Friday, 25 November 2016

CS4201-Programming Language Design & Implementation
Practical 2 Garbage Collection

INTRODUCTION

This report clarifies the simulation of two garbage collection mechanisms proposed by Cheney and Bartlett respectively. Both algorithms have been simulated and tested using JAVA. According to the implemented test cases, both implementations work properly by collecting the non-live objects. In particular, Cheney's (Copying Garbage Collector) has been tested on large random generated heaps including collection after mutation. The output results are available in properly named text files in the zip file. The source code is under the src directory, including Javadoc and regular comments where necessary.

DESIGN DECISIONS

The garbage collector behaviour follows the practical's description. Given a heap and a stack with ambiguous roots, the garbage collector is triggered. In the source code the heap is implemented as an String array of size n. Every array index represents a cell in the heap. The stack is emulated as an integer array and each of its elements contain a number which might be a valid pointer.

Cheney's Algorithm

The CopyingGarbageCollector class is designed to implement the garbage collector as described in a 1970 ACM paper by C.J. Chene (Chene & Nelson, 1970). It is extended to support the Fun language which has weak pointers.

In general, weak pointers are treated as every other node but they cannot promote nodes to the new semi-space in heap on their own. If they are promoted and point to another promoted object, their pointer value is updated using FWD tag to identify the object's new location. This operation takes place in the end of the scavenging phase, where all live objects are in the new semi-space. If promoted weak pointers point to dead objects, their pointer value is set to NULL.

Bartlett's Algorithm

The Mostly Copying Garbage Collector design was derived by Bartlett, J. F. (1988). To achieve the desired collector behaviour, a class named Page was written. The table below illustrates Bartlett's algorithm, as it was interpreted from the paper.

PHASE 1	Algorithm evacuates the roots : If root points to an object in heap ● Update object's page space to 1 (promotion). ● Add page to linked list (*queue*). ● Update the root.

PHASE 2	Collector scavenges the linked list: ● Traverse all the Pages in Linked list, evacuate the pages that are pointed. ● Do this until all the pages that are referenced by promoted objects have space == 1.
PHASE 3	Collector updates the weak pointers in the linked list and prepares itself for the next Collection : ● Update the weak pointers that have been promoted by traversing the Linked List. ● Set the Linked list as the next old memory. ● Clear linkedList and reset the pointers.

This collector uses page frames to handle the heap. Pages in heap have the same size equal to *PAGE_SIZE.* The collector is aware of the heap structure, traverses the stack and promotes the pages that are referenced by the roots. At that point it is able to distinguish the roots from other values in the stack. The collector will check for ambiguous roots and if the check proves the root's "integrity" it will promote the page simply by changing its space associated value from 0 to 1. The last steps in promoting a page are to insert it in into the tail of a linked list called *queue* in the source code and update the pointer to show at the new location. The new location is calculated using the page index in the linked list.

After evacuating the roots, the collector will scavenge the pages in *queue* to promote any other page that accommodate live objects (objects referenced by "strong" pointers) and it will update the values of the pointers. The algorithm proceeds by traversing the queue and searching for promoted weak pointers to update their value, if and only if, the page that they point to has been promoted. The last phase is to copy the queue to the heap and clear the old memory. The queue can be interpreted as the new space.

Mostly Copying Garbage Collector Benefits & Drawbacks

By using the bartlett technique, the collector collects faster the heap due to less IO transactions. It manages the heap in chunks of size *PAGE_SIZE*, hence it does not copy every object in memory but a whole page or chunk. The algorithms will be compared through the following example:

EXAMPLE

MEMORY INFORMATION
Heap size : 8000 words
Objects : 888
Avg Object size : 4.5 words
Number of Live objects : 600

Available memory : 3996 / 8000 words --> garbage collection is triggered

CHENEY'S TIME REQUIREMENT	**BARTLETT'S TIME REQUIREMENT** page size = 80 Avg number of objects in page = 17.7
600 * IO: cost to copy the objects	**600/80 * IO:** to copy the linked list objects with the live objects
X * 888: Time to traverse through all objects in heap to identify live objects	**X * 888:** Cost to traverse through all objects in pages
X * 600: Time to traverse the new semi-space to update the weak pointers	**X * 600:** Time to traverse the new semi-space to update the weak pointers
Z: cost to update all the weak pointers	**Z:** cost to update all the weak pointers

The Bartlett algorithm performs better due to less copy commands. However, it requires more memory thus, a clarification is given below.

The above collection is not realistic with bartlett's algorithm. Having an average object size of 4.5 words means that pages would be full by a proportion of 76.5 / 80 or 95% or more generally, they will contain on average 17 objects. Hence the algorithm requires at least 888/17 * 80 = 4240 (1) words of free memory, in case of all objects being live. As a result, heap size has to be 8240 instead of 8000. An increment of 3% in space requirement.

*(1) Explanation of 888/17 * 80 -> All the objects fit in 888/17 pages = 53 pages. Each page consumes a size of 80 -> 53 * 80 = 4240.

IMPLEMENTATION

The source code is scattered in different classes. It includes the implementation of Cheney's and Bartlett's garbage collectors. Both algorithms were extended to support weak pointers.

Under src folder :

- *cheneys.HeapBuilder.class* : This class acts as builder for heap. It contains methods that add objects in the heap. Moreover, it includes methods to achieve automated heap generation to execute a variety to test cases.
- *cheneys.CopyingGarbageCollector.class* : Implements the garbage collector. A more detailed description of the key methods is provided below.
- *bartlett.Page* : This class was created to support page frames in heap. Page instances are aware of their free space and the next available position in them. In the source code, page instances have a String array of size PAGE_SIZE which represents the memory from the heap that they occupy.
- *bartlett.BartlettHeapBuilder.class* : This class creates heaps that will be collected by the garbage collector. The heap is simulated as an array of the above cited pages.

- *bartlett.MostlyCopyingGarbageCollector.class :* Provides an implementation of a mostly - copying garbage collector.

Under tests folder :

- *CoypingGCTest.class* : Includes the test cases for Cheney's garbage collector.

- *MostlyCopyingGCTest.class* : Provides the tests to analyse grabage collector behaviour.

I decided to write multiple jUnit tests instead of having a main method that builds heap and run the collectors.

Analysis of CopyingGarbageCollector.class (Cheney's Algorithm)

In the package ofCheney's algorithm exists the *CopyingGarbageCollector.class* file which implements Cheney's garbage collector. The critical methods for the collector are :

- *evacuateRoots()* : Evacuates the roots that are in the stack. It copies the object that is referenced by the root, it inserts a FWD to the oldSpace and updates the root value, to point to the newSpace. This method iterates through all roots in stack. If the root points to a copied object then it is updated according to FWD value.

- *scavenge()* : This method implements the scavenging phase. The collector iterates through all the copied objects in newSpace, evacuating the referenced objects (if any) and updating the pointers values.

- *updateWeakPointers() :* Objects are evacuated (copied) to the new space when they are referenced by root set or objects referenced accessible by roots except for weak pointers. The collector will collect objects that are referenced by weak pointers. To achieve this behaviour, the collector will never evacuate objects that a copied weak pointer points at. When the scavenging phase finishes, the algorithm will iterate through all promoted weak pointers and update their value if they point to a FWD object. If not, it will update them to NULL. Hence, objects that are "weak" referenced will never be collected. Weak pointers which point to a live object, will be updated. This process requires an iteration throughout objects in new space.

- *flip() :* This method flips the old and the new space.

- *clearOldMemory() :* This method clears the old memory (sets all values of the old space in heap to null).

Analysis of MostlyCopyingGarbageCollector.class (BartLett's Algorithm)
In the package of Cheney's exists the *MostlyCopyingGarbageCollector.class* file which implements BartLett's garbage collector. The critical methods for the collector are :

- *evacuateRoots()* : Evacuates the roots that are in the stack. First, it updates the page space bit, secondly adds the page to the tail of the linked list and lastly it updates the pointer value.

- *scavenge()* : This method implements the scavenging phase. The collector iterates through all the pages in the linked lists and evacuates the pages that are referenced by the promoted objects.

- *updateWeakPointers()* *:* This phase is similar to cheneys algorithm. Instead of traversing the new semi-space, it traverses the promoted pages which are contained in the linked list.

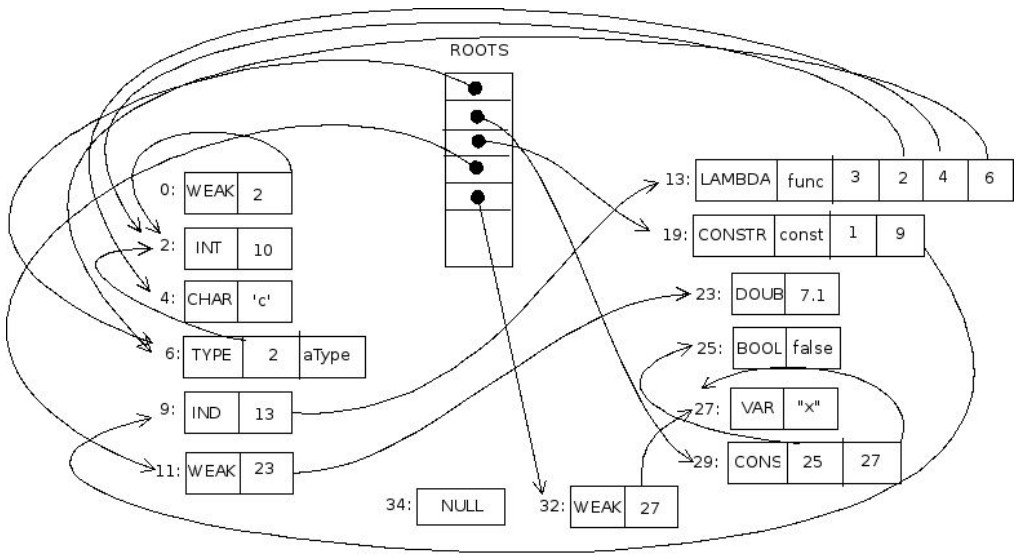
- *copyToNewSpace() :* Copies the linked list to heap.

- *clearOldMemory() :* This method clears the old memory by setting to null all values of the pages having 0 as space bit.

CHENEY'S COLLECTOR TEST CASE

A full example of a garbage collection is illustrated below to clarify Cheney's collector behaviour as it is implemented in the source code.

Heap Image before garbage collection phase (Phase 1):
In the example the heap contains fourteen objects and five roots. The collection will start by evacuating the root sand then the other reachable objects. Graph 1 illustrates the heap image.

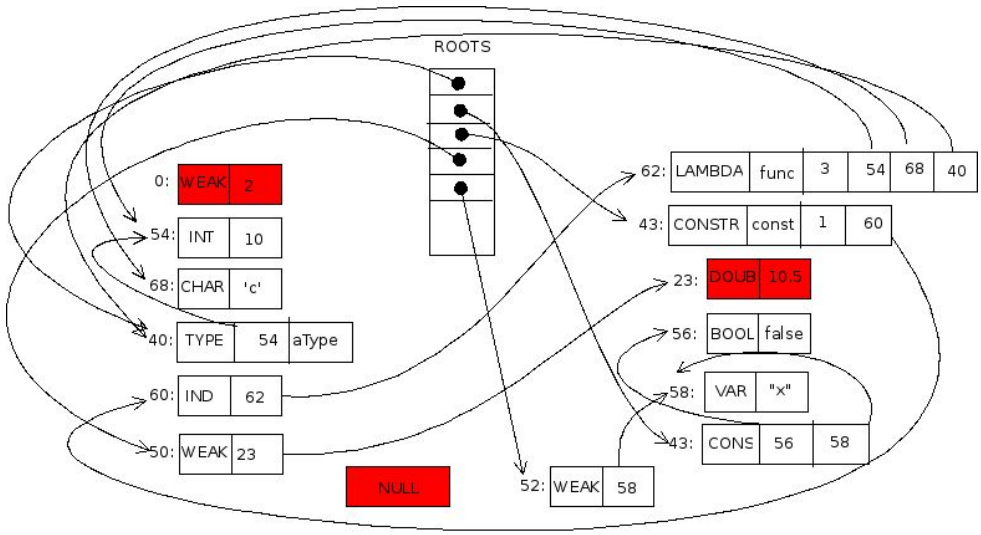


Graph 1. Heap graph before any collection.

Heap Image after first garbage collection (Phase 2)

For this phase a brief explanation will be given to clarify the collector behaviour regarding weak pointers.

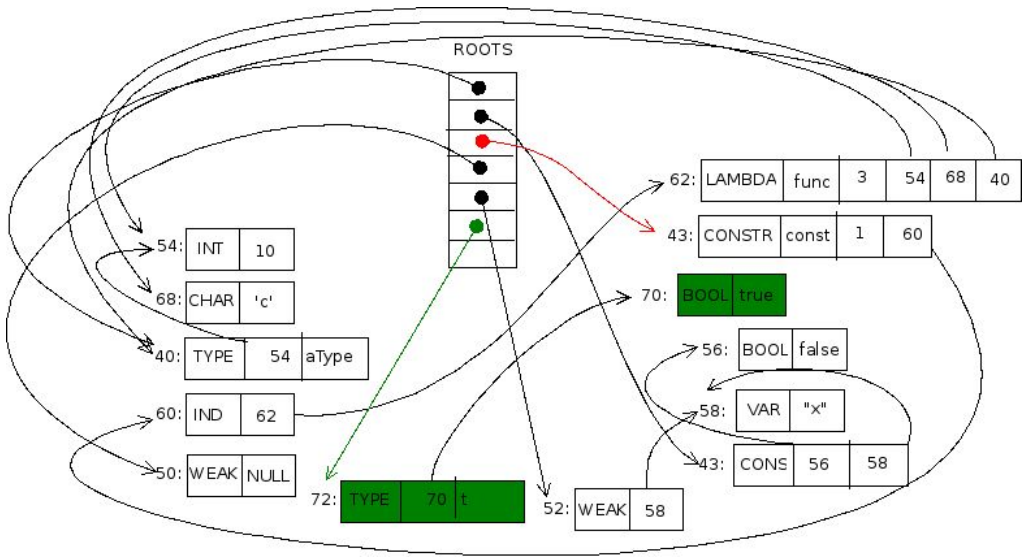
In Graph 2 red objects are considered as garbage. In particular, the weak pointer in heap[0] is not referenced and it will be collected. The weak pointer in heap[50] is live and it is copied to the new space, however the [23:Double] that points does not have any strong pointers so it was considered garbage. Thus, the weak pointer value will be updated to NULL. Lastly, weak pointer in heap[52] is referenced by a root, thus, it is promoted to the new space. In addition, it refers to the node [58:VAR] which has been promoted, as a result of a reference hold by [43:CONS],thus the weak pointers value will be updated to point to the new node location(58).



Graph 2. Heap graph after collection.

Heap Image after Mutation (Phase 3) :

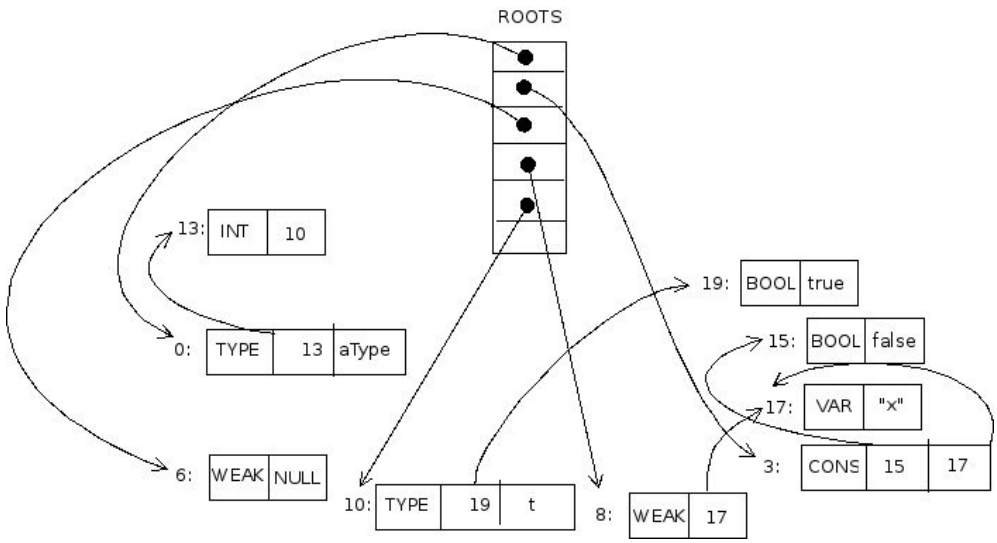
In this phase, application is executed and mutation changes the state of the application. One root is deleted, one added and two objects are created in heap (Graph 3).



Graph 3. Heap graph after mutation.

Heap Image after collection (Phase 4) :

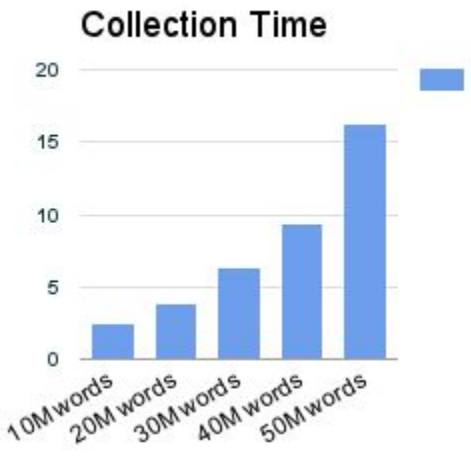
Ih phase 4 the collector is triggered for the second time in this example. The mutation led to different collection results. Graph 4 presents the final applications state.



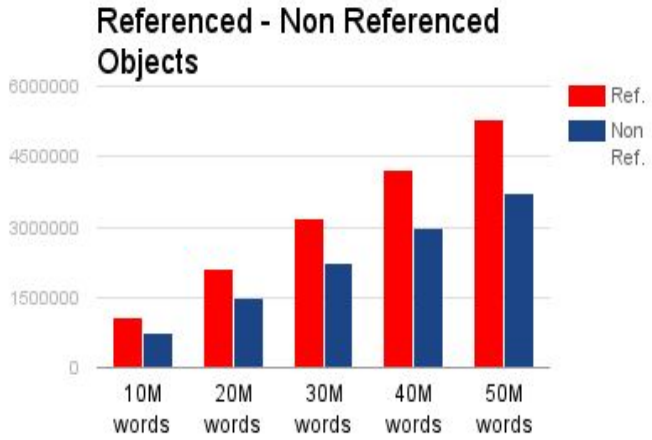
Graph 4: Heap image after collection.

CHENEY' S EXPERIMENTAL TIME COMPLEXITY ANALYSIS

In order to analyse the garbage collector time complexity, an auxiliary class named HeapBuilder.java was written. This class comprises of a method *generateValidHeap()* which takes an integer argument (size of the heap). It creates a heap with random objects of any valid Fun language type. The time analysis will be achieved through charts.



Collection Time Bar Chart : The bar chart illustrates the time in *seconds* that the garbage collection process consumed to clear the memory. There is a linear relationship between the size of the heap and the time required for the algorithm to execute Cheney's algorithm. Overall, there are five test cases. In every test case the size of stack is 10 % of heap size. According to the chart, the relationship between number of objects in heap and the garbage collection process time requirement is linear, O(n).



Reference - Non referenced Bar Chart : The left placed chart highlights the objects that are referenced by pointers directly from the stack and objects that are garbage. The chart data of these five test cases are directly related to the *Collection Time* chart. The red bar illustrates the number of object that are copied to new space, and the blue one those that are considered as garbage.

	Test Case Analysis Table				
	Heap Size	**Stack**	**Objects in heap**	**Ref. objects in heap**	**Collection Time**
1	10M words ~40MB	1M words	1.800.336	1.058.030	2.24 s
2	20M words ~80MB	2M words	3600218	2116518	4.25 S
3	30M words ~120MB	3M words	5.399.630	3.173.483	5.80 s
4	40M words ~160MB	4M words	7.199.688	4.233.491	9.89 s
5	50M words ~200MB	5M words	8.999.517	5.289.600	16.20 s

Size of word is assumed 32 bit or 4 Byte.

BIBLIOGRAPHY

Bartlett, J. F. (1988). Compacting Garbage Collection with Ambiguous Roots. ACM SIGPLAN Lisp Pointers, 1(6), 3–12. https://doi.org/10.1145/1317224.1317225

Cairns, J. L., & Nelson, K. W. (1970). i I, 75(5), 1127–1131. http://dl.acm.org/citation.cfm?id=362798