Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 22

AUTOMATIC MEMORY MANAGEMENT

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

This lecture introduces Dynamic Memory Management and Automatic Memory Management, including a discussion of the requirements of functional languages for the dynamic allocation and freeing of memory at runtime

The lecture then explores the issue of fragmentation, explaining what it is, how it occurs, and discussing ways in which it can be removed

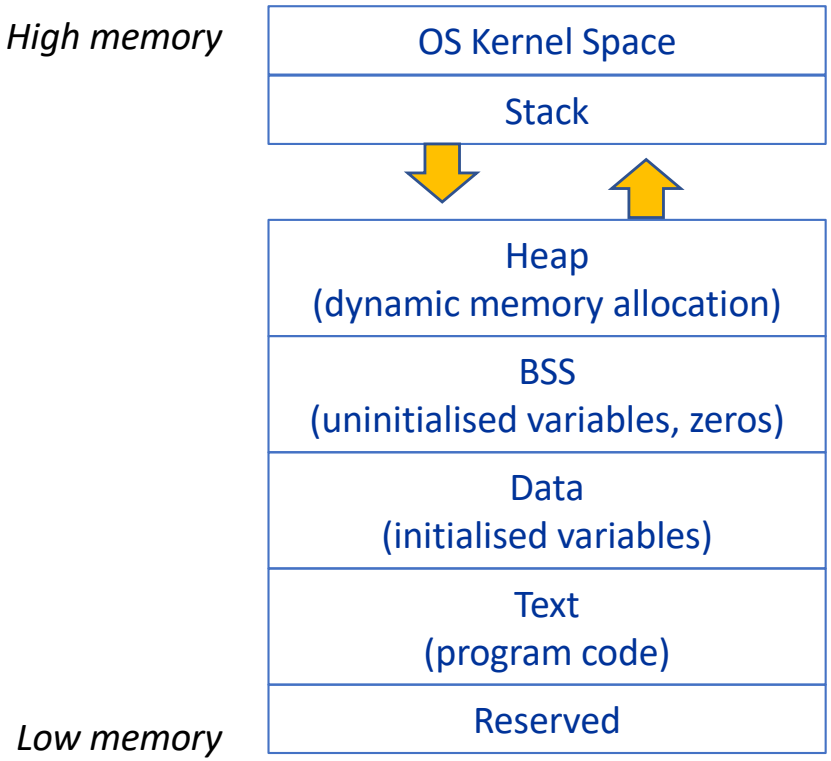
CONTENTS

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

DYNAMIC MEMORY MANAGEMENT

Context

- A typical memory layout for a Unix process is:



Dynamic Memory Management – Context

A typical memory layout for a process (i.e. a running program) in a Unix or Unix-like Operating System is illustrated in this diagram

At the low end of memory, there is a small amount of reserved memory, followed by (moving higher in memory):

- a binary image of the program (called "Text")
- an area for program variables that have been initialised ("Data")
- an area for program variables that have not been initialised, though they may be given values of zero at the start ("BSS" – Block Started by Symbol)
- an area for dynamic memory allocation that the program doesn't know it needs until runtime, whose upper boundary can be raised dynamically ("Heap")
- an area for automatic variables, function parameters and return values etc., which starts at a high memory address and grows downwards in the memory space ("Stack")
- an area for Operating System data associated with this process ("OS Kernel Space")

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

DYNAMIC MEMORY MANAGEMENT

Context

- The "heap" is a large pool of memory that can be used by programs for storing values
- Efficient conservation and re-use of Heap memory is important
- Re-use is motivated by the observation that some heap data becomes no longer required or unreachable
- At any given time, some parts of the heap are in use ("live"), some are unused ("free") and some may be previously-used but no longer required or reachable ("garbage")

Dynamic Memory Management - Context

For dynamic memory management we are interested in management of the "Heap" memory area

The Heap is a large pool of memory that can be used by programs for storing data

Although the upper boundary of the Heap can be extended (upwards in memory), functional language implementations often require a large heap, and there is a limit to how far the Heap can be extended. Therefore the efficient conservation and re-use of Heap memory is important

The re-use of heap memory locations is motivated by the observation that data placed in the heap may later become unwanted and unreachable (since data can only be "reached" if the program still retains the address of that data)

At any given time, some parts of the heap are in use ("live"), some are unused ("free") and some may be previously-used but no longer required or reachable ("garbage")

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

DYNAMIC MEMORY MANAGEMENT

- Need to support dynamic data allocation efficiently
- Need to support making no-longer-needed memory available for future allocations
- The blocks of data that need to be allocated dynamically may be of differing sizes

A Storage Manager (SM) library, with two functions:

- *malloc* takes a size argument, finds a suitable free block in the heap and returns the address of that block
- *free* takes the memory address of a previously allocated block, and makes that block available for re-use

These two library functions will manage the differently-sized blocks of “live” and “free" memory in an optimal way

Dynamic Memory Management

Since data allocated in the heap is not known until run-time, a mechanism is required to support the program's need to allocate such data in an efficient way

Furthermore, since memory is a scarce resource, any allocated data that is no longer required should be made available for re-use for a future dynamic allocation of data into the heap

Also note that the blocks of data that need to be allocated dynamically may be of differing sizes

One solution to this is to provide a Storage Manager (SM) library, with two functions “*malloc*” and “*free*”:

- *malloc* (a contraction of "memory allocation") takes an argument that is the **size** of the amount of memory that the program needs to be allocated in the heap – the function finds a suitable free block in the heap and returns the address of the start of that block
- *free* takes a **memory address** that references a block of memory that has been previously allocated, and it makes that block available for re-use for a future allocation by malloc

These two library functions will manage the differently-sized blocks of “live” and “free" memory in an optimal way

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

DYNAMIC MEMORY MANAGEMENT

With **dynamic** memory management, the programmer must include calls to the malloc and free library functions

- In particular, it is the programmer's responsibility to ensure that:
  - **all** allocated heap memory that is no longer required is released by calling the *free* function
  - **only** allocated heap memory that is no longer required is released using the *free* function

Dynamic Memory Management

With **dynamic** memory management, the programmer must include calls to the *malloc* and *free* library functions

In particular, it is the programmer's responsibility to ensure that:

- **all** allocated heap memory that is no longer required is released by calling the *free* function
- **only** allocated heap memory that is no longer required is released using the *free* function

In a future lecture we will explore different memory allocation techniques

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

AUTOMATIC MEMORY MANAGEMENT

- Prime source of bugs comes from pointers
- Functional languages don't normally have pointers
- But the Storage Manager library functions for dynamic memory management (DMM) must use pointers:
    - malloc returns a pointer
    - free takes a pointer as an argument

Thus (i) DMM can't be used for functional languages, and (ii) many errors arise from DMM in imperative languages

An alternative is **automatic** memory management (AMM)

With AMM, the compiler generates code to call *malloc* and *free* automatically, and at run-time special code runs to:

- identify data to be allocated dynamically in the heap
- detect "garbage" data in the heap
- make garbage blocks available for re-use

Automatic Memory Management

A prime source of bugs in imperative programs comes from the manipulation of pointers (memory addresses). Functional programming languages don't normally give the programmer access to pointers, and this leads to a large reduction in errors

However, the Storage Manager library functions for dynamic memory management must use pointers:

- *malloc* returns a pointer
- *free* takes a pointer as an argument

This means that (i) dynamic memory management can't be used for functional languages (because the programmer never sees the pointers), and (ii) many errors in non-functional languages can arise from the use of dynamic memory management (because of the use of pointers)

An attractive alternative is **automatic** memory management

With automatic memory management, the programmer does not explicitly call *malloc*, nor *free*. Instead, the compiler generates code to do this automatically, and at run-time special code runs to:

- identify which data need to be allocated dynamically in the heap
- detect when previously allocated data is no longer needed ("garbage"), and
- make garbage blocks available for re-use

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

AUTOMATIC MEMORY MANAGEMENT

- AMM is an onerous responsibility
- AMM still uses *malloc* and *free*, but the programmer will never see them or use them
- Runtime system extended, and compiler modified
- Compiler or runtime system inserts calls to *malloc* at appropriate points. The malloc routine will:
  - search for a block of "free" heap memory
  - tag the block "in use" or "live"
  - return a pointer to that chunk, used by code generated automatically by the compiler
- In later lectures we will discuss different algorithms for identifying garbage and tagging "free" blocks

Automatic Memory Management

Taking responsibility for automatically allocating heap memory when needed and detecting which data in the heap are no longer needed is an onerous responsibility – it must never go wrong

An Automatic Memory Management (AMM) system will still use *malloc* and *free* functions, but the programmer will never see them or use them

The run-time system must be extended, and the compiler must be modified to produce program code that interacts with the AMM code. For example, the compiler will identify points in the program code where new memory needs to be allocated in the heap and insert calls to *malloc* at those points. The malloc routine will:

- search for an appropriate block of "free" heap memory
- allocate the block (tagging it "in use" or "live")
- return a pointer to that block, to be used by code generated automatically by the compiler (the programmer never sees the pointer)

In later lectures we will discuss different algorithms for identifying correctly at runtime those blocks of heap data that are "garbage" (no longer needed) and then tagging those blocks as "free" so they can be used by *malloc*

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

MEMORY MANAGEMENT FOR FUNCTIONAL LANGUAGES

- Functional languages normally don't give the programmer access to pointers
- AMM must be used
  - Garbage Collection to identify/re-use garbage
  - Memory allocation
- The Graph Reduction runtime system must identify when to call *malloc* and make correct and safe use of the memory addresses
  - accessible to the Graph Reduction system
  - accessible to the Garbage Collection system

In the previous two lectures we explored the implementation technique of Graph Reduction, including both fully interpretative Graph Reduction and the possibility of the compiler producing greater or lesser amounts of native code

Functional languages normally do not permit the programmer to access memory addresses, and so Automatic Memory Management must be used

This means that Garbage Collection (see later) must be used to identify garbage and make it available for re-use

It also means that memory allocation must be automatic. The runtime system that implements Graph Reduction (with management of greater or lesser amounts of native code) must identify when calls to *malloc* must be made. It must also make correct and safe use of the memory address returned by *malloc*, in such a way that it is properly accessible to the runtime system (both the Graph Reduction system and the Garbage Collection system)

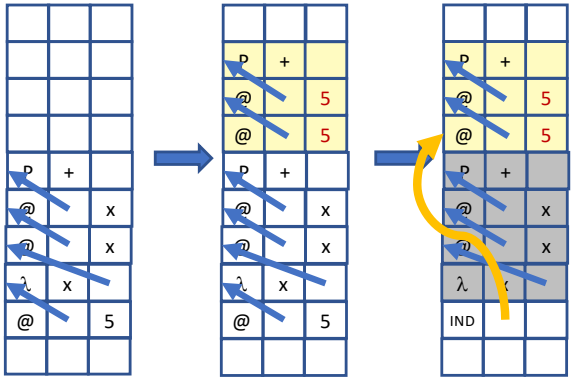
It is in some cases feasible that a block of native code created by the compiler might include a call to *malloc*, and this would require the compiler also to generate native code to make correct and safe use of the memory address returned by *malloc* as described above

Next we will see how the mechanisms of Graph Reduction cause memory to be allocated and to become garbage

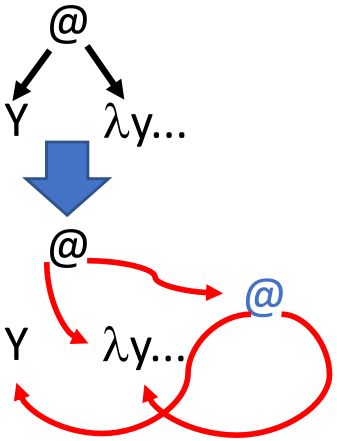
FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

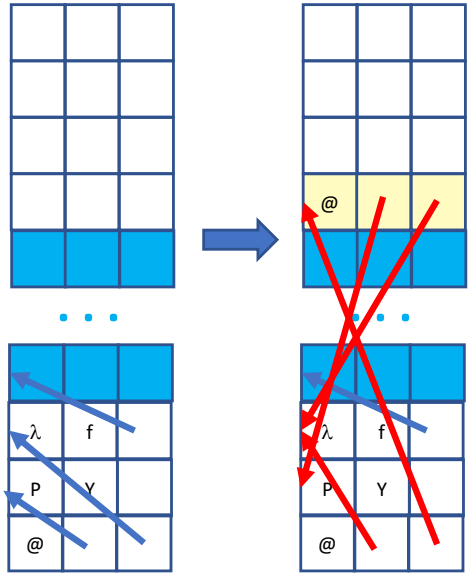
MEMORY MANAGEMENT FOR FUNCTIONAL LANGUAGES

- β reduction:



- δ reduction
  Y combinator
  non-cyclic:





On this slide we explore how the Graph Reduction mechanism of β reduction and δ reduction require calls to the *malloc* library function. The examples use fully interpretive Graph Reduction

Recall from Lecture 20 that the first step of a β reduction is to make a copy of the function body. This copying step is shown in the upper diagram (from Slide 15 of Lecture 20 - the new copy is held in the newly allocated memory locations coloured yellow)

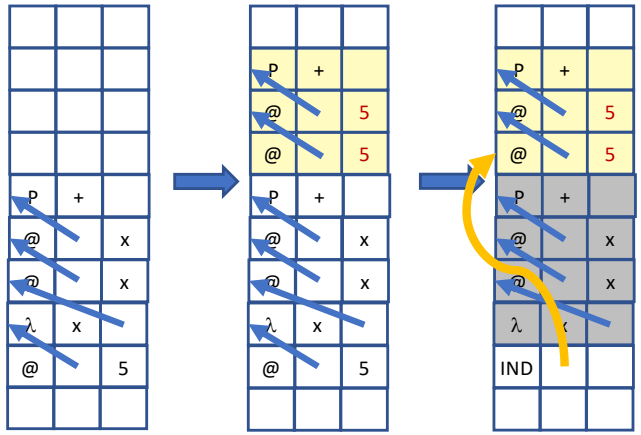
We don't always know at compile time which code will be evaluated, and therefore the compiler cannot always predict which β reductions will be performed – so the copying of the function body will generally require memory to be allocated dynamically. The Graph Reduction code (part of the runtime system) will make a call to *malloc* (or similar – see later) to do this

Normally a δ reduction would not require a call to *malloc*, since the result can be contained within the memory addresses used for the redex (this includes a cyclic implementation of the Y combinator and implementation of a "CONS" operator to build a list node). If a non-cyclic implementation of the Y combinator were required, this would need a call to *malloc*, as shown in the lower diagram (blue locations are the function body, assumed to be large, and yellow locations are newly allocated memory, new/changed pointers in red)

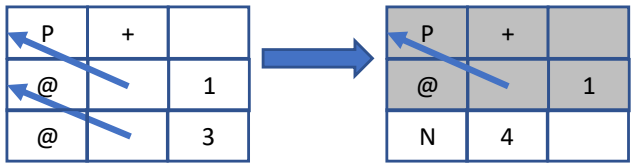
FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

MEMORY MANAGEMENT FOR FUNCTIONAL LANGUAGES

- β reduction:



- δ reduction



Both β reduction and δ reduction can create garbage, as illustrated in these diagrams

The upper figure is the same as used in the previous slide: it shows a β reduction being performed. Notice that, after the root node of the redex has been overwritten with an indirection node pointing to the newly created copy, the memory containing the original copy of the function body is no longer reachable from the root of the redex – these unreachable locations are coloured grey and are "garbage"

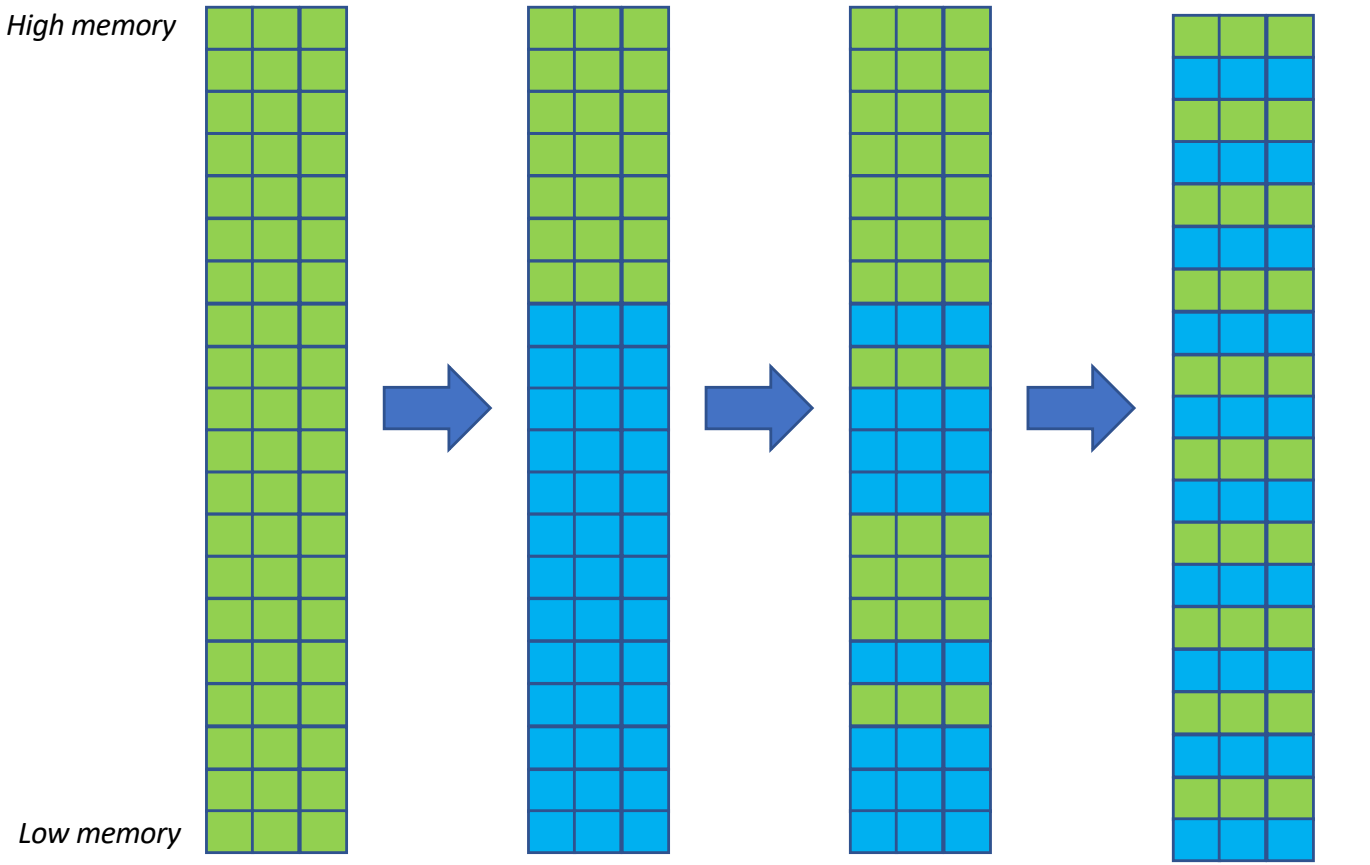
In general, whether the original function body becomes garbage immediately after a β reduction depends on whether the function is only applied once in the program – in which case garbage is created – or whether it is applied multiple times (i.e. the definition is shared amongst more than one application node) – in which case garbage is only created when it is no longer shared

The lower figure illustrates a δ reduction where (1 + 3) is reduced to the result 4. Notice how after the δ reduction has finished the grey memory locations are now garbage. As with β reduction, the amount of garbage created will depend on the extent to which the arguments are shared

It is also possible for partial applications to be shared, which can affect the creation of garbage

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

THE PROBLEM OF FRAGMENTATION IN THE HEAP



No allocations, contiguous free memory

Some allocations, contiguous free memory

After freeing, fragmented free memory

After freeing, worst case fragmented free memory

Having discussed how functional language implementations need to allocate memory dynamically (and the need to detect and "free" garbage automatically), we now explore one of the key problems facing the designers of memory management systems – fragmentation

When a program starts running all of the free memory in the heap will exist in a single contiguous block (though there may be some pre-allocated live blocks). Successive dynamic memory allocations will typically be implemented by successively allocating blocks from the same end of the heap, such that the remaining free memory always exists in a single contiguous block

At various times during the running of the program, it will "free" some previously used blocks. If these freed blocks occur in the middle of two or more "live" blocks, then the available free memory becomes split – as illustrated in the diagram. The free memory has become fragmented

The problem of fragmentation can occur regardless of whether the blocks being allocated are always of the same size (as in this diagram), or of different sizes (most memory management manages variably-sized blocks) – what matters is the order in which blocks are allocated and freed

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

THE PROBLEM OF FRAGMENTATION

Why is fragmentation a problem?

- Out-of-memory errors due to free blocks being fragmented
- Virtual memory performance degradation due to free blocks being fragmented
- Virtual memory performance degradation due to live blocks being fragmented

Fragmentation causes many problems for computer systems in general – and for dynamic or automatic memory management in particular

The most obvious problem that arises due to free blocks being fragmented is that there may come a time where the program requests a block of a certain size (call it "X") and the total amount of free memory is > X, yet the allocation fails with an "out of memory" error because there is no *single contiguous free block* of a size >= X

- When a program requests memory dynamically, it is requesting a contiguous amount of memory of a certain size

A second problem is that the heap is often mapped across several (possibly many) virtual memory (VM) "pages", only a few of which are held in RAM at any one time. Memory access to explore free blocks for suitability to service an allocation request might cause multiple VM pages to be moved into and out of RAM ("paging"), causing performance degradation

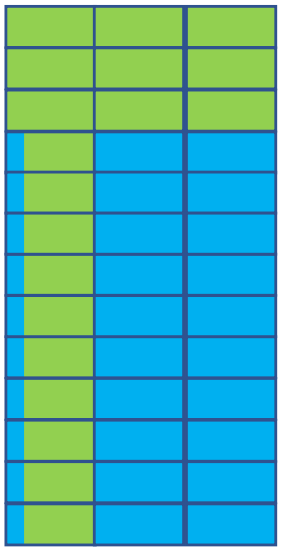
A third problem occurs where there is a correlation in time between proximity of time in allocation of blocks and proximity of time in subsequent live block access – if successively-allocated live blocks have been allocated in different VM pages then this could lead to excessive paging activity and performance degradation

FUNCTIONAL PROGRAMMING
AUTOMATIC MEMORY MANAGEMENT

THE PROBLEM OF FRAGMENTATION

Types of fragmentation

- External fragmentation
- Data fragmentation
- Internal fragmentation:



Internal fragmentation may occur inside live memory

External fragmentation

- The fragmentation described on Slide 13 is generally known as "External Fragmentation"

Data fragmentation

- The fragmentation of live blocks described on the previous slide is generally known as "Data Fragmentation" – especially where data for a single data structure might be fragmented across many blocks in different parts of memory

Internal fragmentation

- By contrast with both of the above, "Internal Fragmentation" occurs when some of the memory in a live block is never used by the program
    - often due to the *malloc* code responding to an allocation request with a pointer to a block that is **bigger** than the requested size – e.g. due to algorithmic constraints (minimum block sizes)
    - may be due to other code systematically asking *malloc* for more memory than is necessary – e.g. using a whole 32-bit word for a cell tag when only 8 bits are necessary (see figure)
- Internal fragmentation is sometimes an accepted cost of improved time performance due to memory/cache alignment

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

REMOVING FRAGMENTATION WITH COMPACTION

- Three phases:
  - Mark the live blocks
  - Compact via relocation of live blocks
  - Update pointers to moved blocks
- Compaction techniques
  - Arbitrary compaction
  - Linearising compaction
  - Sliding compaction

There are several **compaction** algorithms that can be used to remove fragmentation, all of which make several passes over the heap and can be time-expensive operations. In general the algorithms have three phases:

- Mark the live blocks
- Compact by relocating some live blocks
- Updating the pointers to blocks that have moved

There are three classes of compaction techniques:

- **Arbitrary**: blocks are moved without regard to their original order. This can be simple and fast, especially for fixed-size blocks
- **Linearising**: blocks that originally pointed to each other are moved into adjacent positions if possible
- **Sliding**: blocks are slid to one end of the heap, squeezing out free cells and maintaining the original order of allocation

Note that re-ordering of live blocks (as in Arbitrary compaction) gives poor spatial locality and may lead to poor VM performance and fewer cache hits

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

REMOVING FRAGMENTATION WITH COMPACTION

- The Haddon and Waite compactor
  - mark all live blocks
  - move live blocks and create the "break table"
  - sort the break table
  - update all pointers
- Relocation:
  - find first live block
  - save its start address "a", and total free memory found so far "s"
  - slide live block down, and slide break table up
  - store (a,s) at the end of the break table
  - repeat

The Haddon and Waite compactor is a Sliding compactor that operates on variable-sized blocks and makes only two passes over the heap. It works as follows:

- mark all live blocks
- move live blocks and create the "break table" (see below)
- sort the "break table"
- update all pointers

Relocation proceeds as follows:

- find the first live block above the first free block, assume a "break table" already exists in the first free block
- save in the break table the start address "a" of first live block, and total free memory found so far "s"
- incrementally copy the first live block down to the start of the first free block, while incrementally copying the break table upwards
- repeat

The break table is sorted to assist pointer update

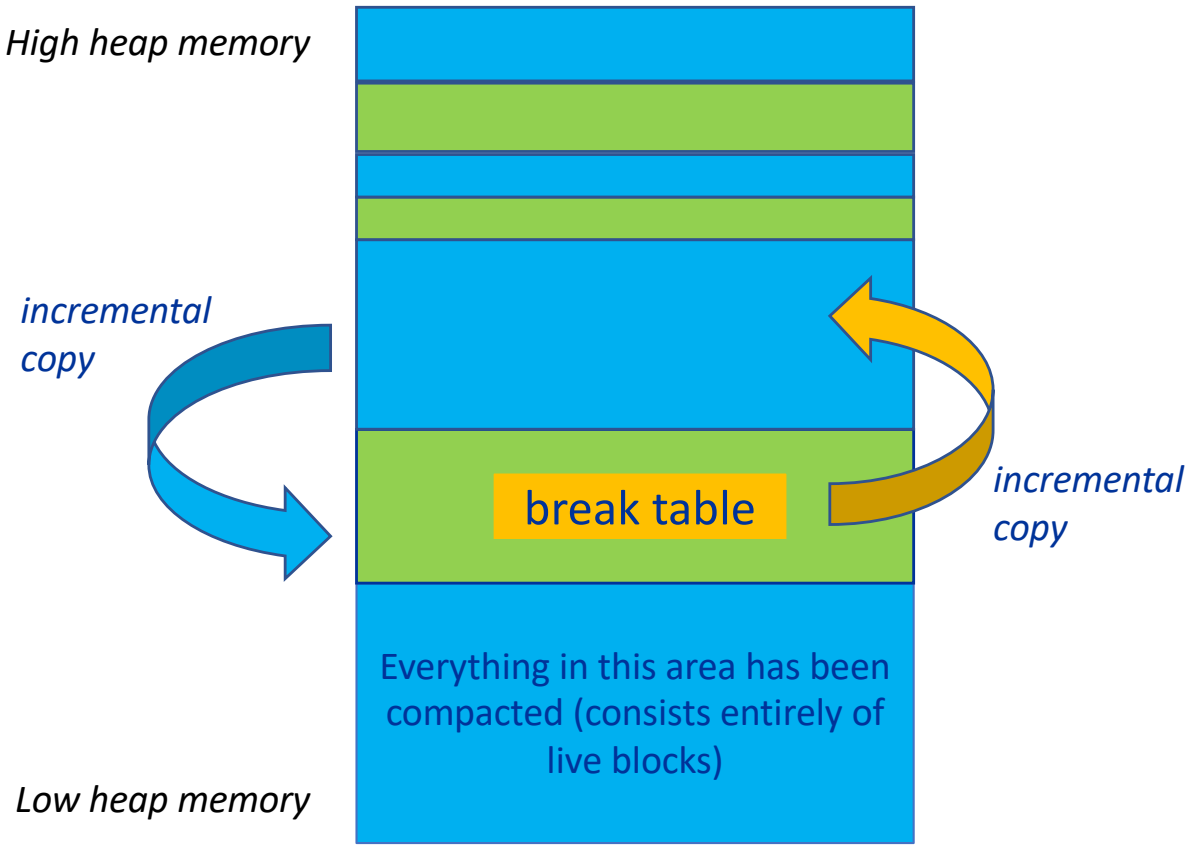
Update proceeds as follows:

- for each pointer p found in any live block, search the break table for adjacent data entries (a,s) and (a',s') such that a <= p < a'
- replace p with p-s

The break table exists in the one remaining free block, and can now be ignored (it is never seen by the program)

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

REMOVING FRAGMENTATION WITH COMPACTION



This figure illustrates the state of the Haddon and Waite compactor halfway through the moving phase

- The first live block above the first free block is incrementally copied (sliding) down so that it is contiguous with the fully compacted area
- Simultaneously the break table is extended with (a,s) and incrementally copied up to the memory that is freed by sliding the live block down
- The algorithm doesn't need any other memory to hold the break table
- The algorithm maintains the ordering of the original blocks
- The algorithm naturally deals with variable-sized blocks
- If live blocks are already marked, it only needs two passes over the heap
- Pointer update assumes pointers to heap blocks only occur in heap blocks (which may not be true!)

During the pointer-update phase (the final pass over the heap), the compactor needs to be able to distinguish memory addresses from other kinds of data of the same size. This requires the compactor to understand the memory layout and conventions used by the compiler and the runtime code that creates and manages dynamic blocks of memory – this is a requirement for any runtime code that seeks to move blocks of live memory

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

REMOVING SOME FRAGMENTATION WITH COALESCING

- Applicable only to a specific form of fragmentation where two or more adjacent blocks are all free
  - for 2+ adjacent free cells
- Coalescing algorithm applied each time a block is freed:
  - inspect the immediately following block and merge with it if it is free
  - inspect the immediately preceding block and merge with it if it is free

There is a form of fragmentation that may occur where the *free* mechanism leads to two or more adjacent blocks are all free

This is a form of fragmentation because it may appear that only small free blocks are available to *malloc*, whereas actually *malloc* should be able to service larger requests for memory without causing an "out of memory" error

All such adjacent free blocks should be merged into a single large free block. This merging can either be done during compaction, or as a periodic search of the entire heap, or as an incremental merging that is attempted each time a block is freed. This latter approach is often called **coalescing:**

- inspect the immediately following block and merge with it if it is free
- inspect the immediately preceding block and merge with it if it is free

The ease with which coalescing can be achieved depends on the memory allocation strategy being used – we shall return to this in a later lecture

FUNCTIONAL PROGRAMMING AUTOMATIC MEMORY MANAGEMENT

CONTENTS

- Dynamic Memory Management (DMM)
- Automatic Memory Management (AMM)
- Memory Management for Functional Languages
- The Problem of Fragmentation
- Removing Fragmentation with Compaction
- Removing Some Fragmentation with Coalescing

In summary, this lecture has introduced Dynamic Memory Management and Automatic Memory Management, including a discussion of the requirements of functional languages for the dynamic allocation and freeing of memory at runtime

The lecture then explored the issue of fragmentation, explaining what it is, how it occurs, and discussing ways in which it can be removed

The next lecture will explore specific memory allocation techniques