LET THE GARBAGE CRASH

Filipe Varjão
Developer @ Erlang Solutions

@filipevarjao
filipe.varjao@erlang-solutions.com

Why Garbage Collection?

- Facilitates software development.
- Increases robustness.
- Efficiency at runtime.

Memory management

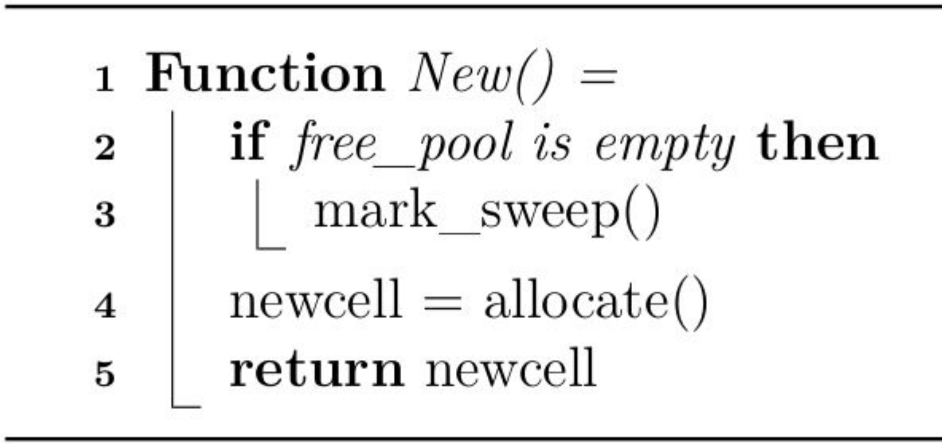
- Mark-Sweep
- Reference counting
- Copy

Mark-Sweep

- J. McCarthy (1960)
- Lisp

Mark-Sweep

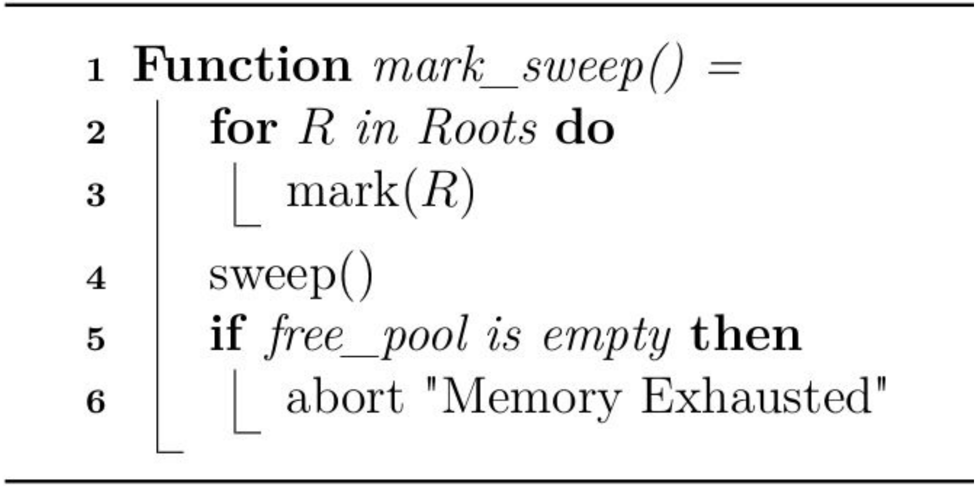
- **Allocate**



```
Function New() =
    if free_pool is empty then
        mark_sweep()
    newcell = allocate()
    return newcell
```

Mark-Sweep

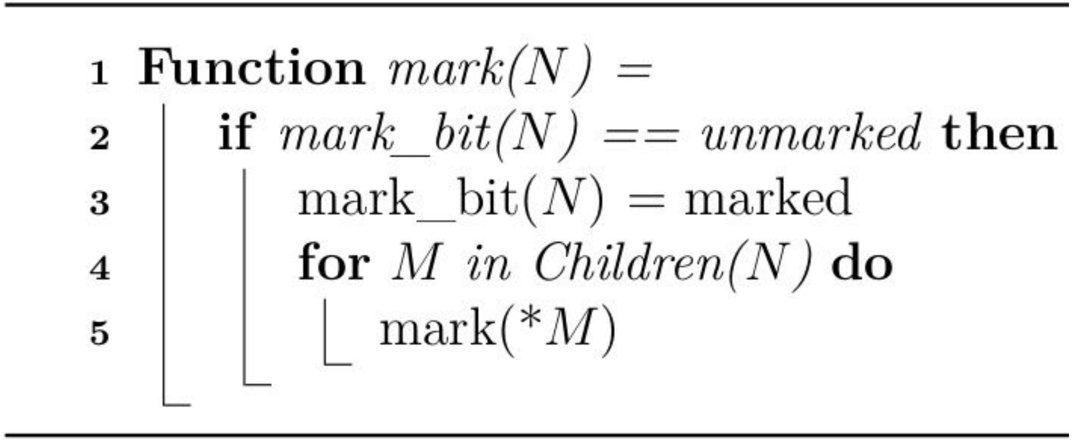
- **Collect**



```
Function $mark\_sweep() =$
    for $R$ in $Roots$ do
        mark($R$)
    sweep()
    if $free\_pool$ is empty then
        abort "Memory Exhausted"
```

Mark-Sweep

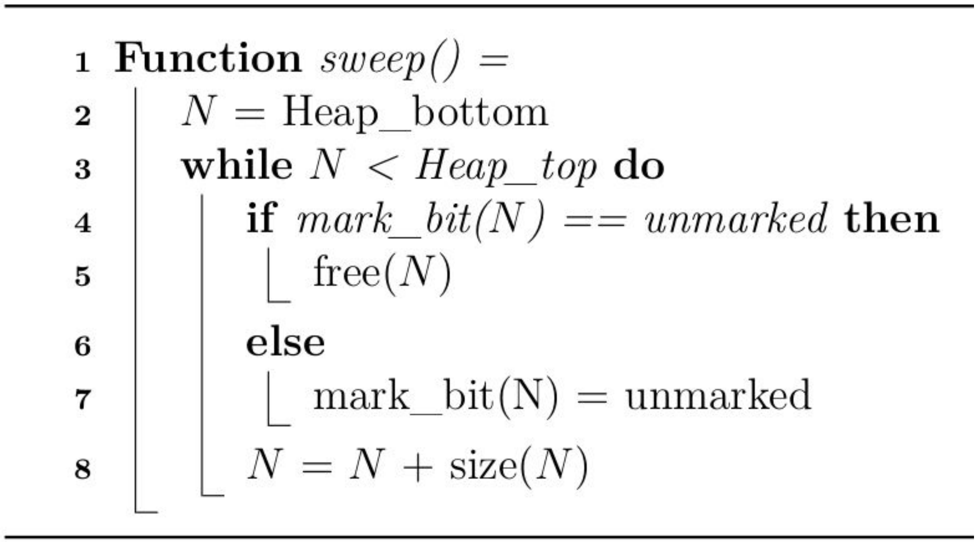
- **Mark**



```
Function $mark(N)$ =
    if $mark\_bit(N) == unmarked$ then
        mark_bit($N$) = marked
        for $M$ in $Children(N)$ do
            mark($*M$)
```

Mark-Sweep

- **Sweep**



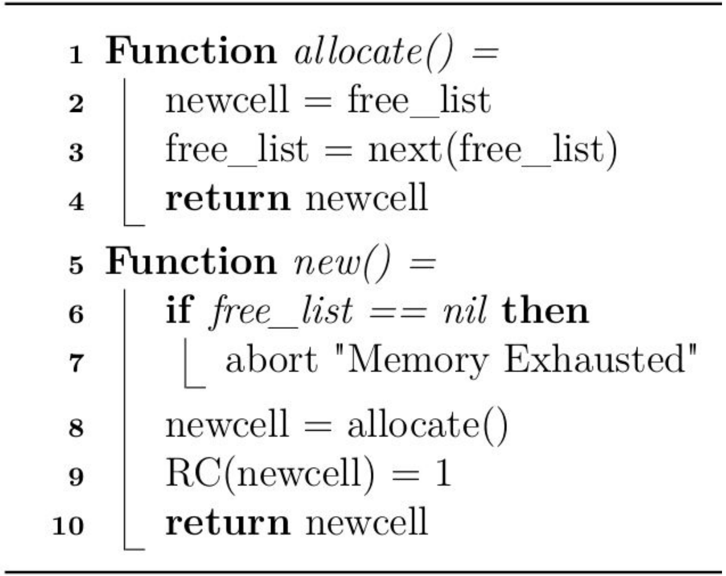
```
Function sweep() =
    $N$ = Heap_bottom
    while $N < Heap\_top$ do
        if $mark\_bit(N) == unmarked$ then
            free($N$)
        else
            mark_bit(N) = unmarked
        $N = N + \text{size}(N)$
```

Reference Counting

- Collins (1960)

Reference Counting

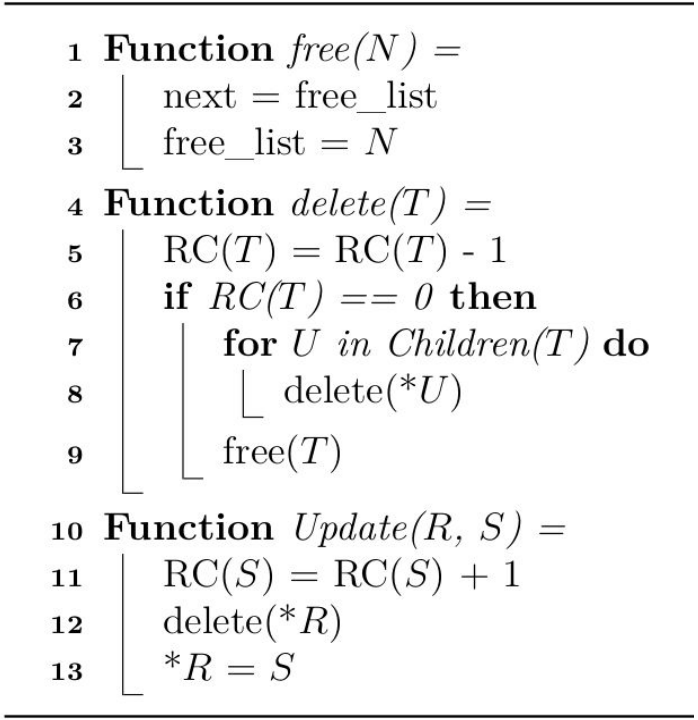
- Allocate



```
Function allocate() =
    newcell = free_list
    free_list = next(free_list)
    return newcell
Function new() =
    if free_list == nil then
        abort "Memory Exhausted"
    newcell = allocate()
    RC(newcell) = 1
    return newcell
```

Reference Counting

- Counting



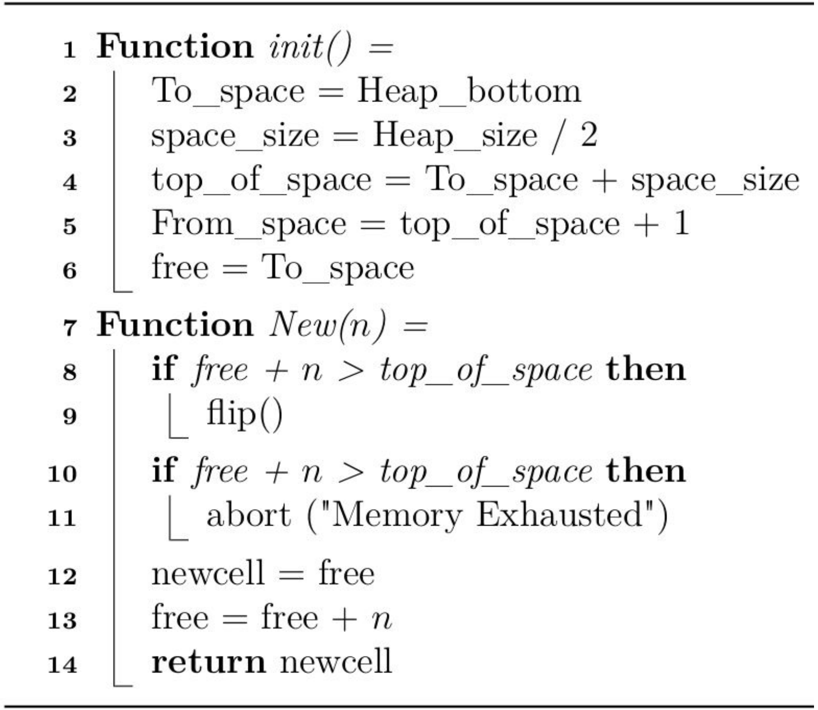
```
Function free(N) =
    next = free_list
    free_list = N
Function delete(T) =
    RC(T) = RC(T) - 1
    if RC(T) == 0 then
        for U in Children(T) do
            delete(*U)
        free(T)
Function Update(R, S) =
    RC(S) = RC(S) + 1
    delete(*R)
    *R = S
```

Copy

- First version - Marvin Minsky (1963)
- Fenichel and Yochelson (1969)
- Cheney (1979)

Copy

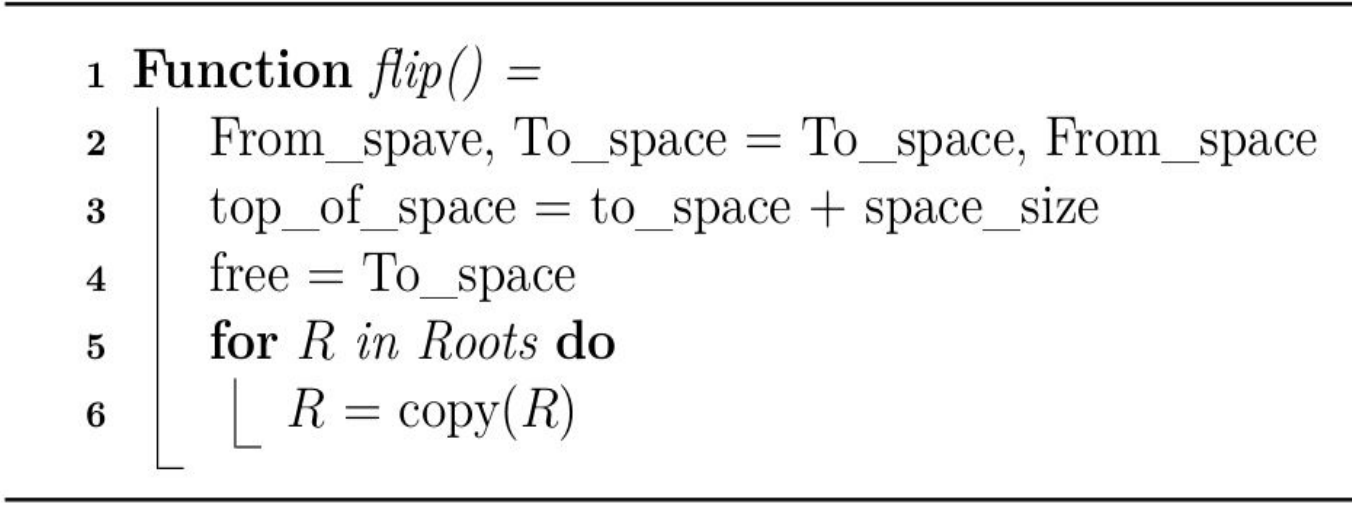
- **Allocate**



```
Function $init()$ =
    To_space = Heap_bottom
    space_size = Heap_size / 2
    top_of_space = To_space + space_size
    From_space = top_of_space + 1
    free = To_space
Function $New(n)$ =
    if $free + n > top\_of\_space$ then
        flip()
    if $free + n > top\_of\_space$ then
        abort ("Memory Exhausted")
    newcell = free
    free = free + $n$
    return newcell
```

Copy

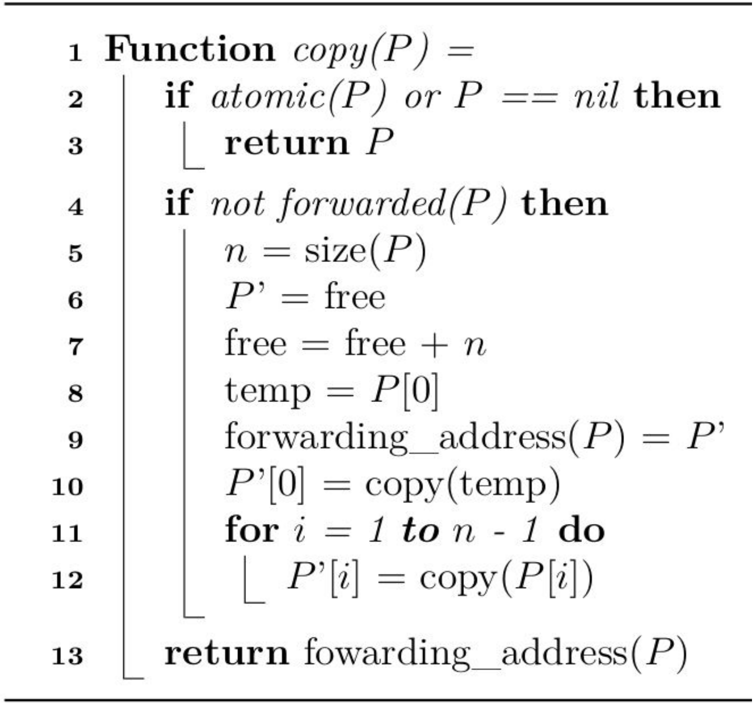
- **Copying**



```
Function flip() =
    From_spave, To_space = To_space, From_space
    top_of_space = to_space + space_size
    free = To_space
    for R in Roots do
        R = copy(R)
```

Copy

- **Collecting**



```
Function $copy(P)$ =
    if $atomic(P)$ or $P == nil$ then
        return $P$
    if not $forwarded(P)$ then
        $n$ = size($P$)
        $P$' = free
        free = free + $n$
        temp = $P[0]$
        forwarding_address($P$) = $P$'
        $P$'[0] = copy(temp)
        for $i$ = 1 to $n$ - 1 do
            $P$'[$i$] = copy($P[i]$)
    return fowarding_address($P$)
```

Golang

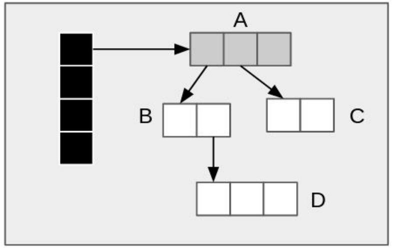
- Robert Griesemer, Rob Pike and Ken Thompson (2007)
- Google
- Hoare - Communicating Sequential Processes (1978)

Erlang

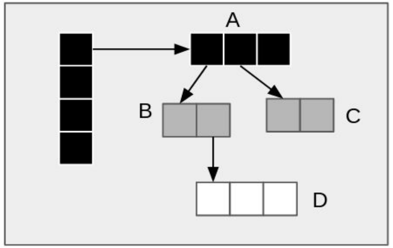
- Joe Armstrong, Robert Virding and Mike Williams (1986)
- Ericsson

Golang memory management

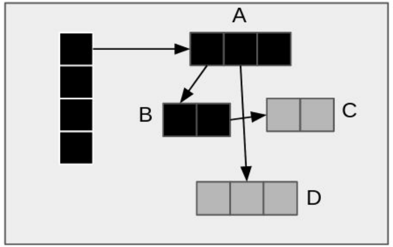
- Mark-Sweep
- Tri-color algorithm (Dijkstra, 1978)



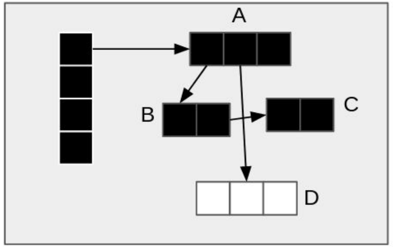
1



2



3



4

Erlang memory management

- Copying semi-space garbage collector
- Generational GC

Memory management

Erlang

- Process heap
- Atom (table)
- ETS (table)
- Binary

Golang

- Objects might live in heap or stack (escape analyses)
- Compiling analyses

Memory management

Golang

- The Go’s compiler will first try to put it on stack but, often it cannot.
- The object is reachable from heap, e.g. assigned to a global
- Object's lifetime is beyond the current function, e.g. x is returned to caller

Memory management

Golang

- An object is sized, e.g. the backing store of slice/string
- When an object is passed to another function which makes x escape, e.g. call f(x) where f does "global = x"
- There is not enough information to decide if it doesn't escape, e.g. passing as arg to function pointer calls

Memory management

Erlang

- Atoms are never deleted.
- The EVM creates a fixed size to all atoms, system crashes when it gets full.

Memory management

Erlang

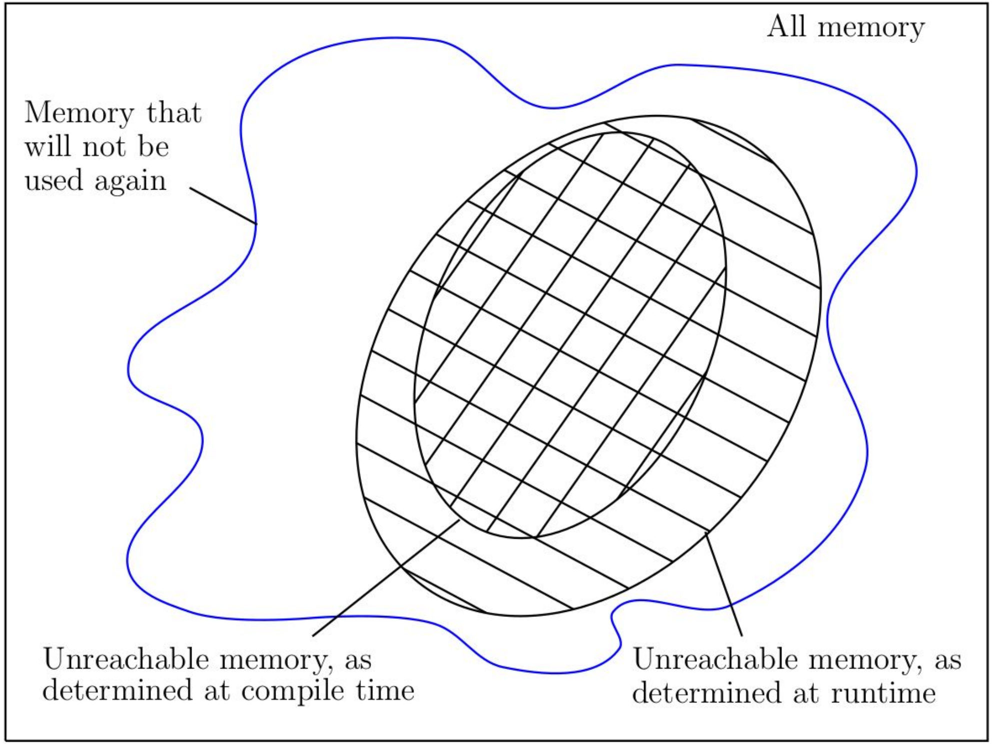
- Small binaries (< 64 bytes)
- Large binaries
- ETS tables are totally isolated where items are copied from/to process heaps

Memory management

Erlang

- Process lifetime is short

Region-Based Memory management



Davis (2015)

Memory management

- Region-Based Cyclic Reference Counting
- gitlab.com/filipevarjao/go-rbmm-refcount

Experiment

Setup

name	CPU	Memory
M1	AMD Phenom II X6 1090T 3,2Ghz	4*GB*
M2	Intel i7-3770, 3,4Ghz	8*GB*
M3	Intel i7-855o*U* 1,80Ghz	16*GB*

** Ubuntu Server 16.04 LTS(Xenial Xerus) 64-bit*

Benchmark Pause - M1

collector	*exec time*	*worst pause time*	*total memory*	*pauses*
ms	1,54*s*	18920*us*	1,02*GB*	11
rc	1,43*s*	630,45*us*	1,02*GB*	0
rbmm-rc	3,37*s*	68,03*us*	24,77*KB*	0

Memory management

Benchmark Pause - M2

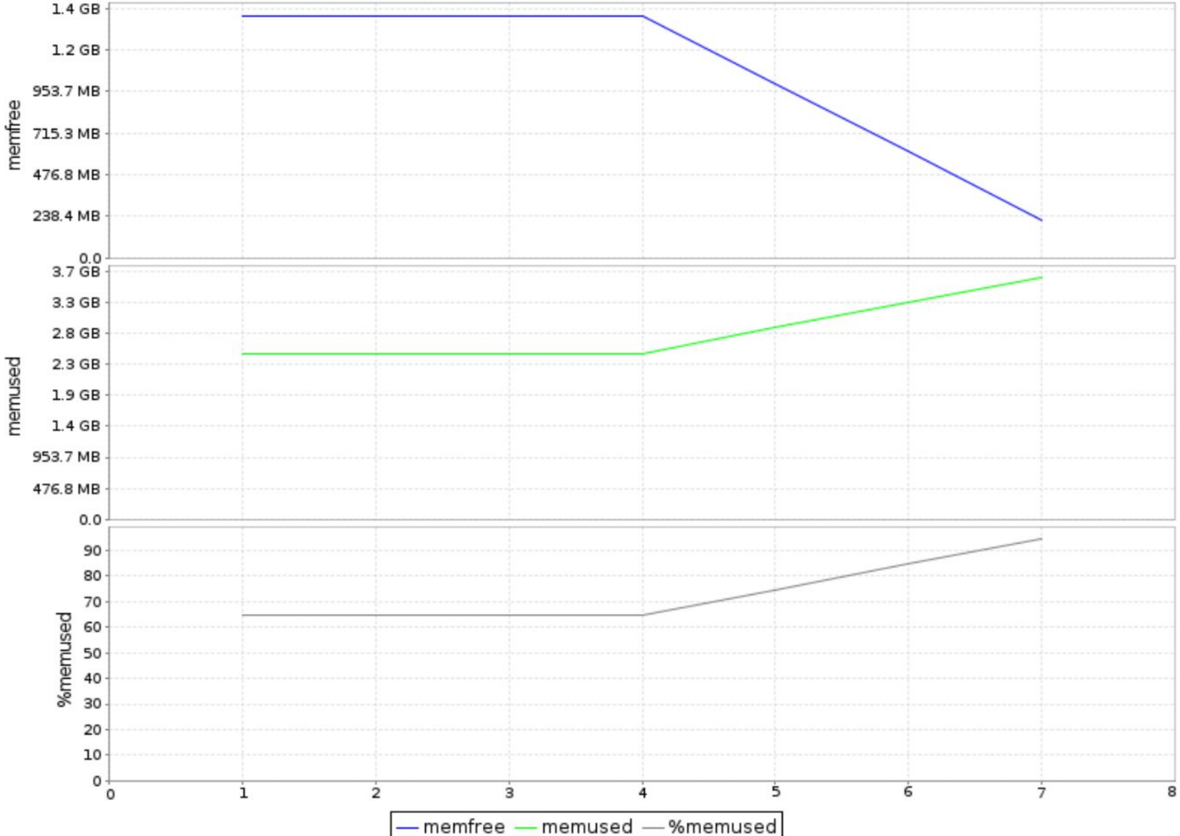
collector	***exec time***	***worst pause time***	***total memory***	***pauses***
ms	668,48*ms*	6220*us*	1,02*GB*	11
rc	645,66*ms*	282,93*us*	1,15*GB*	0
rbmm-rc	295000*ms*	21,97*us*	24,76*KB*	0

Memory management

Benchmark Pause - M3

collector	*exec time*	*worst pause time*	*total memory*	*pauses*
ms	705,97*ms*	5220*us*	1,02*GB*	11
rc	702,63*ms*	297,09*us*	1,02*GB*	0
rbmm-rc	311000*ms*	106,93*us*	24,44*KB*	0

Profiling


memfree
1.4 GB
1.2 GB
953.7 MB
715.3 MB
476.8 MB
238.4 MB
0.0
memused
3.7 GB
3.3 GB
2.8 GB
2.3 GB
1.9 GB
1.4 GB
953.7 MB
476.8 MB
0.0
%memused
90
80
70
60
50
40
30
20
10
0
0
1
2
3
4
5
6
7
8
memfree
memused
%memused

Memory management

Conclusions

- A new concurrent Garbage Collection algorithm.
- Non-suspensive, a feature of paramount importance in real-time systems.
- There is no silver-bullet.

THANK YOU

Q&A

Filipe Varjão
Developer @ Erlang Solutions

@filipevarjao
filipe.varjao@erlang-solutions.com

Erlang
SOLUTIONS