we will describe such a programming model.

4.3 Timber – Reactive objects

Timber is a, strongly typed, object-oriented programming language for reactive real-time systems. The basis of the language is the semantics of *reactive objects*. This section is just a brief description of the parts of the language that are relevant to the rest of this thesis. More in-depth descriptions can be found in the draft language report [7], the formal semantics definition [10], and the description of the reactive object model [38]. The real-time properties of the language has been described from different point of views and in different contexts in several papers [30, 23, 46, 37, 29].

The reactive objects in Timber are implicitly concurrent. That means activity in one object can execute concurrently with activity in other objects. Furthermore, objects are the only state carrying data structures. In addition, the only way to access the state of an object is through its methods, and methods of one particular object executes under mutual exclusion. That is how state integrity is preserved. In order to preserve aliveness, methods cannot block-and-wait for future events. Events are instead interpreted as method calls and each method will eventually terminate, leaving the object in an idle state. An example showing the concurrency model of Timber is shown in Figure 4.3. The arrows in the model represent message-based communication in both an object-oriented and concurrent programming sense.

Messages can either be asynchronous (equivalent to events in their own right) or synchronous rendezvous operations. In Figure 4.4, an example of a program in concrete Timber syntax is shown.

At the top-level, a Timber program is a set of bindings of names to expressions. In the example (Figure 4.4) an object generator expression (identified by the keyword `template`)

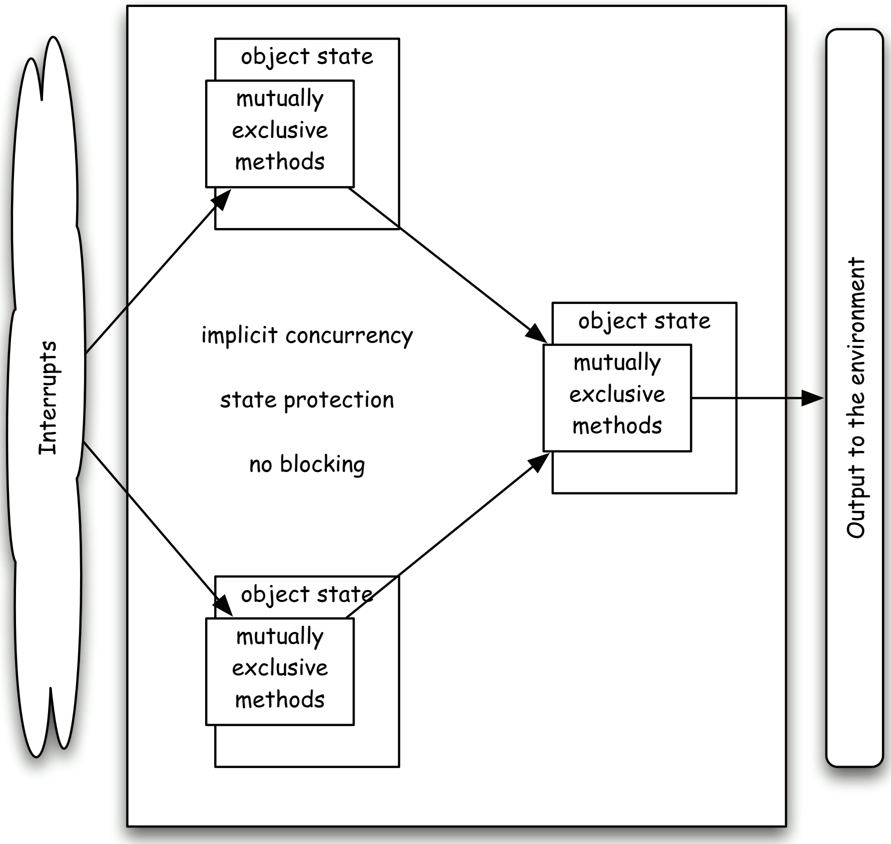
Interrupts
object state
mutually exclusive methods
object state
mutually exclusive methods
object state
mutually exclusive methods
implicit concurrency
state protection
no blocking
Output to the environment

Figure 4.3: Example showing the concurrency model of Timber.

is bound to the name `counter`. The statements within a template construct conceptually consist of state variable initializations and a return statement for the *interface* to the object. The interface is usually a record but could be any data structure. Anyhow, for a template construct to be practically useful, the returned interface should include at least one method. A method, in turn, can be of two kinds: either an asynchronous operation (identified by the keyword `action`), or a synchronous rendezvous (identified by the keyword `request`). An asynchronous method invocation is conceptually an event in its own right, no matter if it is due to an external event or an invocation by another method. Interrupts from the environment is connected to asynchronous methods. In the example, suppose the counter program was running on an interactive machine where one of the input sources is a button. Every time the user pushes the button, an interrupt

```
counter = template
  value := 0
  return {
    read = request
      return value
    incr = action
      value := value + 1
  }
```

Figure 4.4: Example Timber program, A Counter. Example originally found in [37].

connected to the `incr` method is generated. Or, `incr` might only be called from another method based on some internal calculations. Any way, every time `incr` is invoked, it is a unique event.

A statement within a method can primarily be one of three kinds: either an assignment to the state of its object, a template instantiation, or a method call. Local bindings as well as looping and branching constructs are of course also available. Observe the difference between the destructive update of state variables (denoted by `:=`) and the declarative bindings between names and expressions (denoted by `=`). Destructive updates are only available for state assignments within the scope of a template construct. In addition, in the spirit of separating different kinds of assignments, Timber also includes a third binding, which enables one to bind the result of a side-effecting command to a local name (denoted by a left-arrow `<-`). I.e., in contrast to the declarative binding, which binds the name to an expression, the left-arrow binds the name to the result of executing something that may have an externally observable effect.

Aside from the reactive objects, Timber has a pure functional layer based on a call-by-value lambda calculus. It is extended with constructors for representing primitive data types (such as integers, floating point numbers, etc.), as well as immutable data structures (lists, records, tuples, etc.).

4.4 Real-time in Timber

Timber offers a capability to express timing constraints directly in the model. Asynchronous methods (events) are associated with both an *baseline* (earliest release time) and a *deadline* (latest response time). If not explicitly set, the *baseline* of a method invocation is inherited from the method who called it. If the event is due to an interrupt, the baseline is set to the time-stamp of the interrupt. Deadlines are expressed relative to the current baseline. Similar to baselines, deadlines are inherited if not explicitly set. In Figure 4.5, a timeline of a Timber method invocation is shown.

Correct timing behavior of a method is that it must start executing and finish within its timeline. Baselines and deadlines can be adjusted by explicitly expressing it in the definition of an asynchronous method (or by the caller). The keyword `after` is used to denote a extension of the current baseline. Along the same line, `before` is used to extend

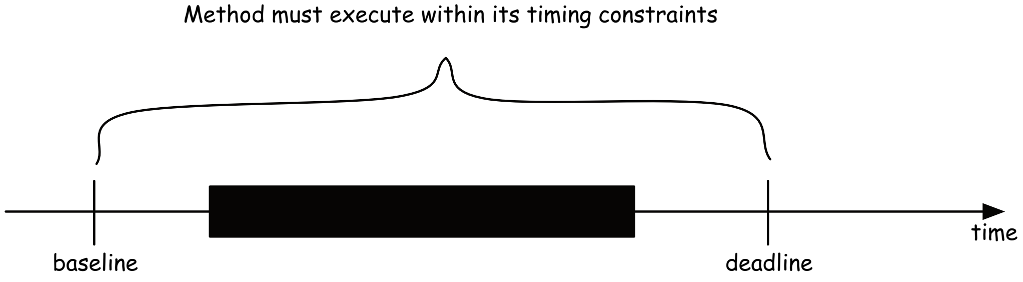

Figure 4.5: Timeline of a Timber method invocation.

```
perobj = template
  ...
  return {
    periodictask = before 50*us action
      ...
      after 50*us periodictask
  }
```

Figure 4.6: Simple example of a periodic task in Timber.

the current deadline. In Figure 4.6, a Timber object with a periodic method is shown. The period and relative deadline is 50μs.

4.5 The Timber runtime kernel

In order to meet the semantic specification of the language, a Timber program will depend on the infra-structure provided by the run-time kernel of the language. Conceptually, the kernel needs to offer the following services:

- Create an object,
- Send a synchronous message, and
- Send an asynchronous message (possibly delayed).

In reality, creating an object maps to allocation of heap storage. The only difference between allocating an object and an immutable data structure (record, tuple, cons cell, etc.) is that in order to enable mutual exclusion between methods of the same object, each object needs to have a *lock* field. A synchronous rendezvous is simply accomplished by locking the object before running the code of the method, and release the lock afterward. An asynchronous method call on the other requires a new *message* (conceptually an execution thread) to be allocated and enqueued in the proper message queue (timer queue if delayed). Once this message gets to run, it simply makes a regular synchronous call.

The scheduling mechanism in the kernel is a preemptive priority scheduler based on deadlines. Delayed messages are scheduled based on baselines. The scheduler interacts with three types of data; objects (includes locks), messages (including executable code, possibly parameters, a baseline, and a deadline), and threads (messages extended with a stack and register file). Objects are the shared resources for which messages acquire and release locks. Asynchronous messages that are latent (either delayed or pending) are, when scheduled to run, promoted to a unique thread. The implicit concurrency of a Timber program is thus accomplished by the scheduler which, for each independent schedulable message, may allocate a new unique thread of execution.

The interface of the scheduler consists of four entry points.

1. Whenever a new asynchronous message is posted (either internally by another method or externally by an interrupt).

2. When a method calls lock or unlock for an object.

3. When a method terminates.

4. When a timer interrupt occurs.

The scheduler manages three main data structures. A stack of active threads (including, of course, each thread's individual stored machine state), a priority queue of pending messages, and a priority queue of delayed messages.

There are a few very important properties of the kernel that are worth mentioning. All lock operations eventually return, either with the acquired lock or with a deadlock error. The lock of an object may only be owned by one message at a time. The highest priority thread, or the thread holding a lock wanted by the highest priority thread, will always be the one scheduled to run. All messages will run after their baselines. The aliveness property of a Timber program is crucially dependent upon the fact that Timber code always will follow a successful lock with an unlock and that all locks are acquired in a nested manner.

Chapter 5

Garbage Collecting Reactive Systems

In Chapter 3, we described a model of an incremental copying garbage collector based on process terms in a labeled transition system. Garbage collector increments were represented by internal transitions and mutator interaction were represented by labeled transition. A labeled transition were either a read, a write, or an allocation. In this chapter we describe how this abstract process model is implemented as part of the Timber run-time kernel.

5.1 The memory set up

Since the algorithm is a copying garbage collector, we need to divide the heap into two parts, *tospace* (which is the active part of the heap), and *fromspace* (which is the inactive one). It is not entirely true that this spaces need to be same size even though it simplifies the underlying intuition. I.e., copying objects from one space into another space of exactly the same size will always succeed. If the objects do not fit into the new space it would not have fit in the old space either. However, the actual need of space to copy to is not dependent on the size of the old space but rather on how much memory the live objects in the old space will require. A survival rate of ten percent would require the new space to be at least a tenth of the size of the old space. Anyhow, allowing the border between tospace and fromspace to slide requires a more rigorous (read cumbersome) scheme for both allocations and free memory calculations. We will use a fixed, equally sized, partitioning of the heap. In Figure 5.1, the memory set up of our implementation is shown.

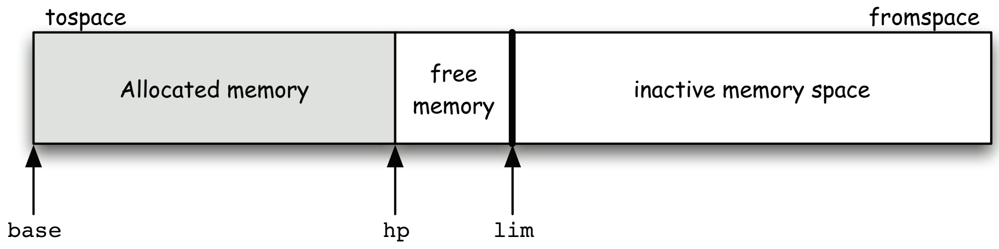

Figure 5.1: The memory set up for our implementation.

5.2 Interfacing the Timber run-time kernel

From the garbage collector's point of perspective, it really does not matter what kind of mutator it is set out to garbage collect for, as long as it follows the rules of the labeled transitions. Even though mutator access is defined by the path notion, it is clear that, in reality, mutator access will commonly be made from pointer fields directly accessible at the place in the execution of the program. Nonetheless, each such access must correspond to a path-based labeled transition beginning in the rootset of the system. Otherwise, the pointer field would not have been locally accessible by the program in the first place. In other words, we assume the program to be *well-behaved* in terms of aliveness (i.e., live $\subseteq$ reachable).

Definition 5.2.1 *Let a mutator be well-behaved iff it can be described by a relation R on heaps such that for each HRH' there exists at least one labeled transition $H \xrightarrow{l} H'$.*

We will, from now on, only reason about the corresponding labeled transitions of the mutator relation.

The first question we would like to answer is that of the entry point from the run-time kernel of Timber to the garbage collector. The scheduler is the facility that distributes control among threads. Adding the possibility to schedule garbage collection as a dedicated thread seems like a straightforward decision since it is in line with the formal model of the garbage collector (i.e., the mutator(s) and the garbage collector are concurrent processes). However, one of the key properties of the kernel is that *always* runs the highest priority thread. So what should the priority of the garbage collector be? Thread priorities of regular Timber messages are derived from their deadlines, but what would the deadline of the garbage collector be? There is no easy answer to this question, as it depends on many factors. We know for sure that the priority is (or should be) partly dependent upon the amount of free memory available at the moment. The other side of the coin is the mutators' future storage needs. However, this may be very difficult to determine.

In terms of scheduling, this thesis will make some simplifying assumptions. We assume that there will always be enough memory available so that it is sufficient to run the garbage collector at idle time. Finding and scanning the rootset incrementally is generally a very difficult task; sometimes even impossible. The problem is that the rootset of an arbitrary system is not constant. At an arbitrary time, finding the whole rootset requires scanning not only static fields and CPU registers but also the run-time stack, which might be of an arbitrary depth. For a incremental rootset scanning to be trustworthy, it cannot let the pause time depend on the size of an arbitrary sized data structure. Scanning the run-time stack incrementally is bound to be very tedious since the content of the stack is extremely volatile. In the worst case, the scanning process would need to be restarted at each increment since the content of stack has been alternated altogether. The idea of restarting the scanner of an arbitrary big rootset at each increment is not very appealing, especially not when it comes to real-time systems.

Restricting the garbage collector to only run in idle time removes this problem altogether. When a Timber system reaches its idle state (i.e. no pending messages), there will neither be any run-time stacks, nor CPU registers, to scan. In fact, the only roots in an idle Timber system is the queue of delayed messages and the the set of bindings between interrupts and messages. This is captured by two pointers in the run-time kernel. Thus scanning the rootset of an idle Timber system can be done in constant time.

In addition, restricting the garbage collector to only run in idle time makes the schedulability of a Timber program decoupled from the behavior of the garbage collector. This is especially important since the garbage collector's work-load is heavily dependent on the choices made by the scheduler. The only thing needed to be assured is that it must, at all times, be enough free memory available to run all pending messages. We will return to this topic at the end of this chapter. However, the necessary analyses for assuring sufficient free memory availability do not fit within the scope of this thesis.

5.3 Implementation details

In this section we will look at the interesting details of the implementation. Most of the internal transitions, defined in Figure 3.1 and 3.2, are straightforwardly implemented (e.g. by adjusting a pointer or index). However, some of the details deserves some extra attention.

5.3.1 The `info`-field

Since Timber is a strictly typed language the locations of pointers can safely be determined a priori. Thus, whenever a new allocation is requested, the positions within the new *node* that can hold pointers are known, and the necessary type information can be attached to each node[1]. This is done by statically create an array of pointer offsets for

[1]We use the term node instead of object in order to avoid confusion between objects on the heap and Timber objects

each unique type, and store a pointer to the corresponding array as a header in each node when it is allocated.

5.3.2 The write barrier

The write-barrier needs to do two things. First of all it needs to set the dirty bit of the mutated node (we will see how this is implemented efficiently in Section 5.3.3). The other thing is to, if the node is black, revert its color back to gray. We do not want to actually move the node just to be on the right side of the border. That would cost too much, and even more alarming, the pause time of the write-barrier would be dependent upon the size of the node. What we do instead is that we allocate a certain node in the gray heap that, when the scanning process reaches the node, notifies the scanner that a black node has been mutated. The scanner then temporarily jumps back to the black node, scans it, and returns back to where it was. These notifier nodes can easily be detected by using a unique type identifier (instead of the array pointer used for regular nodes).

5.3.3 The dirty bit

Another feature that could induce unnecessary overhead is the dirty bit. However, the only nodes where the dirty bit matters is the two nodes currently being scanned and currently being copied. Whenever the mutator needs to set the dirty bit of a node it can check if the node is the one currently being scanned or the one currently being copied. If the node is the one which is currently being scanned, a static dirty bit field for the current node being scanned is set. At each iteration of the scanning process, this field is checked and if set take the SCANRESTART transition. Similarly, a static dirty bit for the node currently being copied is checked at each iteration of the copy process and if set take the COPYRESTART transition. This requires two extra comparisons to be done by the write-barrier. In the next section we will look at how the frequency of write-barrier calls may be reduced significantly due to the concurrency model of Timber.

5.3.4 Reducing the frequency of write-barrier calls

We know that the only mutable data structure in Timber is the object. We further know that the only way to gain access to these mutable state fields is through the methods of the object. Since we assumed that the garbage collector has the lowest priority in the system we can safely say that, at all times, if a method successfully acquires a lock, it will return the very same lock before the garbage collector is started or resumed. This is immediate from the fact that messages in Timber are non-blocking. Thus, when a message acquires a lock within a garbage collection cycle then, and only then, is the write-barrier for that object called. In other words, since the garbage collector never may interleave the execution of a message, only one write-barrier call for the whole message is necessary.

5.3.5 The read barrier

The read transition assumes that forwarding pointers always are followed. However, a read barrier is only necessary for data structures that might suffer from inconsistencies due to mutations. Read only data does not need to be protected by a read barrier. Similarly to the write barrier, the read barrier can also be called only once for every method invocation instead of upon every single memory access. The efficiency gained here is probably even higher than for the write barrier since reads tend to occur much more frequently than writes.

5.4 Garbage collection in a real-time context

A Timber program may create reactive objects dynamically at run-time, just like any other data structure. In schedulability theory terms, this feature corresponds to the ability to dynamically create both tasks and shared resources. In order to achieve a priori schedulability guarantees, we need to limit the expressiveness of Timber somewhat. A sufficient (but probably not absolutely necessary) way to support static schedulability analyses would be to restrict the program from dynamically creating objects. That is, all shared mutable data (objects) and all tasks (methods) would be statically known. If this restriction is set, which all in all is not that controversial in the context of hard real-time systems, it puts the garbage collector in a much more favorable situation. If objects are statically allocated outside the heap, it means that the heap will contain no mutable data structures at run-time. That, in turn, makes both the read barrier and the write barrier superfluous. The same holds for the dirty bits. The only induced overhead we have left from the garbage collector would be the type info field, which, as we discussed earlier, is not necessary for anything else than robust classification of node content into pointer and non-pointer fields[2].

[2]Overhead imposed by the garbage collector technique, such as two semispaces, moving live data, etc., is not accounted for here.

Chapter 6

Conclusions and Pointers to Future Work

The motivating factor behind this thesis is that real-time programming faces more and more complex missions. In order to lessen the burden of the programmer, more sophisticated runtime features, such as garbage collection, become desirable. However, integrating garbage collection in a real-time system is a great challenge.

We identify the key issues needed to be solved in order to accomplish real-time garbage collection. First of all, one needs a thorough understanding of an incremental garbage collector algorithm suitable for real-time systems. The main requirement for an garbage collector algorithm to be suitable for a real-time system is that it has to be incremental. Furthermore, the execution time of each increment must be small and bounded (preferably constant). Once such an algorithm has been found and well-understood, one needs to understand its interaction with the real-time tasks. Ultimately, one need to find a way to schedule the garbage collector in such way that (1) no timing requirements of the system are violated and (2) sufficient free memory is available at all times.

6.1 Contributions

The approach we have presented in this thesis is to fulfill the first schedulability requirement (1) by only run garbage collection in idle time. Of course, we have initially assumed that there is sufficient memory available at all times in order to enable running the worst-case busy period of the system. The key is that, since garbage collection is only scheduled in idle time, regular schedulability analysis of the system may be performed without knowledge about neither the system's memory needs, nor any details about the garbage collector internals. The only thing, concerning the memory system and its garbage collector, that is required is knowledge about the cost of synchronization between the memory system and the mutators. That means, in order to compute correct WCET of each task, the synchronization overhead must be accounted for. However, since we have deployed our garbage collector in Timber, the required synchronization is easily computed – in fact, it is at all times a small constant overhead.

In order to assess the second schedulability requirement (2), one needs a robust model of how the garbage collected memory system behaves in the presence of parallel mutator processes. In this thesis we have presented such a model, where we use an incremental copying garbage collector. The model is based on process terms in a *labeled transition system* (LTS), where garbage collector increments are defined as internal transitions and the mutators' interface to the heap is captured by labeled transitions. We have reasoned about the basic properties of the model. We have shown that the garbage collector eventually terminates and that the meaning of the heap is preserved at all times. These properties constitutes *correctness* of the model. We have also shown that the model enables a priori memory requirement analysis by reasoning about *usefulness* of the garbage collector in terms of reduced memory occupancy.

6.2 Pointers to future work

In order to completely asses the second schedulabilty requirement (2) one needs to determine the memory needs and usage of the real-time tasks. The next natural step is to build a framework for analyzing memory usage of Timber programs. Parameters, such as worst-case memory allocation rate, maximum live memory, etc. are vital in order to assess the systems worst-case memory usage. Once such information is made available, it would be possible to determine if the distribution of idle periods is sufficient for our garbage collector to reclaim memory fast enough.

In order to increase the number of schedulable systems, various extensions and optimizations may be done to the garbage collector algorithm. One such extension would be to let the heap model capture generations.

Since all dynamic data (except the objects) in Timber are immutable, the average survival time is probably short. Nodes that are allocated and known to become garbage within the same reaction, could be allocated in fromspace (i.e. white) in order to reduce the unnecessary conservatism of our collector. However, since this information has to be known at allocation time, it would require a priori analysis to determine for which allocation sites fromspace allocation should apply. We believe that the region inference technique [6], used for region based memory management [49], is a possible and promising way to determine this.

Even though it has its great advantages, restricting the garbage collector to only run in idle time is by no means the optimal scheduling policy for real-time garbage collection. Investigating the possibilities of liberating this requirement without losing its advantages altogether is also a possible track for future work.

Bibliography

[1] Andrew W. Appel. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, 1987.

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

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

[4] E. C. Berkley and Daniel G. Bobrow. *The Programming Language LISP: Its Operation and Applications.* Inc. Cambridge, MA, fourth edition, 1974.

[5] 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.

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

[7] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.

[8] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[9] 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. ACM Press, August 1984.

[10] 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.

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

[12] W. T. Comfort. Multiword list items. *Communications of the ACM*, 7(6):357–362, 1964.

[13] 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.

[14] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, 1993.

[15] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[16] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of the conference on Object Oriented Systems, Languages and Applications, OOPSLA'91*, volume 26, pages 33–46, Phoenix, Arizona, USA, October 1991. ACM Press.

[17] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD thesis, Lund Institute of Technology, 1998.

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

[19] Lorenz Huelsbergen and James R. Larus. A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data. In *Principles Practice of Parallel Programming*, pages 73–82, 1993.

[20] R. John M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1084, November 1982.

[21] Franz Inc. *Allegro CL USer Guide, Version 4.1.* March 1992.

[22] Richard Jones and Rafael Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons Ltd, 1996.

[23] Martin Kero, Per Lindgren, and Johan Nordlander. Timber as an RTOS for Small Embedded Devices. In *Proceedings of the first workshop on Real-World Wireless Sensor Networks REALWSN'05*, 2005.

[24] Martin Kero, Johan Nordlander, and Per Lindgren. A Correct and Useful Incremental Copying Garbage Collector. In *Proceedings of the 2007 international symposium on Memory Management (ISMM'07)*, Montréal, Québec, Canada, October 2007.

[25] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[26] Donald Ervin Knuth. *The art of computer programming. Vol. 1, Fundamental algorithms.* Addison-Wesley, 1973. First edition published in 1968.

[27] C. M. Krishna and Kang G. Shin. *Real-Time Systems.* McGraw-Hill, 1997.

[28] Henry Lieberman and Carl E. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[29] Per Lindgren, Johan Nordlander, Martin Kero, and Johan Eriksson. Robust Real-Time Applications in Timber. In *Proceedings of the 2006 IEEE international conference on Electro/Information Technology*, pages 191–196, May 2006.

[30] Per Lindgren, Johan Nordlander, Linus Svensson, and Joakim Eriksson. Time for Timber. Technical report, Division of EISLAB, Department of Computer Science and Electrical Engineering, Luleå University of Technology, 2004.

[31] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[32] 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.

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

[34] Marvin Minsky. A LISP Garbage Collector Algorithm Using Serial Secondary Storage. Technical report, Cambridge, MA, USA, 1963.

[35] David A. Moon. Garbage collection in a large LISP system. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 235–245. ACM Press, August 1984.

[36] 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.

[37] J. Nordlander, M. Carlsson, M.P. Jones, and J. Jonsson. Programming with Time-Constrained Reactions, 2005.

[38] Johan Nordlander, Mark P. Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black. Reactive Objects. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Arlington, VA, 2002.

[39] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[40] Tobias Ritzau. *Memory Efficient Hard Real-Time Garbage Collection.* PhD thesis, Linköpings Universitet, 2003.

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

[42] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[43] Robert A. Shaw. *Empirical Analysis of LISP Systems.* PhD thesis, Stanford University, 1988. Technical Report CSL-TR-88-351.

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

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

[46] Linus Svensson, Joakim Eriksson, Per Lindgren, and Johan Nordlander. Language-Based WCET Analysis of Reactive Programs, 2005.

[47] M. Swanson. An improved portable copying garbage collector. Technical Report OPnote 86-03, University of Utah, February 1986.

[48] Lars-Erik Thorelli. Marking algorithms. *BIT*, 12(4):555–568, 1972.

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

[50] David A. Turner. Miranda – a non-strict functional language with polymorphic types. In *Record of the 1985 Conference on functional programming and computer architecture (FPCA'85)*, pages 1–16, Portland, Oregon, USA, September 1985. Springer-Verlag.

[51] David Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167, 1984.

[52] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[53] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage collected systems. *ACM SIGPLAN Notices*, 26(6):177–191, 1991.

[54] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational refernces in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.

[55] David S. Wise, Brian Heck, Caleb Hess, Willie Hunt, and Eric Ost. Uniprocessor performance of a reference-counting hardware heap. Technical Report TR-401, Indiana University, Computer Science Department, May 1994.

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

[57] Benjamin Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, 1990.

[58] Benjamin G. Zorn. *Comparative performance evaluation of garbage collection algorithms.* PhD thesis, University of California at Berkley, March 1989.