— Advanced Compiler Construction —

Implementing a Copying Garbage Collector using Tagging

Nathalie Casati

May 25, 2008

1 Introduction

In this report, I describe the necessary steps to implement a copying garbage collector using tagging of integers in a virtual machine.
Some benchmarks of this and other garbage collectors are also presented in section 4, on page 6.

2 Tagging

Non-conservative garbage collectors need to be able to distinguish integers from pointers at runtime in order to update references. Tagging consists of using the least significant bit of a word to differentiate types from each other. Since in our virtual machine all pointers are constrained to be word-aligned, it suffices to ensure that integers are odd by mapping n to $2n + 1$.
As this change of representation can be performed in the compiler, this method is reasonably efficient. However, the virtual machine has to be modified in order support tagging.

2.1 Modification of the compiler

Basically, the only thing to do is to tag integers before loading them into registers with the help of the `LINT` instruction. `LINT` is also used to load labels (addresses) but this is done in another function, which we don't modify.
Instructions are emitted in `Code.scala`, where I changed

```
def emit(op: LINT.type, r: Register, i: Int): Unit =
    emit(new InstructionRC(op, r, i))
```

into

```
def emit(op: LINT.type, r: Register, i: Int): Unit =
    emit(new InstructionRC(op, r, (2*i)+1))
```

However, I also had to modify `Generator.scala` because a few `emits` of `LINT` are hardcoded there. Still, the idea of the change remains the same.

2.2 Modification of the virtual machine

I chose to use the threaded version of the engine, since it is more efficient.
Support for tagging can be added by changing the way instructions are executed. In the following sections, I describe how to achieve this.

2.3 ADD

Two cases must be identified and treated separately:

- In this implementation we don't do pointer arithmetic, hence adding a pointer to an integer never happens. If both operands are pointers, their sum doesn't mean anything. However special care must be taken if one or both of them are `R0`, i.e. the null pointer. This can happen since the instruction `ADD` is currently used to copy registers. In that case I simply add them.
- When both operands are integers, I remove their tag before adding them and tag the result.

+	ptr	int	`R0`
ptr	∅	∅	ptr
int	∅	int	int
`R0`	ptr	int	`R0`

2.4 SUB

Just as for `ADD`, there are two cases:

- When the last operand is `R0`, the result of the subtraction is the second operand.
- When both operands are integers, I remove their tag before subtracting them and tag the result. I do the same when subtracting an integer from `R0`.

-	ptr	int	`R0`
ptr	∅	∅	ptr
int	∅	int	int
`R0`	∅	int	`R0`

2.5 MUL, DIV and MOD

When multiplying, dividing or reducing modulo a number, we know we are always dealing with integers. It is thus sufficient to remove their tags before performing the computation and tag the result.
However, checks for division by zero must be adapted because 0 is tagged as 1.

2.6 ISLT, ISLE and ISEQ

When executing comparison instructions, we need to distinguish two cases:

- If both operands have the same type, the comparison is valid and the returned value is the tagged version of either 1 or 0 (true or false).
- When operands are of different type, comparison is allowed only if one of the operands is `R0`. This implies the other operand is an integer. Thus `R0` must be tagged before comparing the operands. The result is like before.

comp	ptr	int	`R0`
ptr	int	∅	int
int	∅	int	int
`R0`	int	int	int

2.7 JMPZ

The conditional jump compares 0 with an operand to decide whether the jump will be performed. It is sufficient to remove the tag before the comparison for the instruction to execute correctly. In effect, if the operand is zero, removing the tag has no effect. If it is one, we know that it is the tagged version of 0, thus we need to remove the tag. For all other values, removing the tag won't make the operand equal to zero, so the condition won't be changed.

2.8 LOAD and STOR

The last operand of a `LOAD` or a `STOR` is an offset, which is an integer. Hence, we need to remove its tag. The second operand is an address, so it must not be changed. The result of the `LOAD` doesn't need to be changed because other instructions will take care of its type.

2.9 ALOC

The last operand of an `ALOC` is an integer, thus its tag needs to be removed. The result of this instruction is an address, so we keep it as is.

2.10 RCHR

When reading a character, we have to tag it, since it is represented by its ASCII code, which is an integer.

2.11 PCHR

When printing a character, we need to remove its tag for the same reason as above.

2.12 LINT and HALT

No changes need to be done here.

3 Copying Garbage collection

In this section I present Cheney's copying garbage collection algorithm and how to implement it.

3.1 Cheney's algorithm

Cheney's algorithm consists of splitting the heap into two semi-spaces, referred to as *from-space* and *to-space*. Objects are allocated linearly in the *from-space* until it is full and a garbage collection is needed. Live objects are then copied to the *to-space*. Before the program execution is resumed, the roles of the spaces are swapped.

In a more detailed way the garbage collection will be performed as follows.
Starting from the roots (in our case the registers which always contains live objects), the algorithm will perform one of these two actions :

- If the current object has not already been copied, linearly allocate an identical object in the *to-space*. In order to remember that this object has been treated, a pointer to the new object is left in the *from-space*, overwriting the header of the current object. This pointer is called a *forwarding pointer*. Additionally, the object reference must be updated to the copied object.
- If a *forwarding pointer* is encountered the object has already been copied. All that needs to be done is updating the object reference to point to the new object in the *to-space* by assigning the *forwarding pointer* to it.

Once this is done, only the objects directly referenced by the roots have been copied. In order to complete the garbage collection, one should linearly scan the *to-space* thus performing a breadth-first search avoiding infinite loops. This technique requires an additional pointer referred to as the *scan* pointer.
The second part of the garbage collection is performed as follows.
One by one, objects in the *to-space* are linearly scanned for references. The pointed objects (which are located in the *from-space*) are known to be live and the algorithm will perform the same two actions to copy them as described above. Every time an object is fully scanned, the *scan* pointer will be used to remember all its references have been followed. Once the *scan* pointer reaches the end of the allocated memory in the *to-space*, the garbage collection is complete, the roles of the spaces are exchanged and the program can be resumed.

More details about Cheney's algorithm or other garbage collection methods can be found in [1, 2].

3.2 Implementation choices

A few implementation choices must be made in order to be able to differentiate an already copied object from another. A way to find the size of objects is also needed. Since objects are allocated linearly, this can be done using a header, located just before the object data. This header will contain the size of the object including the size of the header. Once an object is copied to the *to-space*, its header will contain the *forwarding pointer*. A way to distinguish the *forwarding pointer* from the size is to encode the latter as size + 1 since both are represented as even integers in our virtual machine implementation. When we encounter an odd integer, we are sure it is a size.

3.3 Allocating the memory

Allocating memory when using a copying garbage collector is not a big deal since it can be allocated linearly in the *from-space*. After having assigned the header to the size of the object including the header itself, a pointer to the object data is returned. However, care must be taken to zero memory blocks before allocating them. In effect, they will contain old pointers which are going to be followed during the garbage collection and results in freeing almost no memory.
To keep track of where we will allocate the next object a *free* pointer is used which will point to the part of memory that is still free. When the *free* pointer reaches the end of *from-space*, a garbage collection is launched. If no memory has been freed, the program stops with a "*no more memory*" message.

3.4 Implementing Cheney's algorithm in the context of a virtual machine

In this section I will use pseudocode to show how to implement Cheney's algorithm. I chose to split the work between two procedures: Procedure 1 which will be called on each root iteratively in order to copy objects that are directly references by them. Then, Procedure 2 will scan the *to-space* for references to *from-space*.

Procedure 1 CHENEY's procedure to copy data to *to-space* and deal with *forwarding pointers*

Input: The *free* pointer, initialized to the start of *to-space* at the beginning of the garbage collection and *obj_hdr_ptr*, a pointer to the header of the object to copy

```
1: if obj_hdr_ptr is a pointer to from-space then
2:     if obj_hdr is odd (it is the size of the object) then
3:         copy obj to to-space, at the location pointed by the free pointer
4:         update obj_hdr_ptr to the new location of obj in the to-space
5:         increase the free pointer by the size of obj
6:         leave a forwarding pointer in obj_hdr
7:     else // obj_hdr is a forwarding pointer
8:         update obj_hdr_ptr to where obj_hdr points
9:     end if
10: end if
```

Let's first analyze Procedure 1. We need to be sure (line 1) that *obj_hdr_ptr* is a pointer (because it could be an integer) and that it effectively points to *from-space* (because it could point to the code, which we don't want to copy nor modify). To ckeck the first condition it suffices to verify that `and`ing *obj_hdr_ptr* with the mask `0x1` gives `0x0`. In that case, *obj_hdr_ptr* is even and thus it is a pointer. Checking the second condition is just a matter of verifying that *obj_hdr_ptr* is inside the *from-space* address range.
In line 2, we perform a check to see if the object hasn't already been copied, hence we want to check that the header of the object contains its size. This can be done by checking it is odd (because we encoded it as $n + 1$ when allocating the corresponding memory block), which can be done by `and`ing it with the mask `0x1` and verify we get `0x1`.
If these conditions are met, we can copy the object to *to-space* and update *obj_hdr_ptr* to the new location of the copied object (lines 3-4). In line 5, the *forwarding pointer* is simply the new value of *obj_hdr_ptr*.
Finally, if we encounter a *forwarding pointer* in the header of the object, we just update the reference to it (lines 7-8).

After all roots have been processed by Procedure 1, Procedure 2 is run. Its aim is to scan objects in *to-space* for references that still point to *from-space*. These need to be updated to point to the location of the referenced objects in *to-space* if they have already been copied (otherwise they must also be copied). This is exactly what Procedure 1 does. Thus Procedure 2 will simply call Procedure 1 on each cell of each object of the *to-space* (lines 2-3), until there are no more objects (line 1). In line 5, they *scan* pointer is increased by the size of the object that is currently being scanned. When it meets the *free* pointer, we know that the garbage collection is over (this is what *next_obj*() does).
Finally the two semi-spaces are swapped[1] and program execution is resumed.

Procedure 2 Scanning the *to-space* for object references

Input: The *scan* pointer, initialized to the start of *to-space*

```
1: while $obj = next\_obj()$ do
2:     for all $cell$s in $obj$ do
3:         CHENEY($cell$)
4:     end for
5:     $scan$ += $obj\_size$
6: end while
```

4 Benchmarks

In this section you will find a few notes about the performance we can achieve with a “naive” implementation of Cheney’s algorithm for copying garbage collection – as presented above – compared to a conservative garbage collector, in this case using a Mark & Sweep algorithm.
All the following benchmarks have been performed on an Intel Core2Duo processor running at 2.33GHz using a 64-bit version of Linux. The heap size of the virtual machine is 1000000 (simple) or 2000000 (double) bytes. Timings measure full executions of programs.

4.1 `Maze`

`Maze` is a program that computes a maze according to a given size and seed value. For each seed another maze is obtained.
In graph 1, you can see that the Mark & Sweep garbage collector and the copying garbage collector with simple heap size are only able to execute mazes (15,2), (16,1) and (17,22)[2] even though the Mark & Sweep garbage collector has the whole heap to allocate objects. This happens since it has also a free list and a bitmap to store. Moreover, maintaining these structures takes time, as can be seen in graph 1. The copying garbage collector with double heap size can execute all the tested mazes and is faster since fewer garbage collections are needed.

[1]One single bit can be used to remember which space is the current *from-space*/*to-space*. Swapping can be done by `xor`ing this bit with `0x1`.

[2]This notation stands for (size,seed).

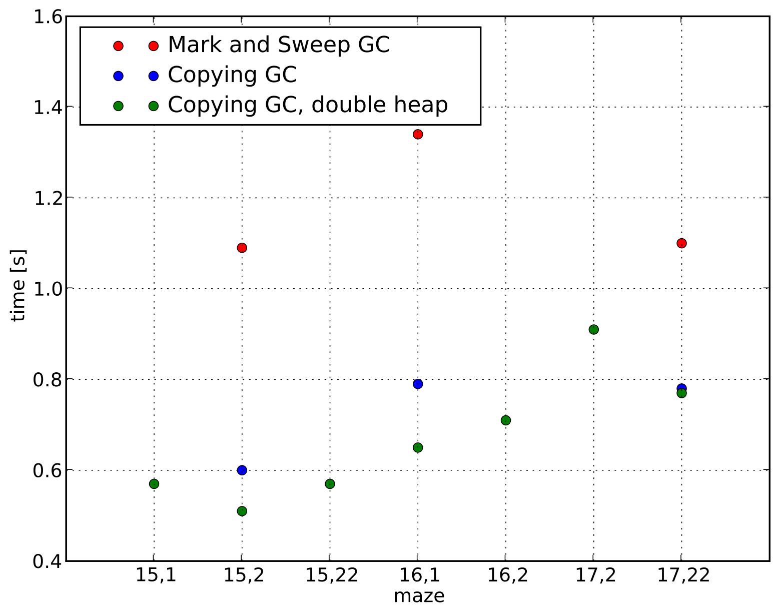

Figure 1: Maze

4.2 `Bignums`

`Bignums` computes the factorial of big numbers. The benchmark starts at 2000.
The same effects can be observed as with `Maze`: Mark & Sweep garbage collector and copying garbage collector with simple heap size cannot execute the biggest benchmarks due to lack of memory while the copying garbage collector with double heap size is able to and is also faster.

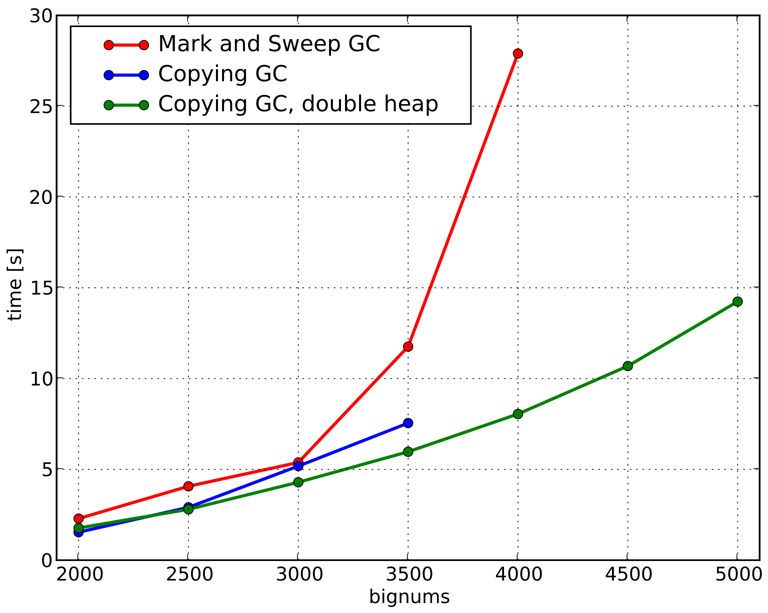

Figure 2: Factorial

4.3 `Fib`

`Fib` computes the `N`th Fibonacci number.
In this situation, the heap contains almost only garbage, thus all three garbage collectors perform the garbage collection nearly equally fast. Since after garbage collection most of the heap is free, the program doesn't run out of memory.

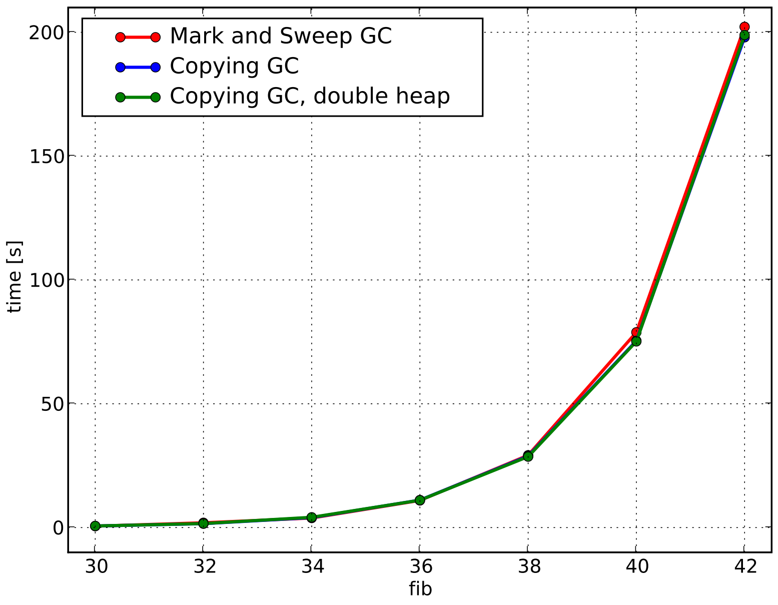

Figure 3: Fibonacci

5 Conclusion

Copying garbage collectors are better for speed but need more memory. They are also quite simple to implement, as opposed to Mark & Sweep garbage collectors that need to manage a free list and bitmap table. However, copying garbage collectors need a way to distinguish pointers from integers and thus modifications of the virtual machine and possibly the compiler have to be made.

References

[1] *Cheney's algorithm tutorial*, University of Maryland, Department of Computer Science, `http://www.cs.umd.edu/class/fall2002/cmsc631/cheney/cheney.html`

[2] *Paul R. Wilson*, Uniprocessor Garbage Collection Techniques