Reference counting

Miroslaw Blazej
blazej@student.agh.edu.pl

Michal Dygas
dygas@student.agh.edu.pl

AGH University of Science and Technology

17-06-2020

Introduction

Reference counting is an **intuitive garbage collection** algorithm that **records in each chunk the number of pointers that point to it**; when the number drops to zero the chunk can be declared garbage; in a literal sense, reference counting collects garbage, unlike the other garbage collection algorithms, which actually collect reachable chunks. In line with the name reference counting, we will call pointers references in this section.

Also, they may still need a **one-shot** garbage collector as backup for situations in which they cannot cope with the demand.

On-the-fly

Reference counting is **On-the-fly** type of garbage collector (also called incremental): some actions are performed at each call of **Malloc** and/or **Free**. These actions make some local modifications to the chunk structure to increase the probability of finding a free chunk when needed.

Rules

1. When a chunk is allocated from the heap, its reference count is initialized to one.
2. Whenever a reference to the chunk is duplicated, its reference count is increased by one (incremented).
3. Whenever a reference to the chunk is deleted, its reference count is decreased by one (decremented).
4. If the reference count drops to 0, the chunk can be freed because it is no longer reachable

Rules

Implementation challenges

1. keeping track of all reference manipulations
2. recursively freeing chunks with zero reference count

keeping track of all reference manipulations

The compiler **inserts special code for all reference manipulations** and at each call of Malloc and/or Free - incrementing the reference count when a reference to a chunk is duplicated and decrementing it when such a reference is deleted.

Besides assignment statements, the compiler also has to add reference increasing code to **parameter transfers**, since a reference that is passed as a parameter is effectively assigned to a local variable of the called routine. Note that not all references in the running program are references to chunks on the heap; many of them point to blocks in the program data area, and all referencecounting code must make sure it does not follow such references.

p := q additional reference manipulations pseudocode

```
if Points into the heap (q):
    Increment q.referenceCount;
if Points into the heap (p):
    Decrement p.referenceCount;
    if p.referenceCount = 0:
        FreeRecursively (p);
```

recursively freeing chunks pseudocode

```
procedure FreeRecursively(Pointer);
    if not IsPointerIntoHeap (Pointer): return;
    if Pointer.referenceCount != 0: return;
    for each i in 1 .. Pointer.numberOfPointers:
        if IsPointerIntoHeap (Pointer.pointer [i]):
            Decrement Pointer.pointer [i].referenceCount;
            FreeRecursively(Pointer.pointer [i]);
    FreeChunk(Pointer);
```

Recursively freeing cons

Cons

Recursion is, however, an unwelcome feature in a garbage collector since it requires an **unpredictable amount of stack space**.

Having a garbage collector fail for lack of memory is kind of embarrassing, though, and several techniques have been invented to avoid the problem. The best solution is using **pointer reversal**.

Identification techniques

figures that will illustrate this algorithm

Identification techniques

figures that will illustrate this algorithm

Algorithm issues

Cons

1. efficiency
2. memory fragmentation
3. failes to reclaim a cyclic data structure

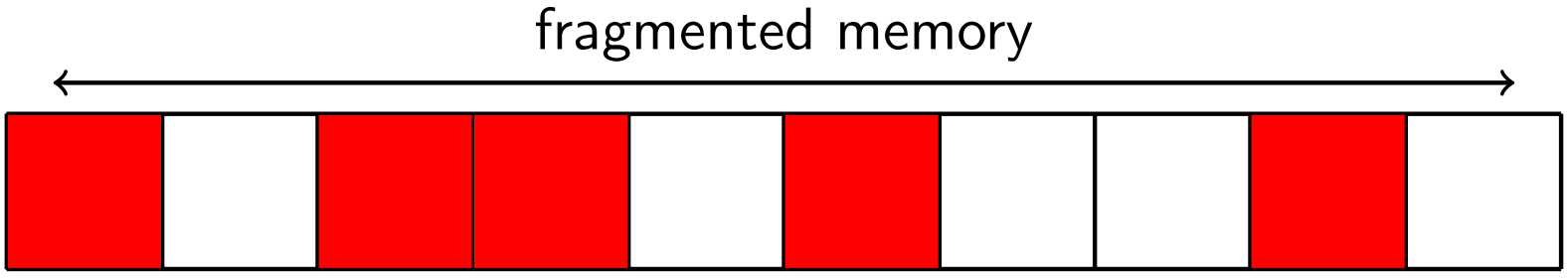
Efficiency

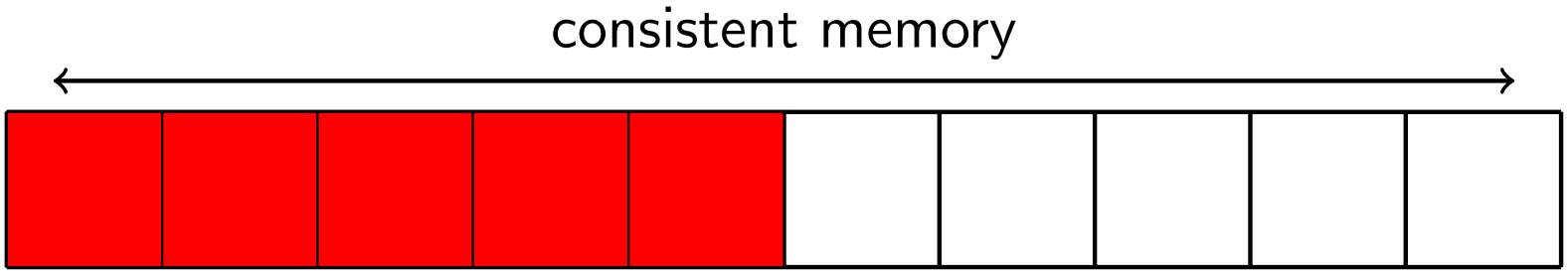
The compiled code has to monitor **all reference manipulations** and each and every reference manipulation requires the adjustment of the associated reference counts.

This is a considerable **overhead** in comparison to other garbage collection techniques that do not monitor any pointer action and reclaim garbage chunks only when needed.

Memory fragmentation

The free list is augmented with the reclaimed chunks, but it remains **fragmented**. In principle doing a compaction phase during a reference counting allocation request is possible, but few reference counting garbage collectors go to such lengths.





Cycles

The problem with reference counting is that it takes its decisions by considering **only one node** in the graph at a time and *in order to reclaim a cyclic data structure all nodes in the data structure should be considered as garbage together*.

Once reference counting has failed to reclaim a cyclic data structure, the chunks involved **will never be reclaimed**. This has the unfortunate effect that free space leaks away, which might even cause the program to run out of free space when other garbage collectors would be able to reclaim the cyclic structures and allow the program to continue.

Identification techniques

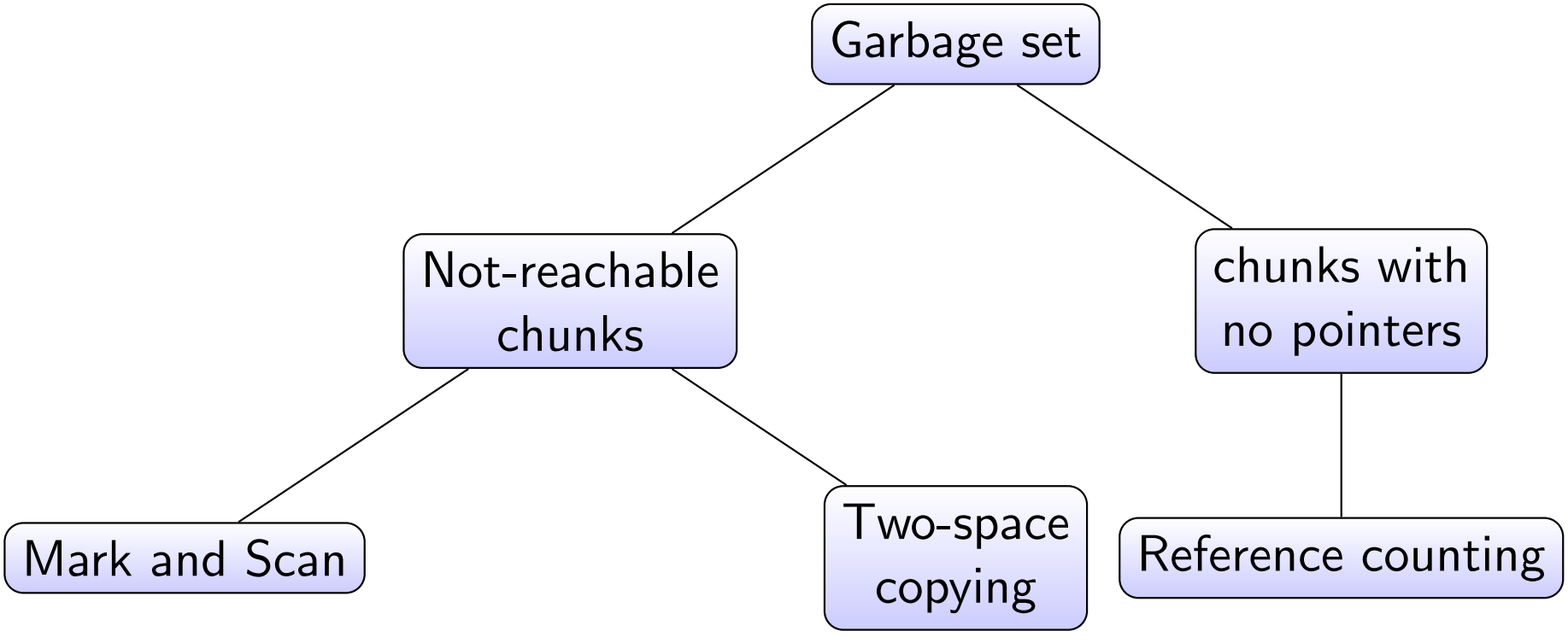


Figure: Garbage set identification techniques

Identification techniques

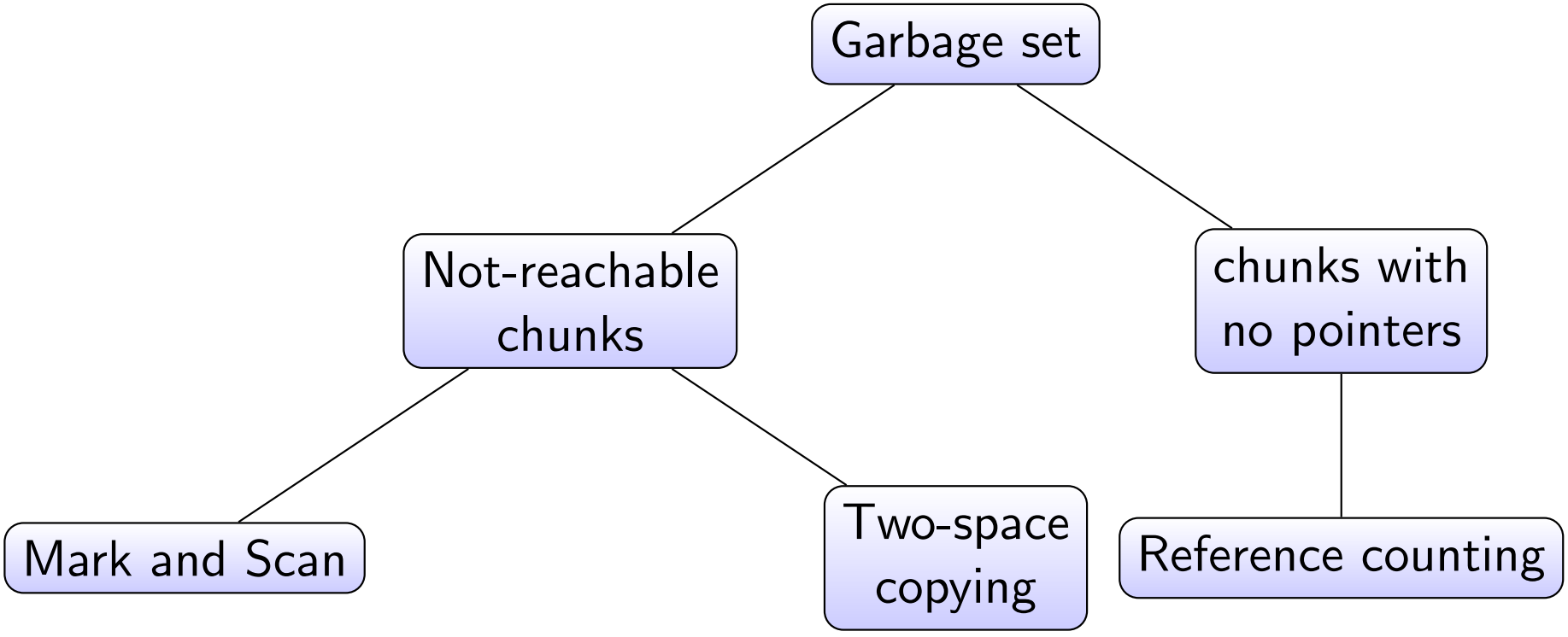


Figure: Garbage set identification techniques

Reference counting: brief introduction

It is a really simple technique which, as the name suggests, keeps count of the number of references to each object.

Pros

- Directly identifies garbage chunks
- Simple and reasonably efficient

Cons

- Requires all pointer actions to be monitored during program execution
- May not recover all garbage chunks

Mark and scan: brief introduction

Mark and scan identifies reachable chunks and concludes that the rest is garbage.

Pros

- Reasonably efficient and does not require pointer monitoring

Cons

- Quite complicated
- Recover all available memory

Two-space copying: brief introduction

Two-space copying copies the reachable chunks from a memory region called "from-space" to a memory region called "to-space" - the remaining space in to-space is a single free chunk.

Pros

- Very efficient
- Does not require pointer monitoring
- Moderately complicated

Cons

- Wastes half of the memory

Short notice of compaction

Unfortunately locating and freeing proper chunks may not always be the final step as some of the techniques lead to **memory fragmentation**.This behaviour would cause significant problems in runtime as, even though, total free memory is satisfied the program cannot allocate data larger than the largest free chunk. Solution for this situation is called **compaction**. Compaction groups free memory blocks on one side and used on the other creating a single large free chunk. It is more complex and time consuming than just freeing the unused memory but it's fundamental as it makes the memory really *available*.

Conceptual example of compaction

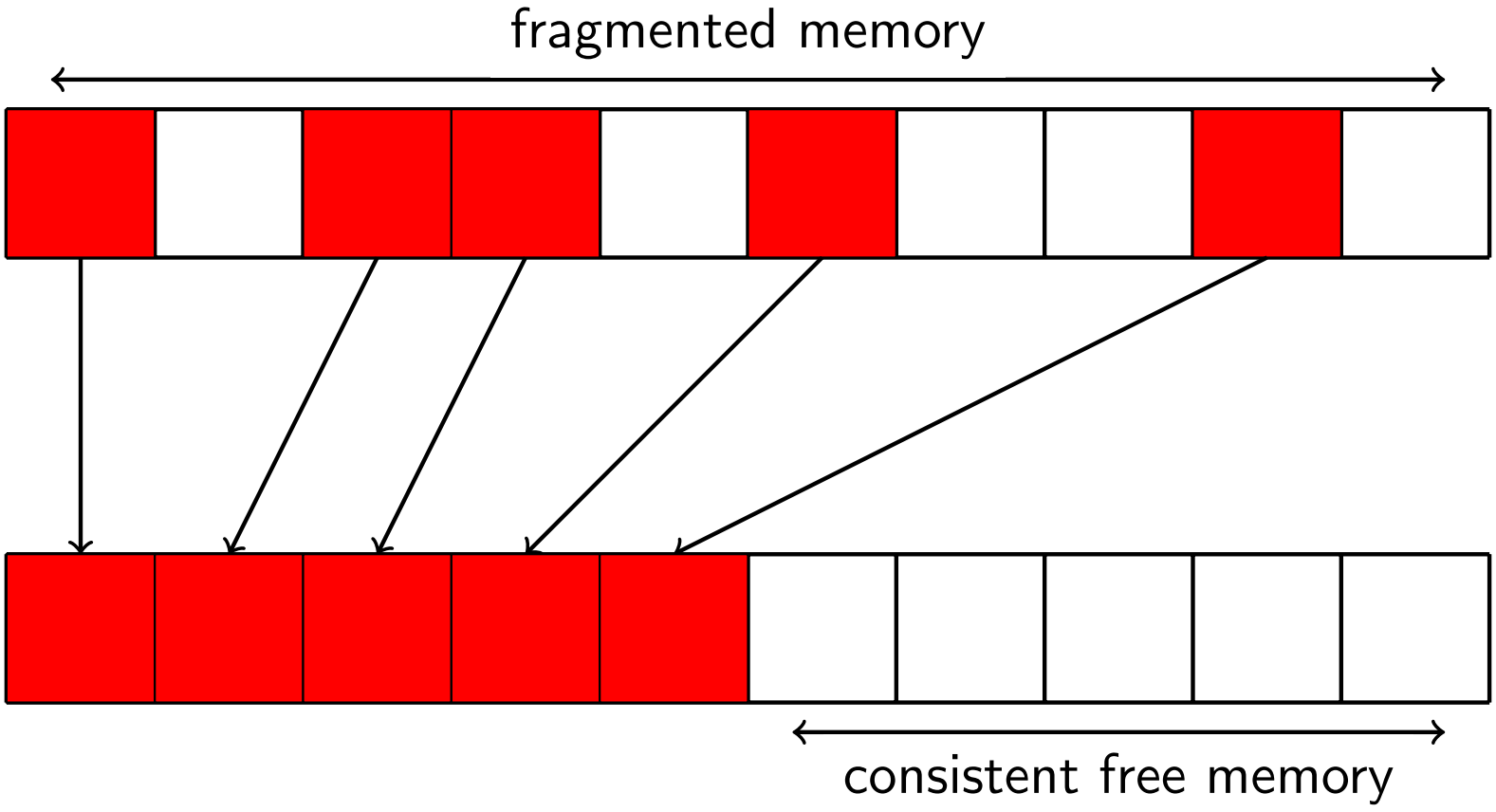


Figure: Compaction example

Fundamental types of garbage collection algorithms

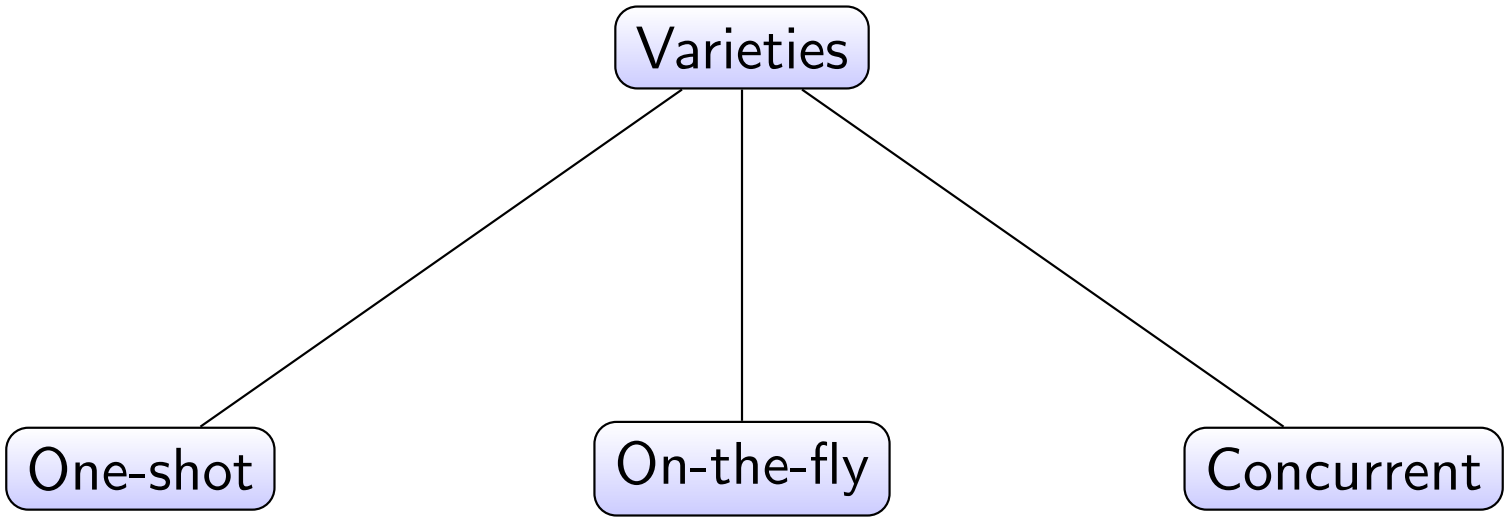


Figure: Garbage collection algorithm types

One-shot

This group of algorithms is fairly simple since the garbage collector is in full control while running, however, because of that its execution can be disruptive. That's why it's more suitable for compilers than interactive programs.

Depicted routine

1. Starting the garbage collector
2. Execution till completion while in full control of all chunks
3. Returning

On-the-fly

Some garbage collector actions are performed at each call of Malloc and/or Free. These actions make some local modifications to the chunk structure to increase the probability of finding a free chunky when needed.

- On-the-fly garbage collectors are usually much more difficult to construct than one-shot garbage collectors, but are smoother and less disruptive in their operation.
- They may still need a one-shot garbage collector as back-up for situations in which they cannot cope with the demand.

Concurrent

In concurrent approach garbage collector **runs on a second processor**, different from the one that runs the program. It runs continuously and concurrently, and tries to keep memory garbage-free. Unfortunately, concurrent garbage collection is sometimes also called on-the-fly, in spite of the fact that this term suggests one agent rather than two.

Implementation of garbage set techniques

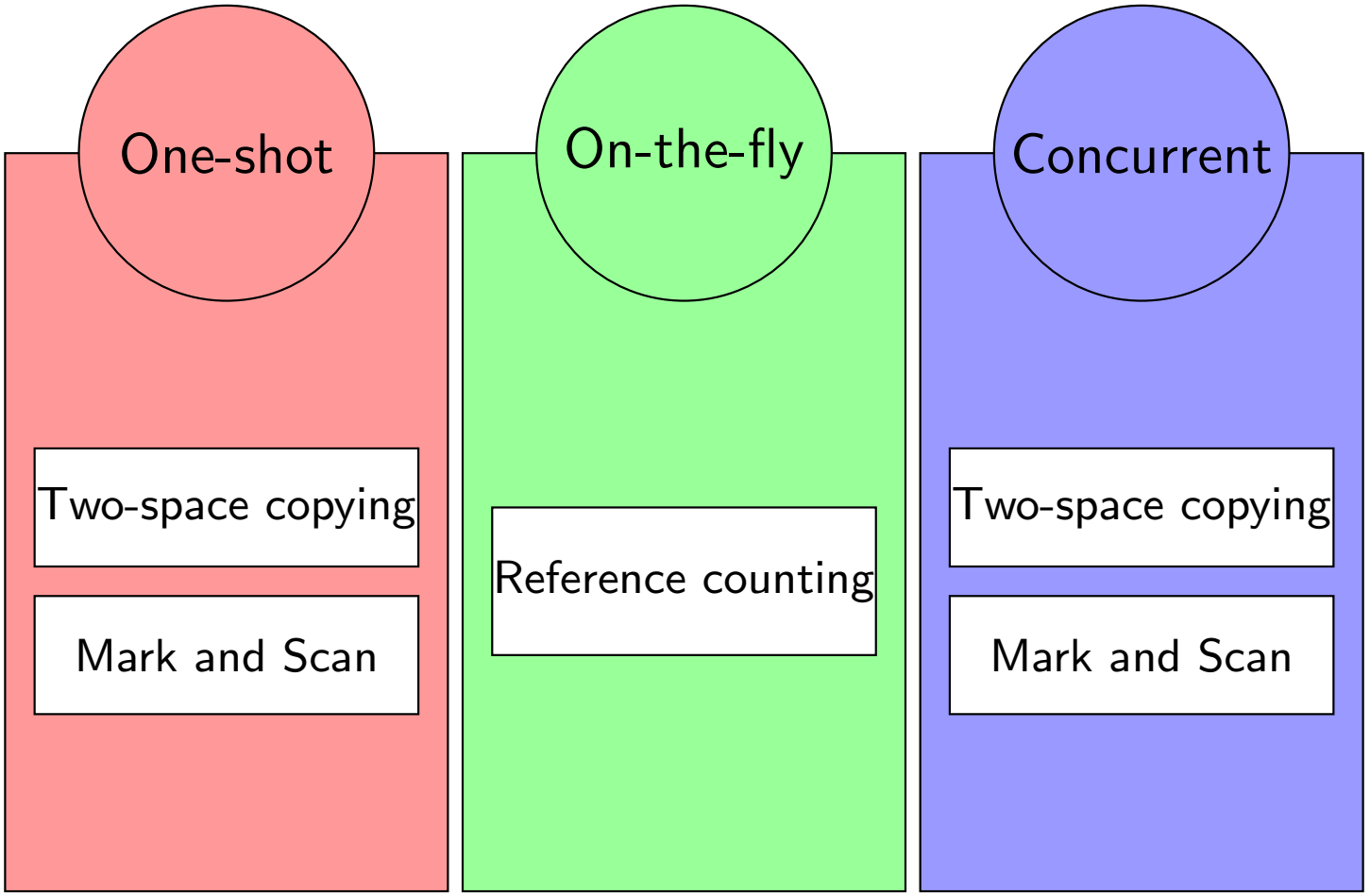


Figure: Categorization of different garbage set implementations

Getting to know pointers before implementation

We can distinguish two kinds of pointers:

- Direct - A chunk of the memory is reachable for the program using just this one (direct) pointer.
- Indirect - A chunk of the memory is reachable for the program using some other (indirect) pointer.

The directly available pointers can be located in various places, depending on the implementation. These places may include the global/local variables, registers and perhaps many others.

Non-heap memory and root set

Non-heap memory

Also called the program data area, and consists of the memory that is directly accessible to the program code.

Root set

Is a conceptual notion that depicts the set of pointers in the program data area. Usually it's not implemented but only conceptually present in the garbage collector's program code. It's worth noting that it isn't a list of pointer values but just the set of all pointers in the program data area.

Garbage collection problem decomposition

The pointers from the root set may point to chunks in the heap (controlled by the garbage collector) what makes this chunks reachable. Those reachable chunks can obviously contain pointers to other areas in the heap which are then reachable as well.

That's why we can divide the problem of finding all reachable chucks (thus problem of the garbage collection) into three subproblems:

1. Finding all pointers in the program data area with their types (so finding the root set)
2. Finding all pointers in a given chunk with their types
3. Finding all reachable chunks using the information from subproblems 1 and 2

Garbage collection problem decomposition

To solve subproblems 1 and 2 the garbage collector needs compiler support. It has to deliver information about the pointer layout of the program data area and each chunk type. We cover this scenarios in the next slides.

Garbage collection algorithms themselves can be treated as the solutions for the third subproblem which is usually solved by implementing them as runtime routines.

Root set and heap example

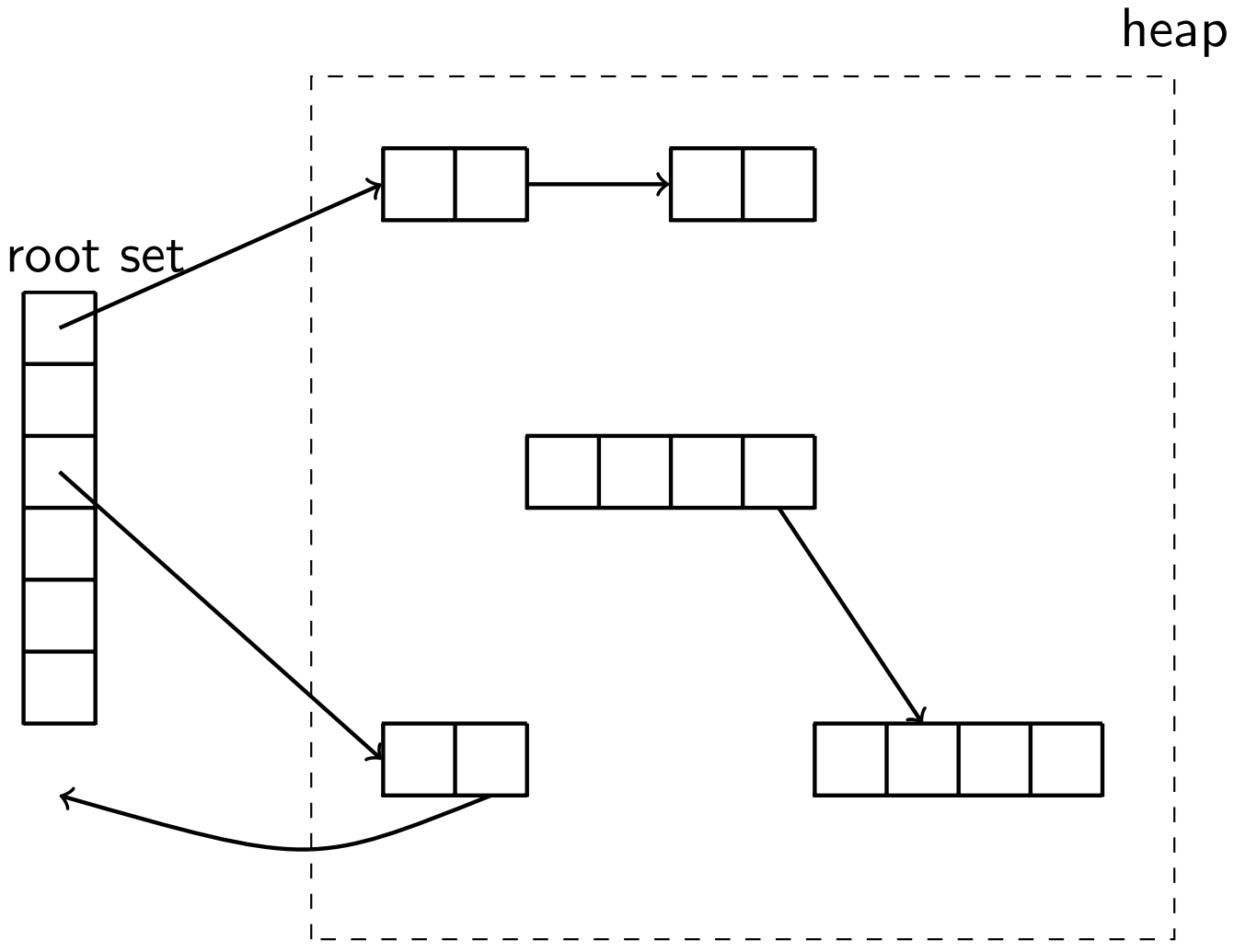


Figure: A root set and a heap with reachable and unreachable chunks

Pointer consistency

To do its job the garbage collector has to be able to understand all the pointers that it deals with, since it will follow any pointer to find the garbage.

That's why the pointer consistency (also called pointer validity) must be ensured (usually by the language definition and the compiler).

Compilers support for garbage collection

To give a proper support for the garbage collection process compiler has to:

1. **Provide the root set** and information about the **pointer layout** of each chunk to the garbage collector (the pointer layout of a chunk C describes the position of each pointer P in the chunk, together with the type of the chunk that P points to).
2. Make sure that all **reachable pointers**, both in the program data area and in the heap, **are valid** when the garbage collector is activated.

Chunks pointer layout

The compiler is in full control of the layout of chunks. The only problem is how to transfer the knowledge of the pointer layout to the garbage collector - chunks needs to be **self-descriptive**.

Two common approaches:

- Chunks can carry their pointer layout information in each copy
- They can all have the same layout

Storing chunks pointer layout

To properly store the pointer layout compiler can:

1. Generate a **bitmap** for each chunk type. With this method, chunks must be self-descriptive, since just having the pointer must be sufficient for the garbage collector to continue. So each chunk must either contain its bitmap or a pointer to its bit map
2. Generate a specific **routine** for each chunk type, which calls a garbage collector routine passed as a parameter for each pointer inside the chunk.
3. The compiler can organize the chunks to start off with an **array containing all pointers**, followed by the other data types. With this organization, the collector only has to know the location of the pointer array and the total number of pointers inside the chunk.

Program data pointer layout

The root set is supplied by running a **library routine** that **scans the program data area** and calls a specific garbage collection routine for each pointer the program data area contains. It is then up to the specific garbage collection routine to see if the pointer is interesting and perform the proper actions. To perform its task, the library routine must be able to find all pointers in the program data area, with their types, and be sure each pointer is valid.

The problem is that the pointer layout of the program data area, unlike that of chunks, is complicated and dynamically variable.

Program data area

The program data area consists of the **global data area** and a stack holding one or more **stack frames** or **activation records**. The pointer layout of the global data area is known and constant, although it may be distributed over several source program modules. To know the pointer layout of the stack, the garbage collector has to know which activation records it contains, and what the pointer layout of each activation record is.

Both pieces of information are dynamic, so activation records must be also **self-describing**.

Conclusion

As you can see garbage collection can be a really complex field. It consists of seemingly simple ideas which require a lot of work and knowledge to be properly implemented.

We hope that this presentation borough this matter closer to you and made you familiar with basic concepts of the garbage collection and basic garbage collection algorithms.

Bibliography

Our research was based on the book:
Dick Grune et al. "*Modern Compiler Design*"