University of
St Andrews

The School of Computer Science

CS4021

PL Design & Implementation

2015/16

Assessment 2:
Generational Garbage Collection

ID: 150022355

Submitted: 27/11/2015

Tutors: Kevin Hammond

Overall Goal for Our Generational Garbage Collection Program

To optimize this algorithm, the idea was for memory to be managed in generations which simply holds various objects of different ages. Garbage collection occurs in each generation when the generation fills up. Objects are allocated in a generation for younger objects or the young generation, and because of infant mortality most objects die there. When the young generation fills up it causes a minor collection. Minor collections can be optimized assuming a high infant mortality rate. However, as you will find, there is some missing functionality from the code which enables our GC algorithm to promote objects into various '`n++`' generation spaces: `(G0 -> Gn)` ...

To put the following program into perspective, first the program should use some simple heap structure, memory, allocation and object referencing functions. These can then be broken into two generations: '*from-space*' and the '*to-space*'.

The '*to-space*': Most of the newly created objects are located here. The '*from-space*': Objects that did not become unreachable / survived the to-space are copied here. When objects disappear from the to-space, we say a *'minor GC*' has occurred. When objects disappear from the '*from-space*': we say a '*major GC*' (or a '*full GC*') has occurred. There are a number of different algorithms that incrementally perform this procedure. For example, the traditional 'mark and sweep' algorithm, however this lacks any generational procedure. Rather the algorithm, uses traversing methods over unreachable and reachable objects in the heap. This is known as the 'marking phase', - subsequently, the 'sweep' phase occurs soon after. Sweeping takes all the unreachable objects and assigns them too an empty space. Other various algorithms have derived from 'mark and sweep', such as the '*Stop and Copy*' algorithm this program is derived from.

Heap Structure, Allocation & Reachability

Firstly, we must establish our '*heap*', which will contain several uniformed 'boxed' elements. We can pull out elements from the ARC (*Reference Counting*) approach. ARC is used as a simple technique of storing the number of references, pointers, or handles to a resource such as an object, block of memory, disk space or other resource. Essentially, we need somewhere to store our objects and references. Each address at which an object is stored will be located inside our heap array below.

```
var heap = makeHeap(20); // heap of size 20

// Current from-space (working) space.
var FROM_SPACE = 0;
// Start of the to-space space.
var TO_SPACE = 10;
```

Now we can implement our heap divisions. The heap is divided into two spaces for our objects to be assigned in. The '*from-space*' or '*old generation*' is where our currently reachable objects live. The '*to-space*' or '*young generation*' in contrast is initially reserved for GC needs. First half (*0-9 indices*) is '*from-space*', the second half *(indices 10-19*) is the '*to-space*'.

Set up two variables for the two space divisions and their indices. Reasons for doing this are because: In many programs, recently created objects are also most likely to become unreachable incredibly quicker than first thought.

This is known as the infant mortality or the generational hypothesis. With generational GC's it is easier to divide our heap into two divisions for this purpose. Therefore, many generational GC's use separate memory regions for different ages of objects.

Uniform Heap Representation

Example of the source code:

```
// Set up function for creating 'int' with a value of 'n'.
function makeInt (n, heap) {
    // The function returns our impleneted heap stack with the
    // value associated with 'n'.
    return heap.pointAddress ({ intValue: n });
}
// Set up function for creating 'cons' with a value of 'head, tail and
heap'.
function makeCons (head, tail, heap) {
   // The returned value proceeds with a function call implementing
   // our 'head', 'tail' elements.
   return heap.pointAddress ({ head: head, tail: tail });
}
// Set up function to create our uniformed heap. The function returns
// several values which store object, object_array, size and next,
// and the to_space.
function alloc (object, object_array, size, next) {
      // If our next space is smaller and equal to size of heap:
      if (next <= size) {
        // Perform allocation of array to object:
        object_array[next] = object;
        // Return next object:
        return next;
      } else {
        // Let us return failure code, such as 'null':
        console.log(next);
        console.log(size);
        console.log('failed');
        return null;
      }
} // End pointAddress()
```

Our `makeHeap()` function denotes several internal functions which subsequently allocate several functional operations. The idea of this is to enable flexibility, extending our uniformed heap structure beyond just some global elements.

The function passes the variable of '*n*', which is used to allocate a dynamic heap size for our simple array structure initiated further above on line 43. The function returns a number of helper '*tags*' which helps create the 'from-space' and our 'to-space', with an additional '*next_to*' space. The '*next_to*' space stores our objects after they leave the nursery, before the GC algorithm is called upon the objects.

```
[      Old Space         ||  {2}, {4}  |    To Space    ]
[      Old Space         ||   Next To  |    To Space    ]
```

Note: Remember that our '*n*' resembles an abstract representation of the dynamic heap and the number of heap elements. This is why '*n*' is passed in our return elements *'size'*, *'from'* and '*to*' because we want the heap space to capture and assign all of our dynamic elements added onto the stack. The '*next*' variable is used for passing in our objects within a contained space. This needs to be done for incremental purposes.

Our *`pointAddress()`* function is used to return our `alloc()` upon our object. The function is basically used for assigning our objects and storing onto the heap. For whatever object on the heap, it has the function return our initiated elements from `makeHeap()` function `return` above.

The `pointAddressTo()` is used to explicitly keep the 'address' of the object. The heap array index and where the object is allocated. Incrementally this increases the '*next_to*' variable. This allows us to move objects into the '*next_to*' space.

Set Up the Structure of the Heap

We first use the `makeInt()` to create ourselves an integer object. Integer created stores the value 10 and associates itself with the new abstracted heap. This allows us to create a more *'uniformed'* representation of the abstracted heap elements.

```
var z = makeInt(10, heap);
```

Next, we do the same with our `makeCons()` function however we add the value of a standard 'null'. Notice how we also tag '*z*' which is a variable we are storing our integer in.

```
z: {10} -> x: {null}
x: {null} <- z: {10}
```

This particular functional assignment is continued for other various elements. These are subsequently passed along with our '*heap*' variable and added onto our stack representation.

```
var x = makeCons(z, null, heap);
var f = makeCons(makeInt(2, heap), x, heap);
var y = makeCons(makeInt(4, heap), f, heap);
var k = makeCons(makeInt(6, heap), y, heap);
var j = makeCons(makeInt(8, heap), k, heap);
var m = makeCons(makeInt(12, heap), j, heap);
```

The criteria for our GC is too target non-reachable objects. Therefore, we must cut off some root level association path for one of our objects for the GC to efficiently work. This is called further down in the code where we call our GC algorithm: `gc_sc()`;

Stop and Copy GC, Fixing Pointer Issue & Forwarding Pointers

In this moving Stop and Copy GC, all memory is divided into a '*from-space*' and *'to-space*'. Initially, objects are allocated into a '*to-space*' until the space becomes full. Next, a GC algorithm is triggered, below we highlight the *Stop and Copy* GC algorithm which is a more redefined 'mark and sweep' algorithm.

Abstractly, there doesn't need to be any bit-for-bit copy inclusion. *Bit-for-bit* inclusion would enable us to assign more detailed references. Because of this, there would be copying issues. As our abstract representation does not cater for more precise references, therefore some properties of objects may be referencing to other objects. This is unfortunately one of the pitfalls when trying to implement a strict GC in JavaScript, other languages such as C++ would potentially provide more concrete implementation techniques.

After the *'copying'* we should try adjust all pointers and their objects to point to the new space where the objects were copied. A technique which may help us to solve this issue is a '*Forwarding Addresses*' technique. This technique contains a special marker value which we can put on the object when copy it.

At runtime, we can assign and mark the copied objects more efficiently by forwarding addresses. This will update any other objects and their pointers incrementally and we can denote markers accordingly by dividing the to-space into three parts:

- Copied & Scanned objects -> '*from-space*'
- Just copied objects - > '*next_to*'
- The Free space -> *'to-space'*

Our new allocation pointers are set to the boundary of both the '*to-space*' and '*from-space*'. This is ideal because as we mentioned earlier, our heap becomes flexible in size.

Set up some basic variables for our pointer and to be used *'abstractly'* within our GC function and algorithm:

```
POINTER = TO_SPACE;
// And the Scanner pointer is set initially here too:
var SCANNER_POINTER = POINTER;

// Now let's set up a function to connect objects to
// their new locations. Remember, our 'object' is abstract and is
'empty'...
function copyNewSpace(heap, object) {
  // We create a variable to store an empty object array:
  var newCopiedObject = {};
  // Find items in our 'from-space' heap and associated objects we have
created.
  for (var item in heap.from[object]) {
    if (item != 'address' && item != 'forwardingAddress') {
          newCopiedObject[item] = heap.from[object][item];
    }
  }
```

```
  // Now mark the old object as copied or 'assigned':
  heap.from['forwardingAddress'] = POINTER;
  // Next let us return these onto the heap
  // this subsequently increases the pointAddress() pointer.
  // Finally, the function takes a copied object,
  // from 'heap.to' and increments over into 'next.to' heap.
  return heap.pointAddressTo(newCopiedObject);
}
```

Now let's set up a function to check if 'value' is an 'address' marker. For abstraction the function and algorithm uses simplified versioning. We take the address in the heap array using its index (which are numbers). *E.G. address = 1*.

```
function checkAddress(name, value) {
  return typeof value == 'number' && name != 'address' && name !=
'forwardingAddress';
}
```

Now we have the `copy()` function, this is a very important mechanism in the overall functionality of the program. This is our secret weapon and what we use to copy and assign reachable and unreachable objects from one space to another.

```
function copy(x, h) {
     // Important: We need a special marker checker for any
     // of our special 'null' objects.
    if ( x == null) {
        return null;
    }
    // For output reasons, let us console log our address
    // and our value 'x' for which is being passed.
    console.log('address: ' + x);
    // Next, let us assign a variable which marks our object
    // address but inside our 'to-space'. The idea is not to
    // loose track of our 'addresses' and assigned 'values'
    // from one space to another.
    var new_x = copyNewSpace(h, x);
    for (f in heap.to[new_x]) {
        console.log('new address in to space: ' + new_x);
        if (f !== 'intValue') {
            copy(heap.to[new_x][f], h);
        }
    }
    // Finally, we return our 'new_x' variable and this allows
    // us to use this in our GC algorithm.
    return new_x;
}
```

Start The Abstracted 'Stop & Copy' GC Algorithm

Firstly, we must copy *'root'* objects to the new space. We only have one reachable object here and that is the *'root'* object. Therefore, we can do this by assigning our *gc_sc(root)*. Nevertheless, we also need to pass our functional 'heap' variable which allows the GC algorithm (*derived 'stop-and-copy'*) over our abstract heap structure.

Let's copy it to the '*to-space*', by automatically creating a simple for loop that iterates from our 'root' object and traverses down. We are also keeping the scan pointer at its position.

From this, we have now differentiated our scanner and allocation pointers. Our scanner now points to our '*from-space*' heap. For now, we have only the 'root' object. During our scanning, we copy all these sub-objects and mark them as copied as well.

This is done by pushing our 'form-space' objects to the '*new_roots*' empty array. Reasons, for doing this are because we need to keep track of the 'root' objects when swapping objects over spaces. If this was not implemented below, our function would fail because it would not be able to keep track of reassignment in one space to another. Further down the line, this became an issue for trying to implement the generational promotion functions.
This is explained in more detail further down in the code.

```
function gc_sc(root, heap) {
  var SCANNER = heap.from;
  console.log('root: ' + root);
  var new_roots = new Array ();

  for (var i = 0; i < root.length; i++) {
      new_roots.push(copy(root[i], heap));
  }
    // Now let us store and reset spaces.
    // To do this, we call our simple re-assignment
    // heapSwapSpaces() function we initiated at the
    // begining of our code.
    heap.heapSwapSpaces();

    return new_roots;

} // End gc_sc();
```

Mechanism to Promote the newly Assigned Live Objects into Generations

We need a mechanism to copy the live data of one region of memory to a contiguous group of records in another region.

Concrete '*stop-and-copy*' requires copying every live object from the source heap to a new heap before you could free the old one, which translates to lots of memory. With blocks, the GC can typically use dead blocks to copy objects to as it collects. Each block has a generation count to keep track of whether it's alive. In the normal case, only the blocks created since the last GC are compacted; all other blocks get their generation count bumped if they have been referenced from somewhere.

This handles the idea of short-lived temporary objects. Periodically, a full sweep is made - large objects are still not copied (*just get their generation count bumped*) and blocks containing small objects are copied and compacted.

Unfortunately, trying to implement the generation promotion turned out to be incredibly tricky and after several versions of the different code. This had to be re-evaluated and re-designed. Time constraints restricted the program from evolving further.

POTENTIAL IDEA FOR CODE ALGORITHM. THE ALGORITHM WAS GOING TO BE USED FOR JOINING TOGETHER VARIOUS GENERATIONAL FUNCTIONS BELOW BUT WAS NOT USED.

1 - Take objects of similar age. (Current state of heap).
2 - Objects containing similar age : our first heap.
3 - Divide, objects, add to a generational space recursively.
4 - Perform GC_SC algorithm, with each result, add to a new space.

Let's break it down a bit more:

1 - Take all objects in [heap]
2 - Divide and segment objects from [heap]
3 - Push all objects in [heap] into our first generation: G0.
3 - Push objects into older g1, g2 as they survive successive collection cycles.

Ideally, we needed to have some empty arrays to store our objects associated spaces for the generation objects to be passed after GC.

For example:

```
var G_0 = {};
var G_1 = {};
var G_2 = {};
```

However, it seems as though the functional approach would of been easier to implement and it would allow us to adapt to multiple 'n++" generation heaps. `G0...-> Gn`

Starting off, the idea was to have a function which created an empty array of heaps.

For example:

```
function genHeaps() {
      var heaps = {};
      return object[];
}
```

Next the idea was to have a function which would subsequently push the initial heap representation (using the *'makeHeap'*) function. This heap would be pushed into the empty '*heaps*' array we initiated above in the `genHeaps()` function. The function would take the '*n*' variable as it would push per heap.

For example:

```
function createNextHeap(n)  {
      this.heaps.push(makeHeap(n));
}
```

Finally, we needed generation function which takes the 'heaps' array and its current associated objects. Inside this function, it simply looks for objects in a desired array. The idea for this function was for it to connect the dots together and act as a promotional anchor for our generalised GC routine.

For example:

```
function generation(object, array, index) {

       // Pull elements from heap and push into our array.
       if (typeof(object) == "object") {
       for (item in object) {
       // Segment our heap items.
       for (var i = 0; i < item.length;)
            // Then for each item push into array
            i++;
            array[item] = object[item];

            // However, it seems as though the function needed some
            // conditional statements to identify live objects and from
            // here we would be able to implement our GC algorithm 'gc_sc'
            // upon each generational promotion into our heaps array.

            if (objects in set(g0, g1, g2) == "object") {
             continue;
            }

          }
       };

   // Finally, return our array full of items.
   return array;
}
```

Call our generation function passing in our object, desired heap implementation and our key index.

For example:

```
generation(h, heaps, f);
```

Conclusion and Overall Summary

Although the generational aspect of the GC routine is absent, it was easier to describe a potential working algorithm for implementing the generational promotions than programming a fix. It seems as though the program was incredibly close to fulfilling the generational promotion mechanism.

Reasons for not being able to successfully implement the generational promotion is because the `generation()` function does not provide an adaptive solution for an arbitrary amount of generation promotions. Although the heap implementation enables the program to cope for this, there was significant difficulty in designing an efficient bridge between the waterfall of promotional functions:

For example:

```
[  genHeaps()  ->  ||  -> createNextHeap(n) ->  ||  ->   generation()  ]
```

My overall understanding seemed to be initially correct, however my lack of pure functional programming experience let me down on the last hurdle. In hindsight, using the *'stop-and-copy*' GC algorithm was an interesting algorithm to program. It has a more complicated allocation process when compared with the traditional 'mark-and-sweep'. The algorithm has a running time that is *O(R)* with, *R = reachable* and it reclaims memory by moving reachable storage to another 'space' in memory. The idea was to abstractly represent these transitions of 'space' in memory, however it turned out that too much abstraction created errors. I had difficulty when trying to implement a generational approach, mainly because I could not find a successful mechanism for 'copying' and tracking 'reduced storage amounts' over several '*n*++' spaces.

'Stop-and-copy' is considered the faster GC algorithm. We do not have to worry about the post traversal over any '*n*++' heap to reduce and clear data. Which is a great relief, however it would of been interesting to view the runtime if the model had a more distinct generational collection routine.

References:

Byers, R. (2007). Garbage Collection Algorithms. Washington D.C: University of Washington.

Conrod, J. (2009, September 23). A tour of V8 Garbage Collection. Retrieved November 25, 2009, from Jay Conrod: http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection

COS 441 - Garbage Collection Methods - April 2, 1996. (1996, April 02). Retrieved November 25, 2015, from Princeton University: http://www.cs.princeton.edu/courses/archive/spr96/cs441/notes/l14.html

Management, M. (2015, September 01). Memory Management: Glossary: G. Retrieved November 22, 2015, from Memory Management: http://www.memorymanagement.org/glossary/g.html

May Yip, G. (1991). Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments. Palo Alto, California: Western Research Laborator.

Soshnikov, D. (2013, October 19). Non-recursive DFS and BFS. Retrieved November 22, 2015, from GitHub: https://gist.github.com/DmitrySoshnikov/63f9acfac4651da5d21f

Soshnikov, D. (2013, January 27). Reference Counting (ARC). Retrieved November 21, 2015, from GitHub: https://gist.github.com/DmitrySoshnikov/4646658

Soshnikov, D. (2013, Feburary 08). Stop and Copy GC. Retrieved November 20, 2015, from GitHub: https://gist.github.com/DmitrySoshnikov/4736334

Ungar, D. (2003). Generation Scavenging A Non - Disruptive High Performance Storage Algorithm. Department of Electrical Engineering and Computer Sciences. University of California, Berkeley.