The secret lives of

Garbage Collectors

Jonathan Worthington

Things I work on

EDUMENT
Development and Mentorship

Writing and teaching courses, mostly about software architecture, TDD and C#

Various bits of mentoring and consulting

Perl

Lead developer and architect of the Rakudo Perl 6 compiler; focus on OO, type system, etc.

Various other contributions (native calling, debugger, ...)

So, this is Build Stuff and...

...if I'm going to talk about GCs here, I better have been building one, right?

So, this is Build Stuff and...

...if I'm going to talk about GCs here, I better have been building one, right?

I have been during the last year ☺

For a small VM centred around meta-object programming, as part of my Perl 6 project work

Not just me; ~15 contributors so far. I'm doing both architectural and implementation work, with a focus on the object system and GC

Suddenly, I'm a GC designer/hacker!

Suddenly, I'm a GC designer/hacker!

Already had a reasonable grasp on how GCs work

Had debugged one before, and was quite used to explaining the basics of the .Net one when teaching

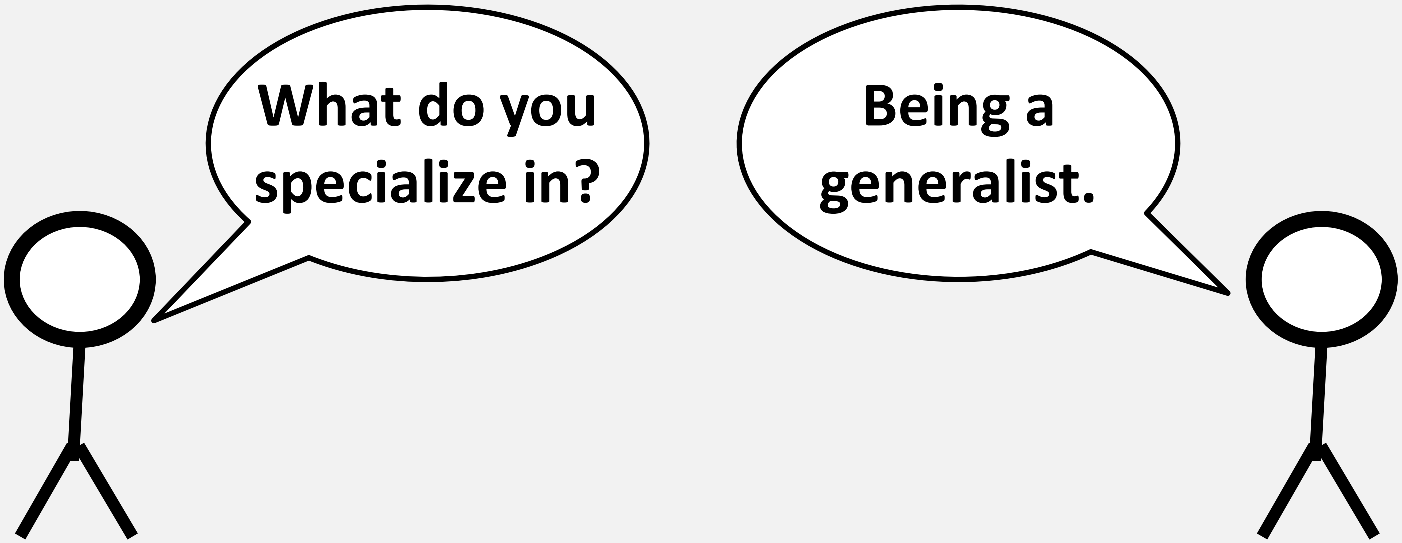
But explaining and doing bug fixes are rather different from doing design

Doing design well means understanding lots of options and being able to make sensible trade-offs

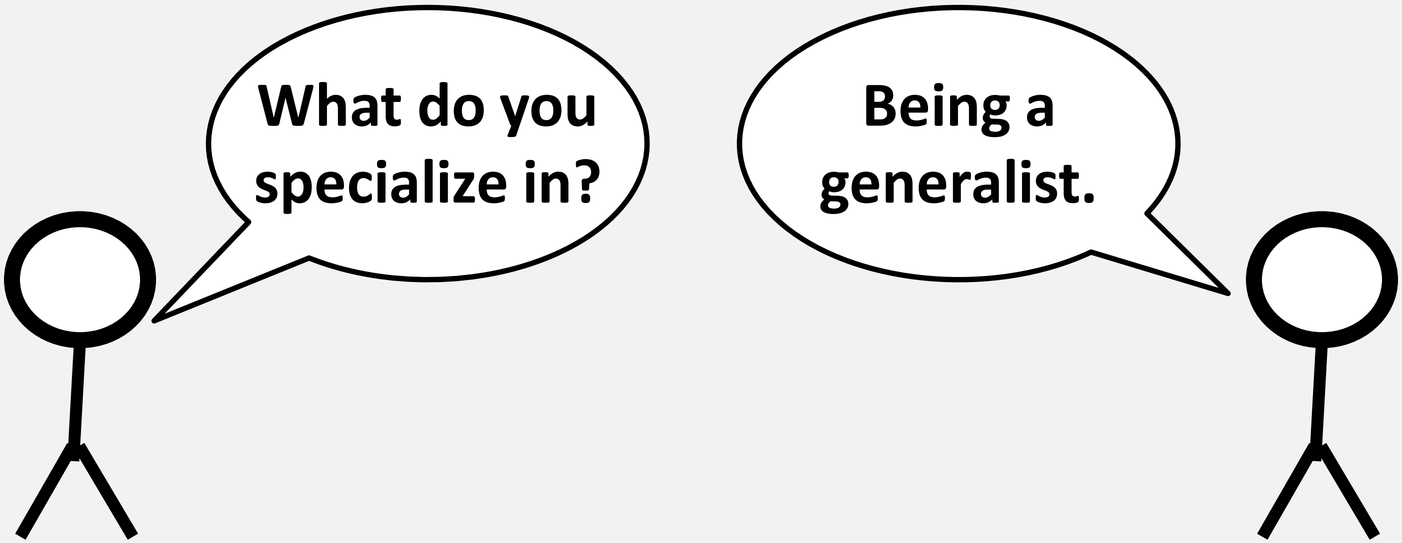
Isn't GC a, like, really specialist area?

Isn't GC a, like, really specialist area?

Isn't GC a, like, really specialist area?



Isn't GC a, like, really specialist area?



GC is a very well researched area. Loads of well-documented algorithms and many decades of experiences to learn from.

I didn't need to invent, "just" select and implement

The bad news

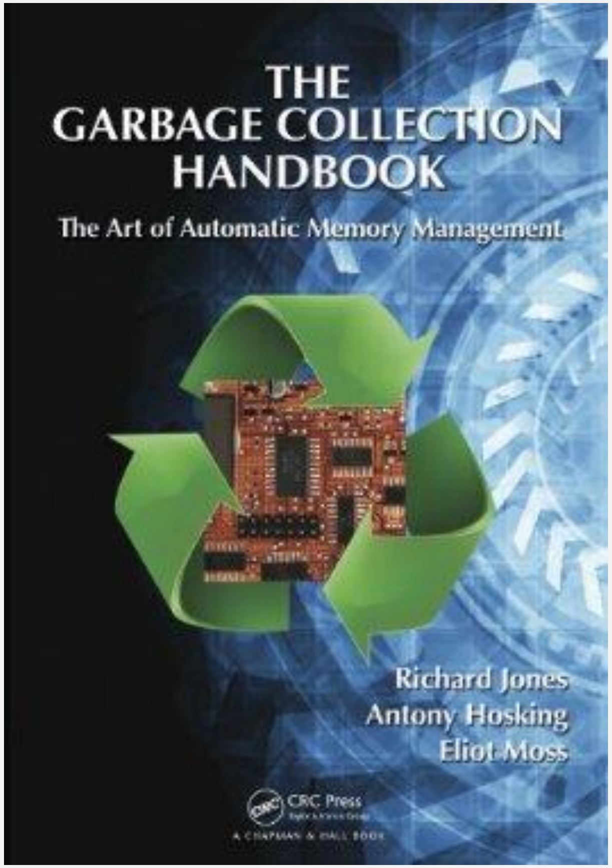
When I design systems, I like to collect concerns into strongly focused, loosely coupled units

Garbage Collection is a real challenge here, because it is interested in memory allocation and even memory accesses - which happens *everywhere*!

Additionally, while many of the algorithms are quite pretty on paper, real world implementation is full of subtleties (threads block, CPU caches can be weird, optimizers and CPUs re-order things...)

I picked a decent time to work on this...

2012 gave us a brand new edition of the leading handbook on Garbage Collection!



A bit over 500 pages, with loads of references

That sounds a lot at first, but it's still 400 pages shorter than the second edition of "Programming Entity Framework"...

So, the basics...

As we execute code, we allocate objects

```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
        badger mushroom mushroom');
my @top_5;
# ...
```

So, the basics...

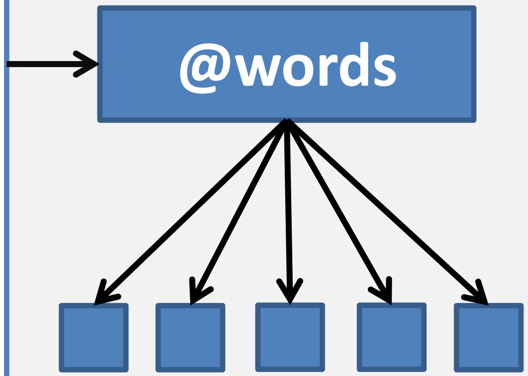
As we execute code, we allocate objects

```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
       badger mushroom mushroom');
my @top_5;
# ...
```

So, the basics...

As we execute code, we allocate objects

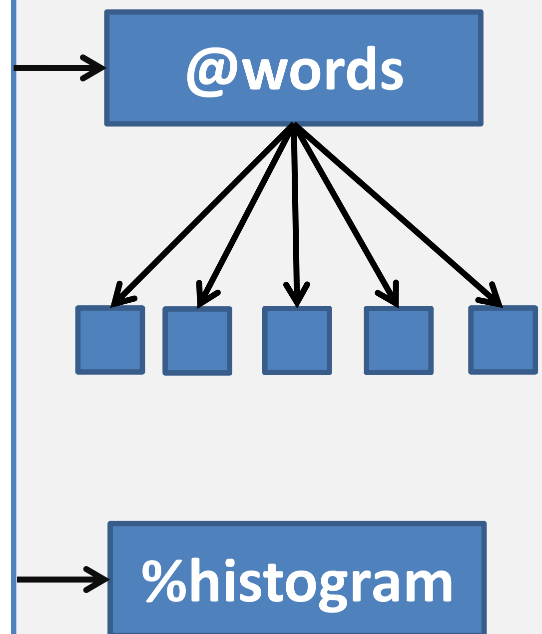
```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
       badger mushroom mushroom');
my @top_5;
# ...
```



So, the basics...

As we execute code, we allocate objects

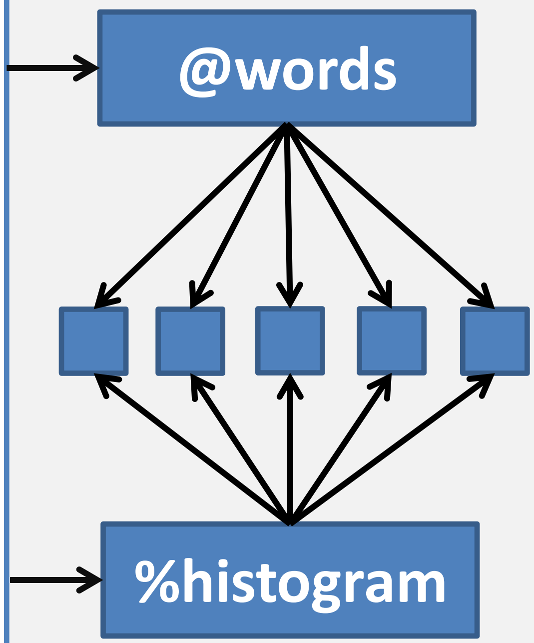
```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
      badger mushroom mushroom');
my @top_5;
# ...
```



So, the basics...

As we execute code, we allocate objects

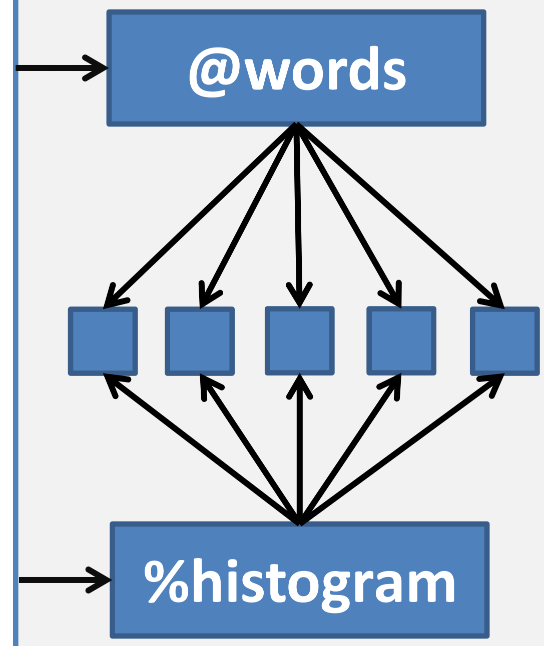
```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
        badger mushroom mushroom');
my @top_5;
# ...
```



So, the basics...

When we return, some things go out of scope...

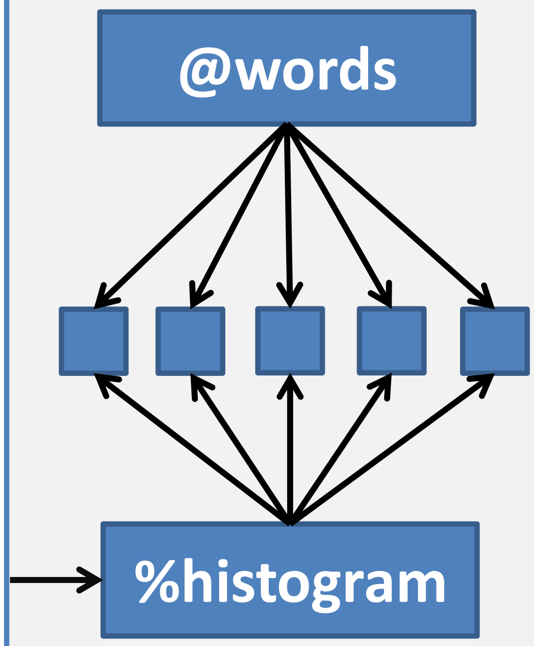
```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
       badger mushroom mushroom');
my @top_5;
# ...
```



So, the basics...

When we return, some things go out of scope...

```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
        badger mushroom mushroom');
my @top_5;
# ...
```

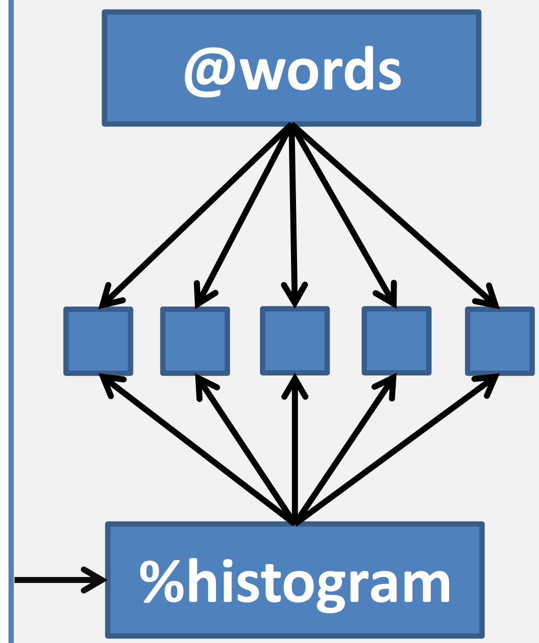


So, the basics…

At some point, we can allocate no more

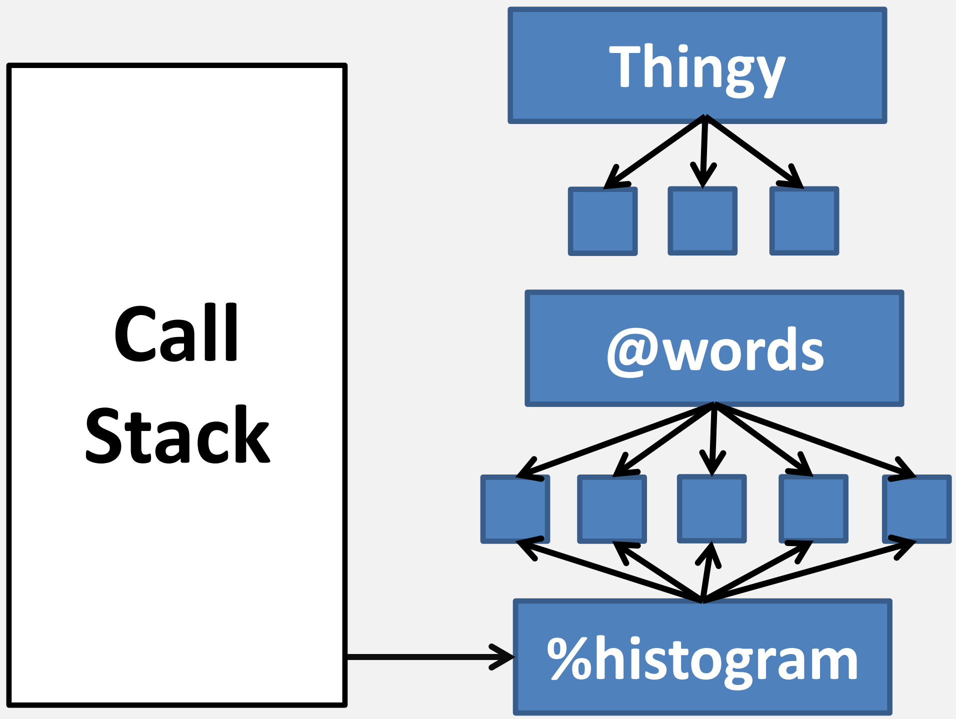
```
sub word_histogram($text) {
    my @words = $text.lc.comb(/\w+/);
    my %histogram;
    for @words -> $w { %histogram{$w}++ }
    return %histogram;
}

my %hist = word_histogram('Badger badger
        badger mushroom mushroom');
my @top_5;
# ...
```

Oh noes, out of memory!



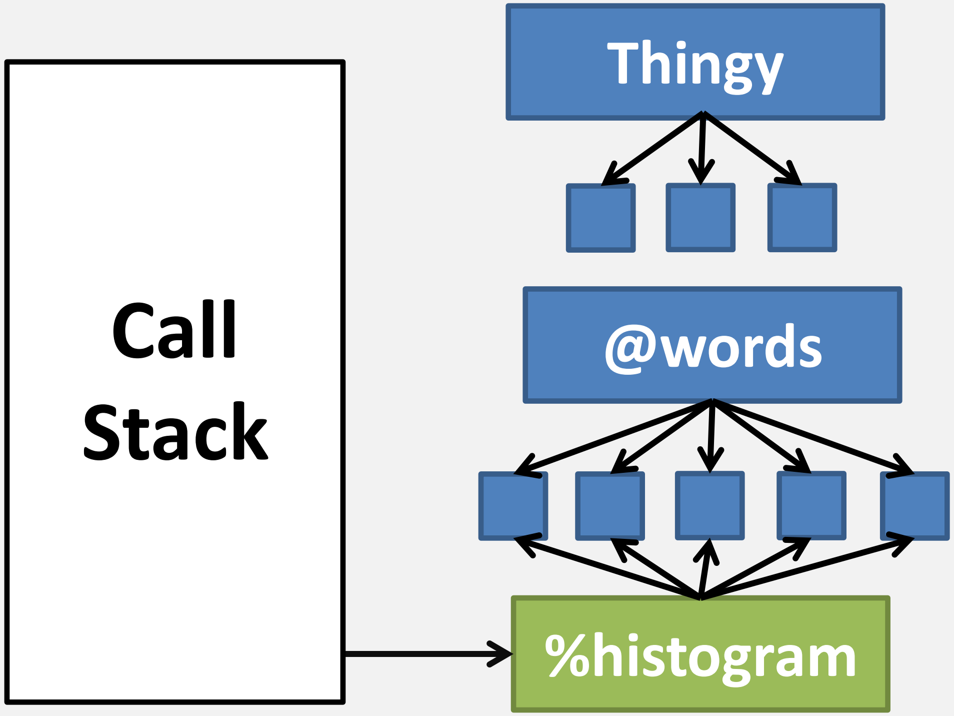
Reachability analysis

The vast majority of automatic memory management schemes are based on reachability



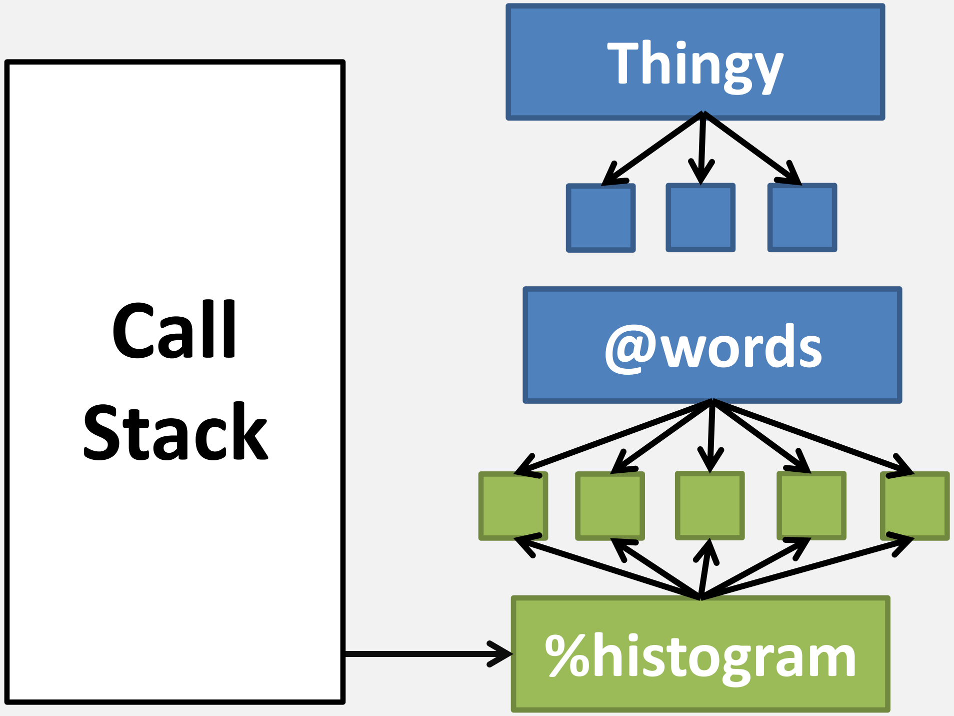
Reachability analysis

Reachability analysis starts out from a set of roots (things referenced from local variables, statics, etc.)



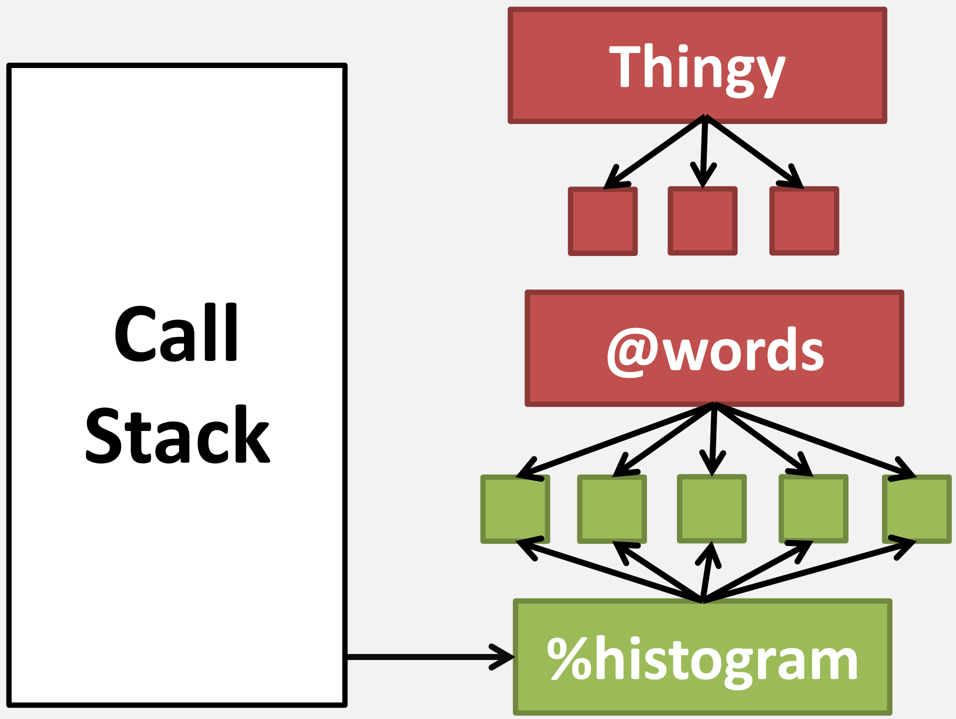
Reachability analysis

It then looks at what objects the roots reference, and then their references, and so forth…



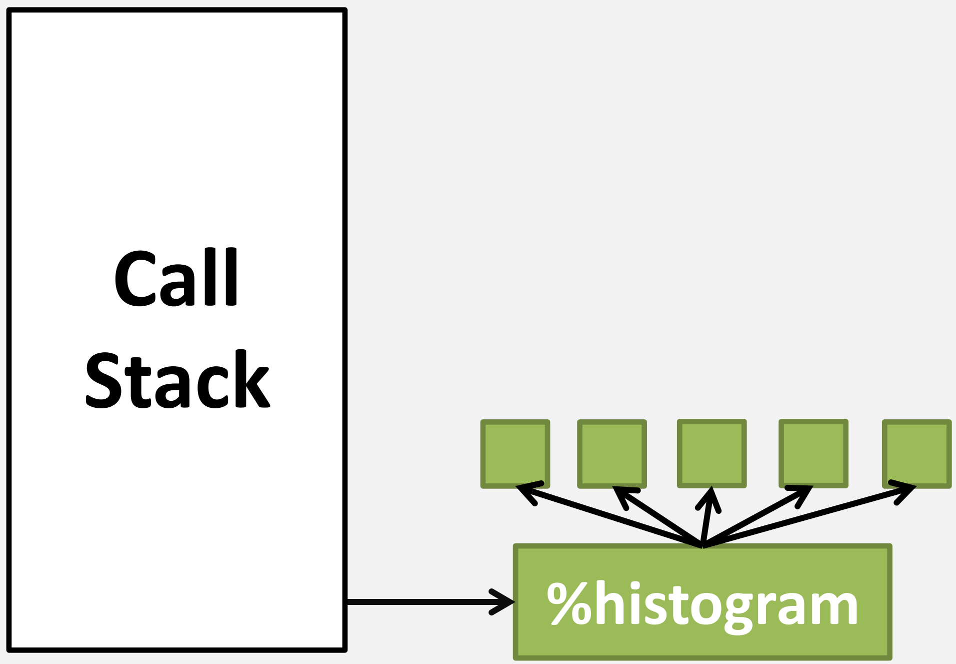
Reachability analysis

Anything that we never discover is unreachable, meaning the program can never use it again



Reachability analysis

The memory associated with these objects can therefore be released



And, that's basically it

So, now you understand what a GC does. Beer time!

And, that's basically it

So, now you understand what a GC does. Beer time!

Well, actually...

There's some not-so-basics too...

How do we find a piece of memory to allocate?

What are the set of roots we start the reachability analysis from, and how do we find them?

How do we find the references held in an object?

How do we keep track of where all the pieces of memory are, so we can redeem the memory we discover is no longer in use?

Mark and sweep

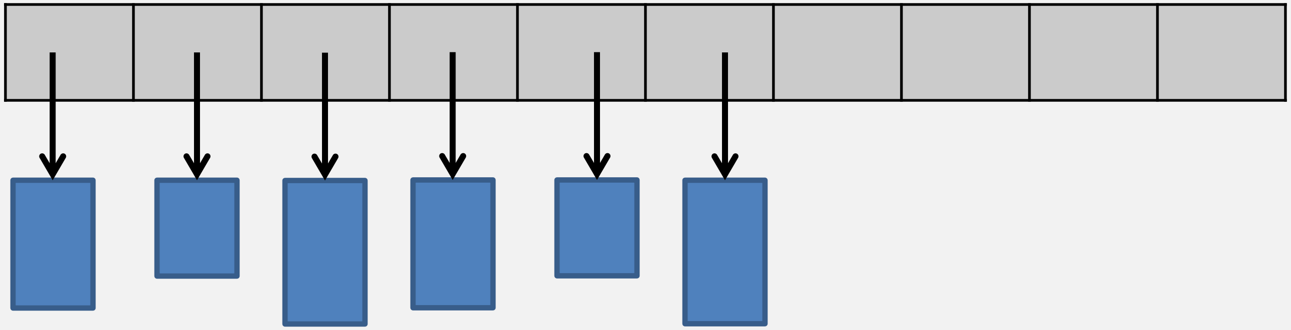
Let's start out simple

Mark and sweep

Let's start out simple

The simplest way to handle allocation is to not handle it at all, but instead delegate to `malloc`

Our allocator keeps an array of pointers to all the pieces of memory we obtained from `malloc`



Mark and sweep

Each object in memory should point to some kind of type table, saying what type of object it is and which of its fields are references to other objects

Furthermore, each object needs storage for a "mark bit", to be used in reachability analysis

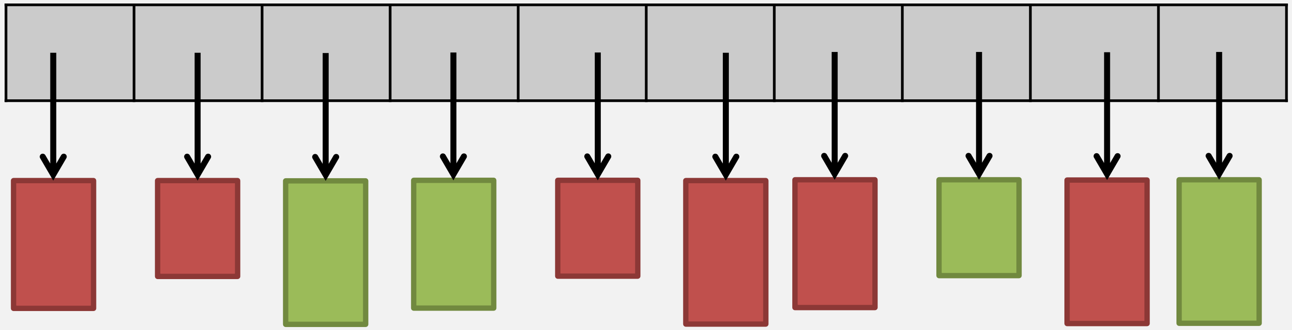
Type Table Pointer
Mark Bit
Field 1
Field 2

Mark and sweep

Marking is done by reachability analysis

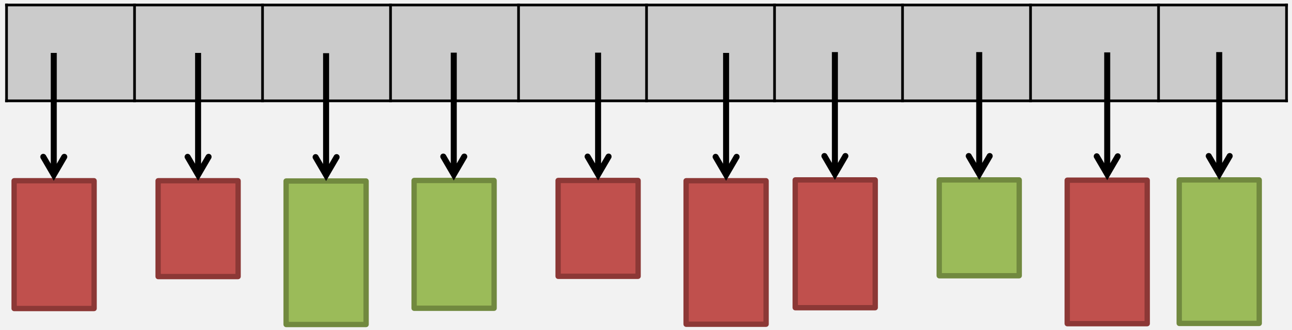
Whenever we reach an object, if its mark bit is not set, we set it, then also mark its references

Don't re-process already marked objects, otherwise we'd never terminate on cyclic data structures



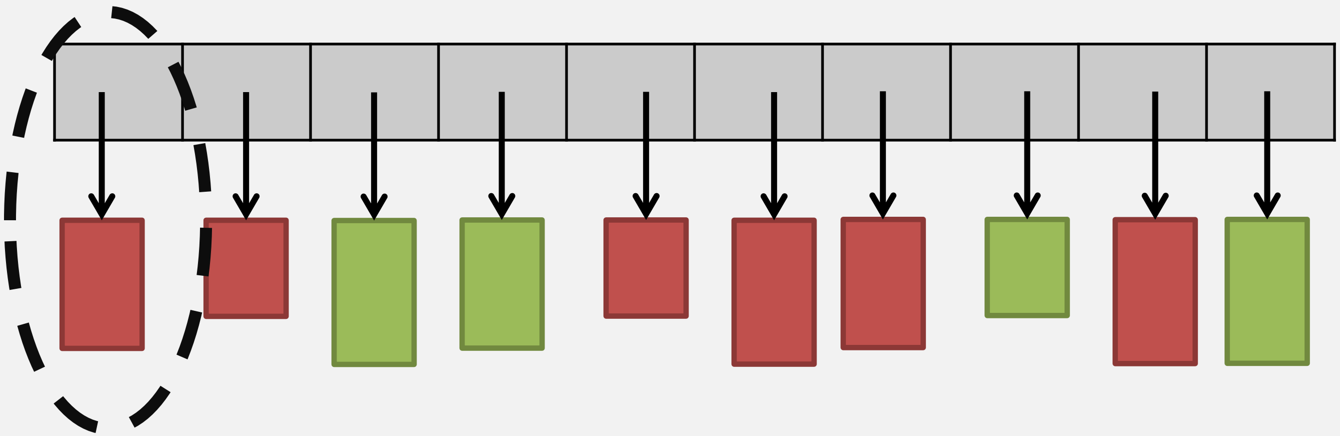
Mark and sweep

The sweep phase moves through the objects array, redeeming memory and clearing mark bits



Mark and sweep

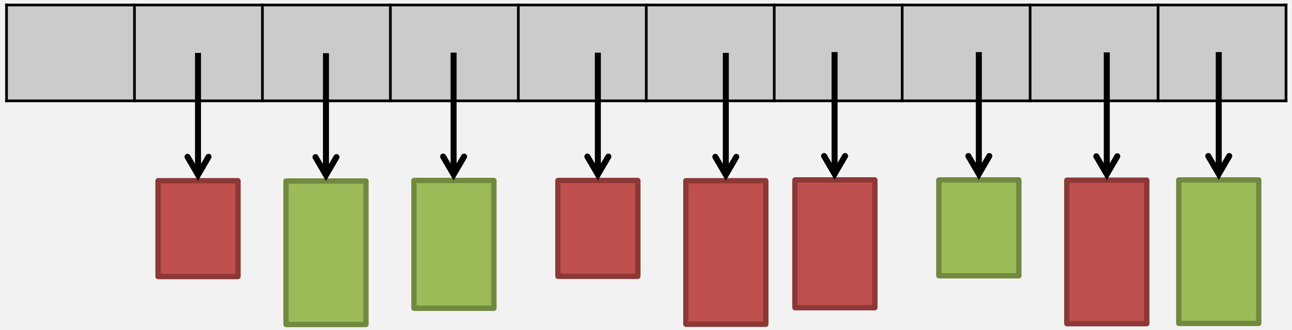
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit was not set, the memory is freed - in our very simple collector just by calling `free`

Mark and sweep

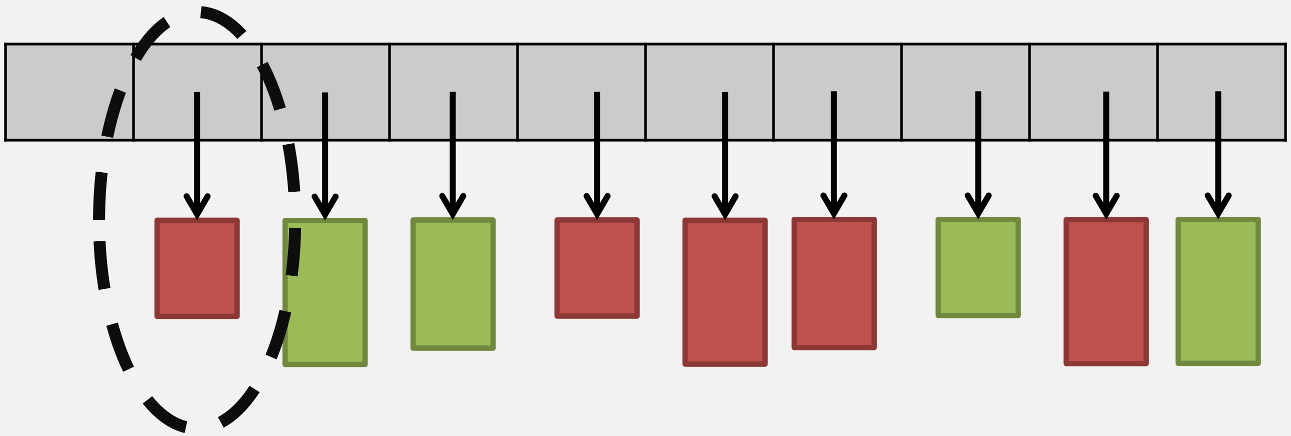
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit was not set, the memory is freed - in our very simple collector just by calling `free`

Mark and sweep

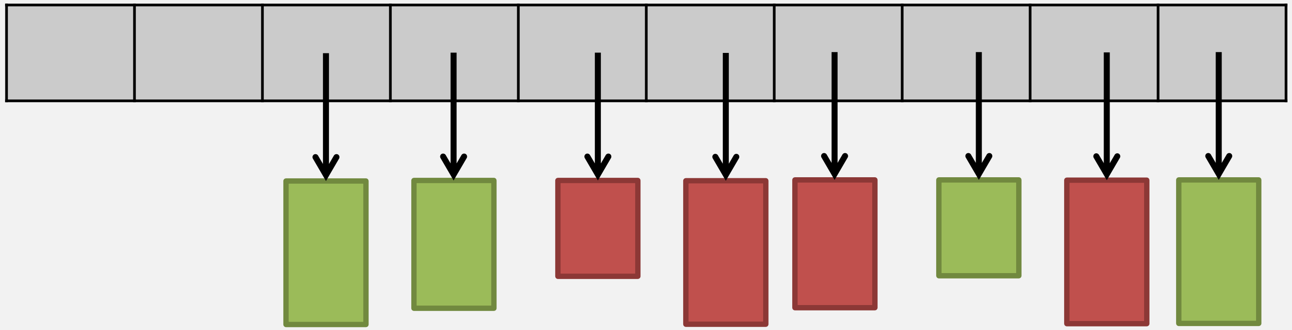
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit was not set, the memory is freed - in our very simple collector just by calling `free`

Mark and sweep

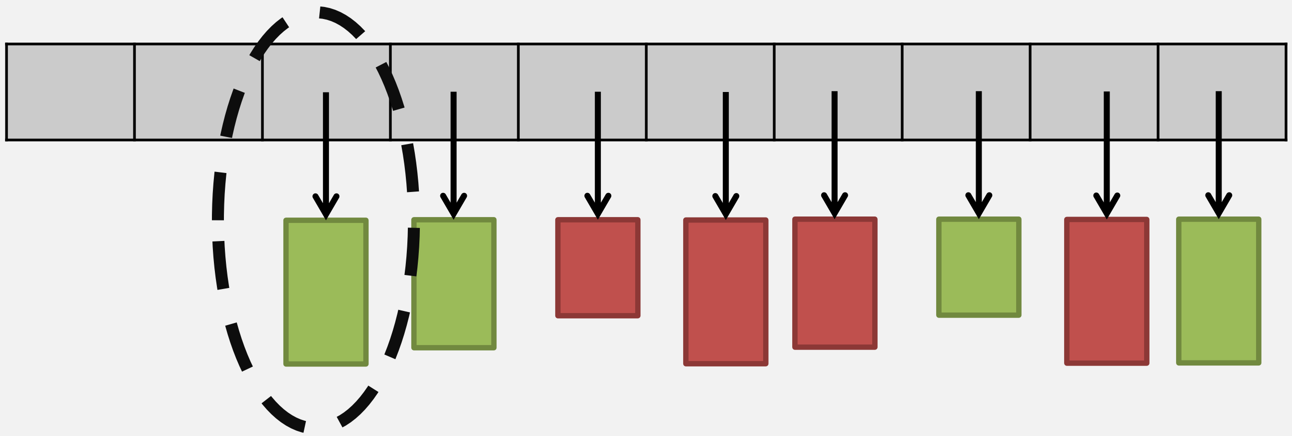
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit was not set, the memory is freed - in our very simple collector just by calling `free`

Mark and sweep

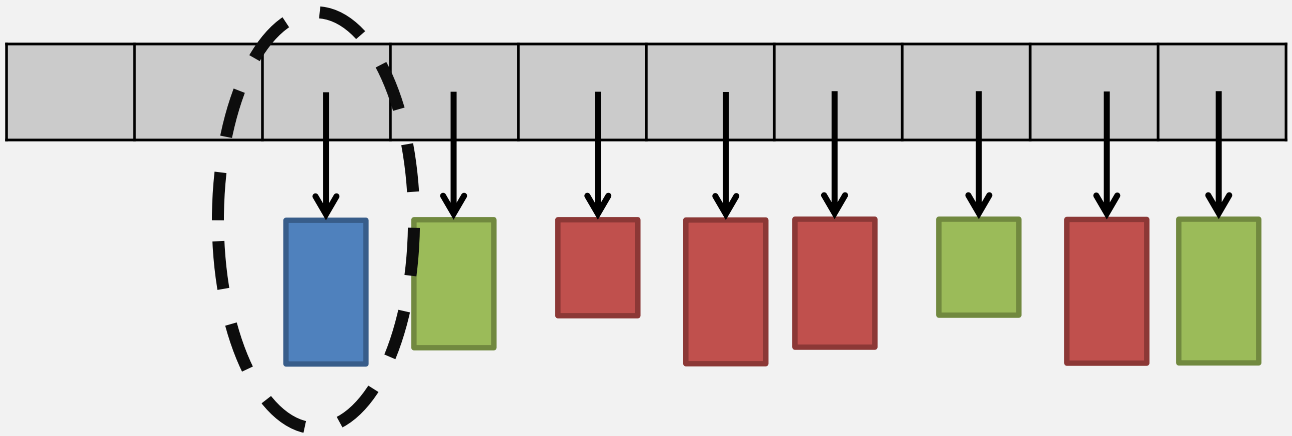
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit is set, we clear it, and then copy the pointer to the first free slot to the left, so future allocations will be easy

Mark and sweep

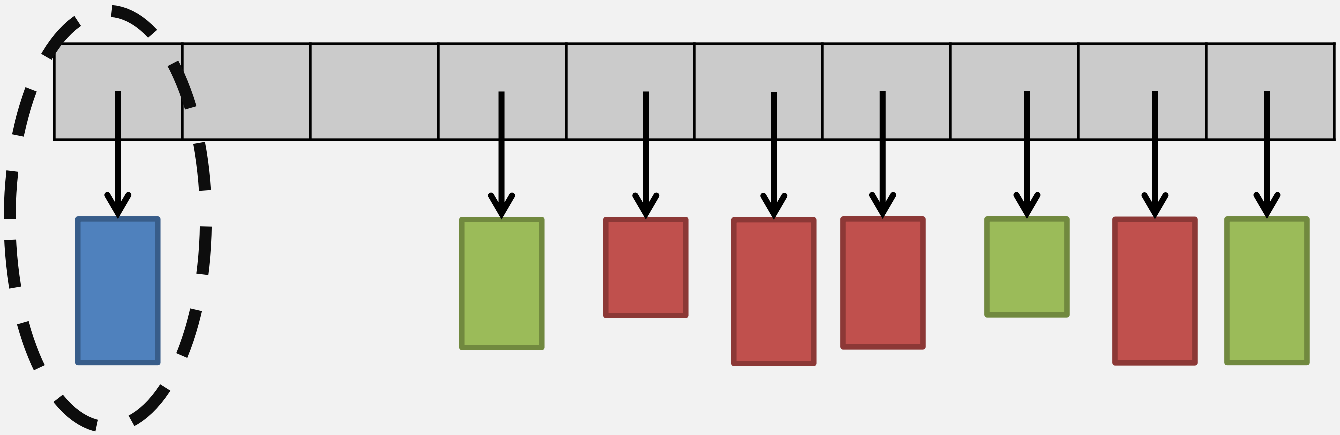
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit is set, we clear it, and then copy the pointer to the first free slot to the left, so future allocations will be easy

Mark and sweep

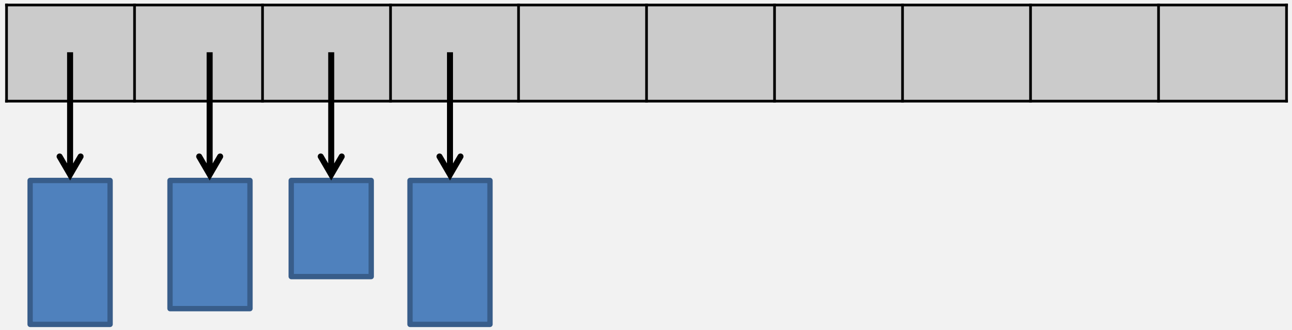
The sweep phase moves through the objects array, redeeming memory and clearing mark bits



If the mark bit is set, we clear it, and then copy the pointer to the first free slot to the left, so future allocations will be easy

Mark and sweep

The sweep phase moves through the objects array, redeeming memory and clearing mark bits



By the end, we've redeemed the memory of the unreachable, cleared all the mark bits, and can go back to running code, allocating memory, etc.

Mark and sweep

As GCs go, this is pretty easy to implement

Unfortunately, it's going to be rather slow as soon as we have any non-trivial number of objects, since...

`malloc` itself is rather slow

We have to consider every allocated object

We have to touch every object twice (bad for cache)

Finding the roots

Things in static variables are not so bad to track down, but local variables are another matter

These may live on the system stack if you are doing some kind of JIT compilation or a recursive interpreter

Even if they don't, and your runtime is allocating its own stack frames, then you may still have object references in your runtime implementation code - which, if you're in C, are on the system stack!

Conservative GC

The system stack is just an area of memory

You are allowed to access it at random

So, we can go hunting for object references on it, using our pointer array to check if things that look like pointers really are GC-managed pointers

We may get some false positives, but still safe

But walking the pointer list is O(n), each time... ☹

Precise GC

By contrast, a precise GC always knows where all of the pointers to objects are. No guessing!

If you JIT, you need to keep stack maps

For VM implementation code, need to track each of the local variables in scope when GC may happen

This is typically done by keeping a list of temporarily rooted things, which are considered by the GC

Temporary rooting

In an attempt at doing this in a structured way, I ended up defining a macro for this:

```
MVMROOT(tc, cu, {
    MVM_bytecode_unpack(tc, cu);
});
```

Which is defined as:

```
#define MVMROOT(tc, obj, block) do {\
    MVM_gc_root_temp_push(tc, (MVMCollectable **)&(obj)); \
    block \
    MVM_gc_root_temp_pop(tc); \
 } while (0)
```

Taking allocation into our own hands

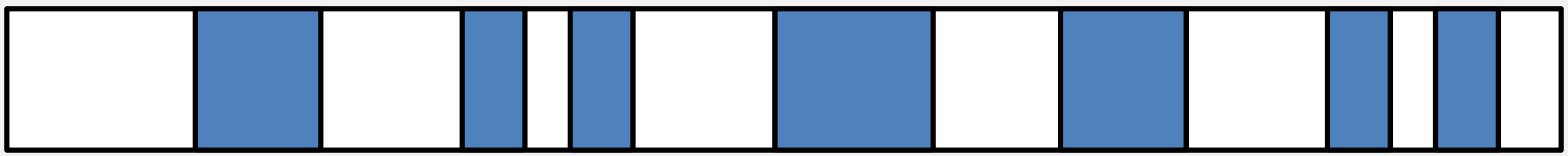
GC may be mostly about deallocation, but we can do a better job of that if we handle allocation ourselves

Just use malloc to get big blocks of memory, and allocate objects within those

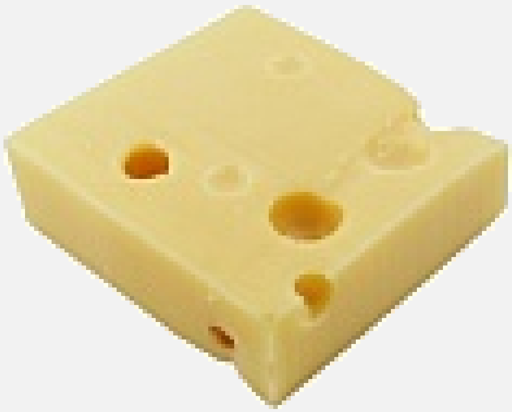
Heck, we can just "bump the pointer", allocating our way sequentially through the buffer! That'll be fast!

Ummm...not so simple!

After a GC run, we will have freed up some of the memory - but some will be in use



Our nice memory block now resembles a tasty morsel of Swiss cheese



So, what to do?

There are data structures that can help with finding memory blocks of the right size

Another popular scheme is sized pools: have a block of memory dedicated to objects that need 24 bytes, 32 bytes, 40 bytes, 48 bytes, etc. Then you just chain a free list through the pool.

Naturally, all of this is slower than the trivial bump-the-pointer allocation we'd like ☹

Aside: fun with caches

I once hunted a GC performance bug

It used conservative GC, and walked a linked list of fixed size blocks to see if a pointer was within them

In theory, fairly cheap

In reality, fixed sized blocks were page aligned, and some CPUs just use the least significant bits as the key into their L1 cache ➔ awful cache thrashing; got a 20% win from keeping a compact lookup table

Compacting collection

A useful insight:

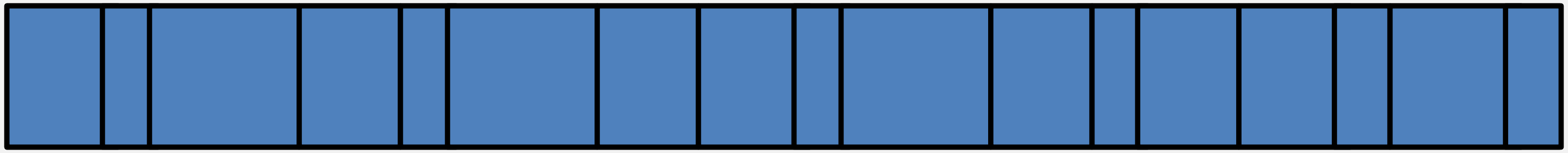
If we know where all the pointers to an object are, (which precise collection gives us), then we can move the object during a GC run!

We just need to be sure to update all the pointers (this is why precise GC matters)

Opens the door to numerous alternative algorithms involving compaction or copying

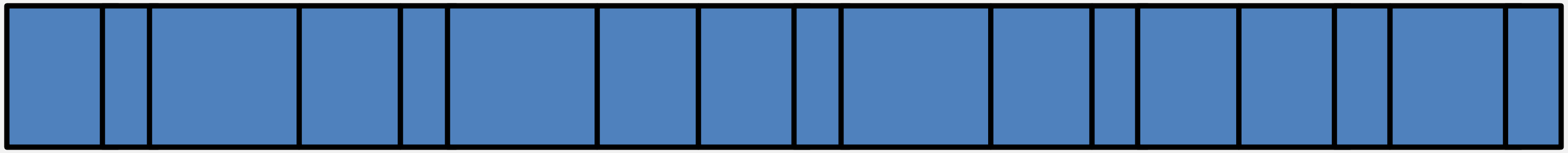
Compacting collection

Do bump-the-pointer allocation, until the memory block is filled

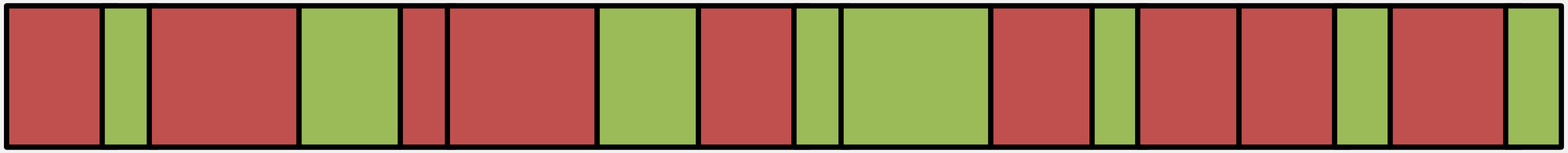


Compacting collection

Do bump-the-pointer allocation, until the memory block is filled

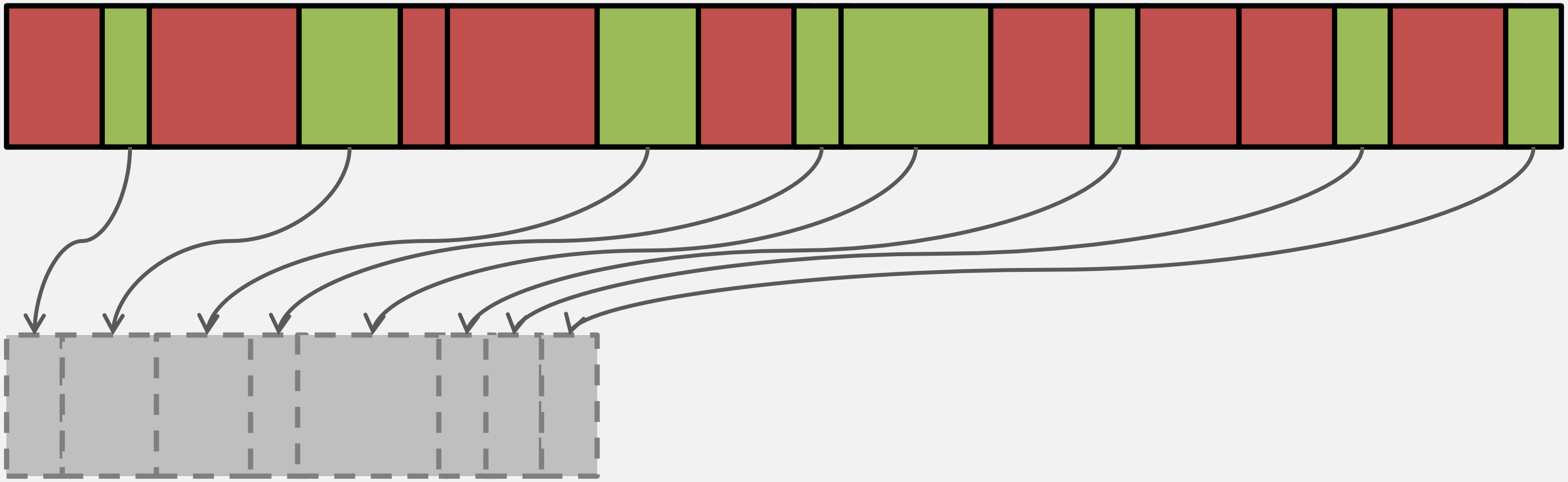


Then, do the usual reachability and marking, as seen in the mark-and-sweep collection



Compacting collection

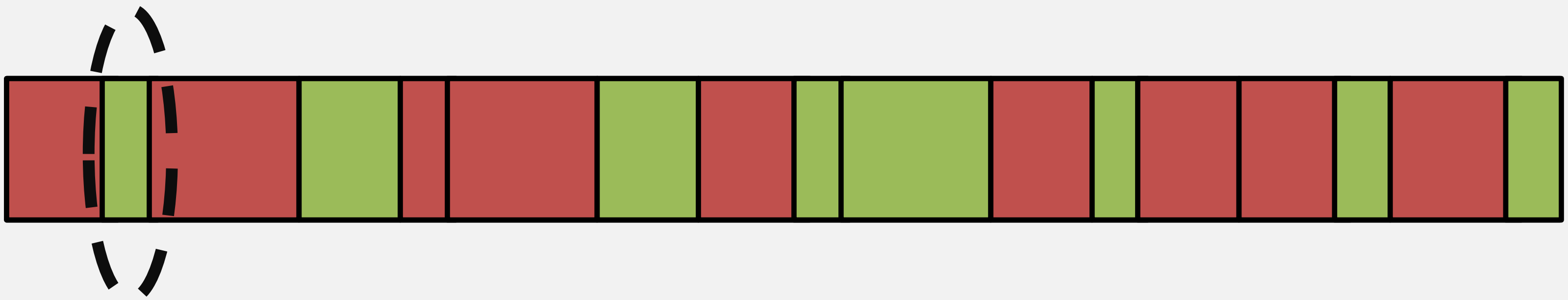
Next, we need to compute a new address for each of the living objects, such that they will end up all at the start of the block



This address mapping needs to be stored, perhaps in some kind of hash table

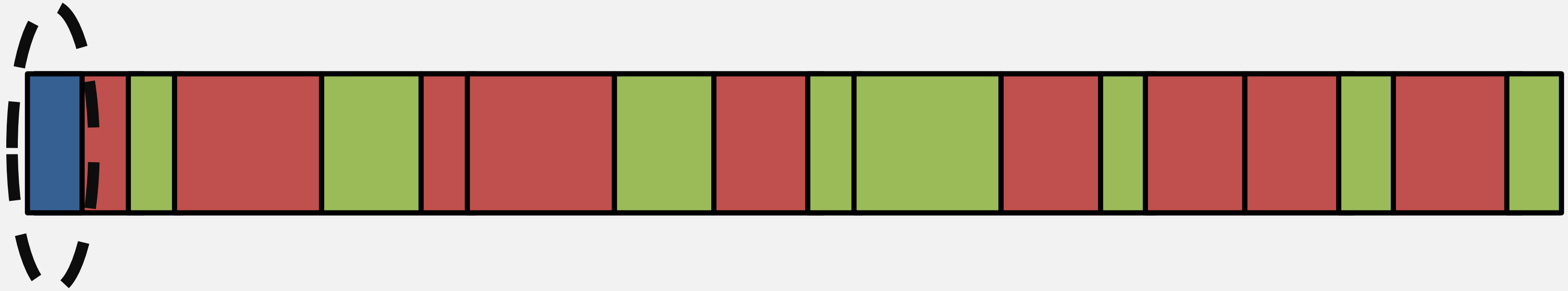
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



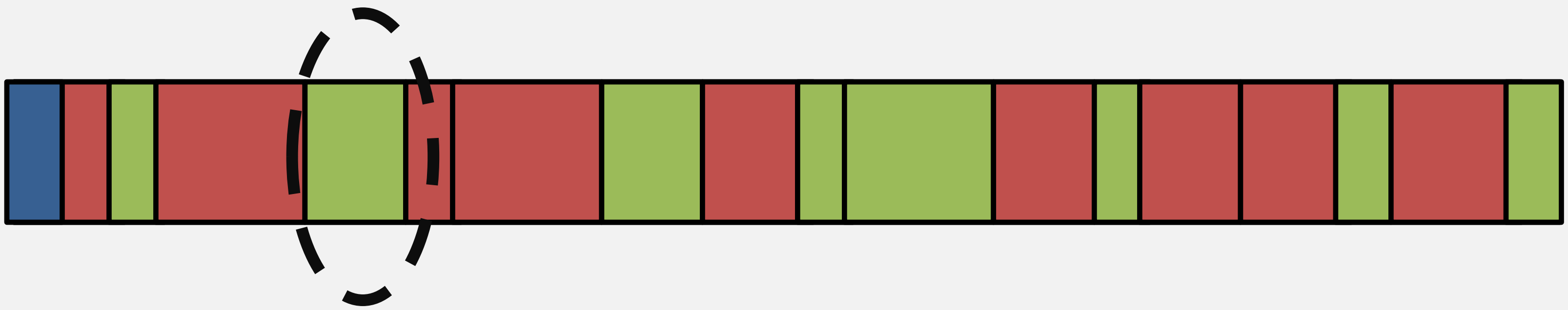
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



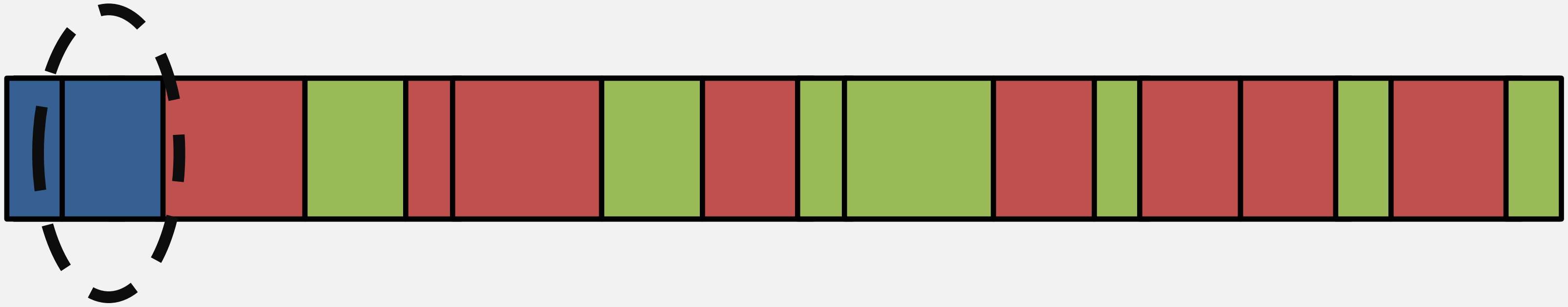
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



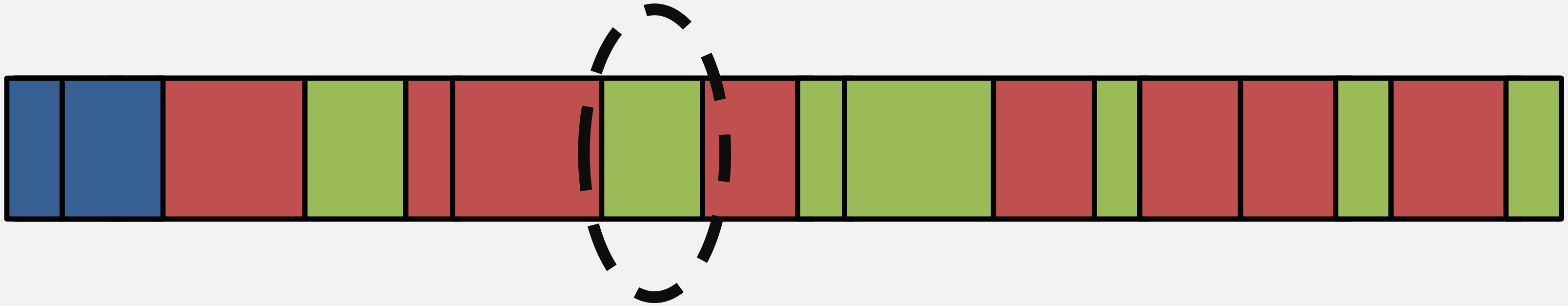
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



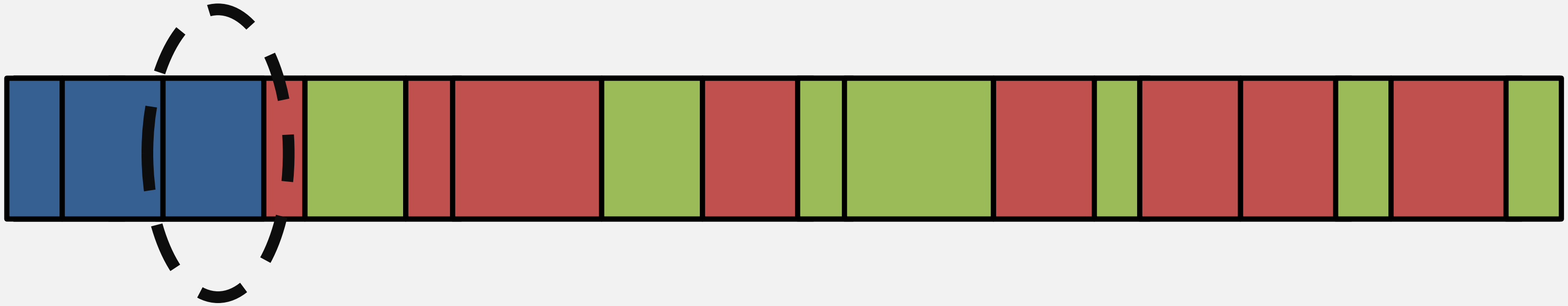
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



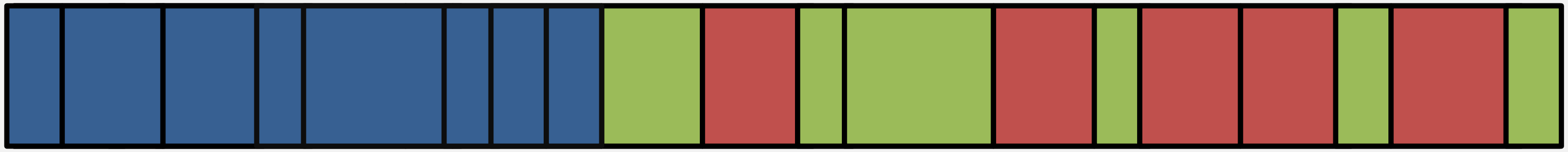
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



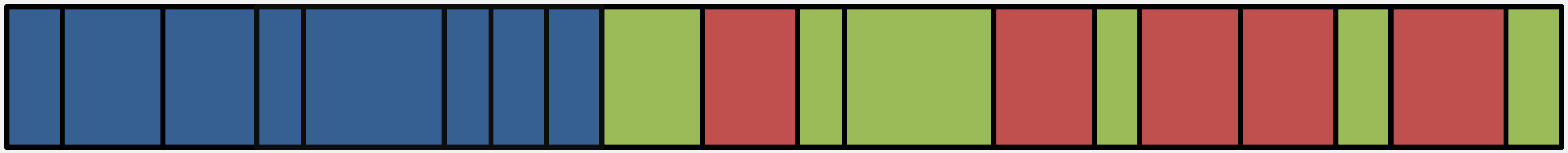
Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



Compacting collection

We then go through the living objects. For each one we copy it to its new address, clear the mark bit, and update any references within it



Finally, we zero the rest of the area

Compacting collection: improvements

This algorithm ended up making three passes, though there are tricks to help with that

Computing new addresses in reachability analysis is tempting, but then you can get them in any order and compaction becomes much harder

Can build pointers-to-update list as we mark

Then do new address computation, copying and pointer updates in a single pass

Compacting collection: pros

Cheap bump-the-pointer allocation

Objects are bunched together post-collect (good for cache hit rate on them)

In theory, careful algorithm choice means we can re-arrange objects for cache locality by understanding how they reference each other

In practice, fancy approaches on this don't seem yield more benefit than the analysis they need

Compacting collection: cons

We must be precise (know all the pointers)

If we pass an object to native code, then we must pin it (meaning we promise not to move it). This complicates new address computation

Interior pointers are tricky to support

We must make at least two passes over an object: one to mark it and look at its references, and another to move it; this is not so cache friendly

Semi-space copying

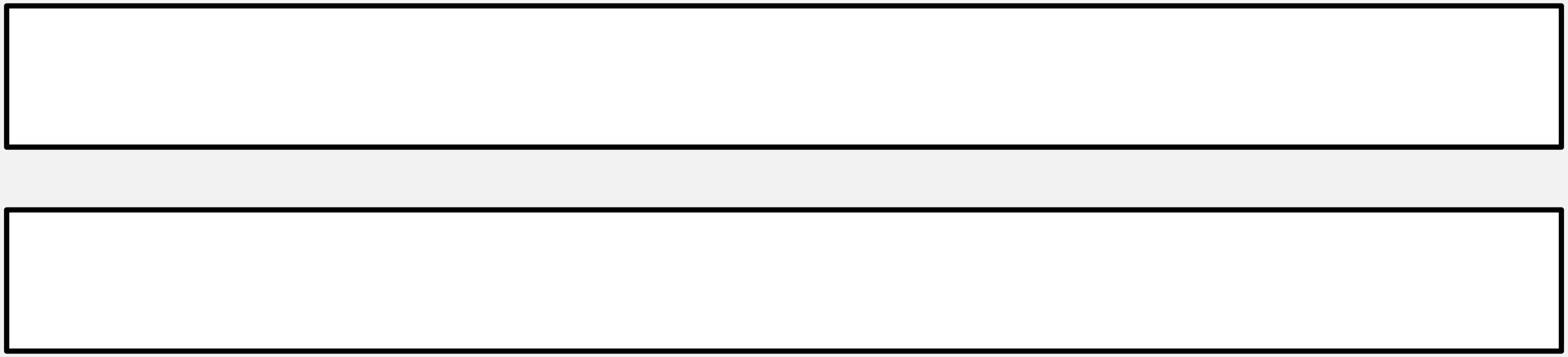
What if we could do bump-the-pointer allocation and just make one pass over the objects?

Semi-space copying

What if we could do bump-the-pointer allocation and just make one pass over the objects?

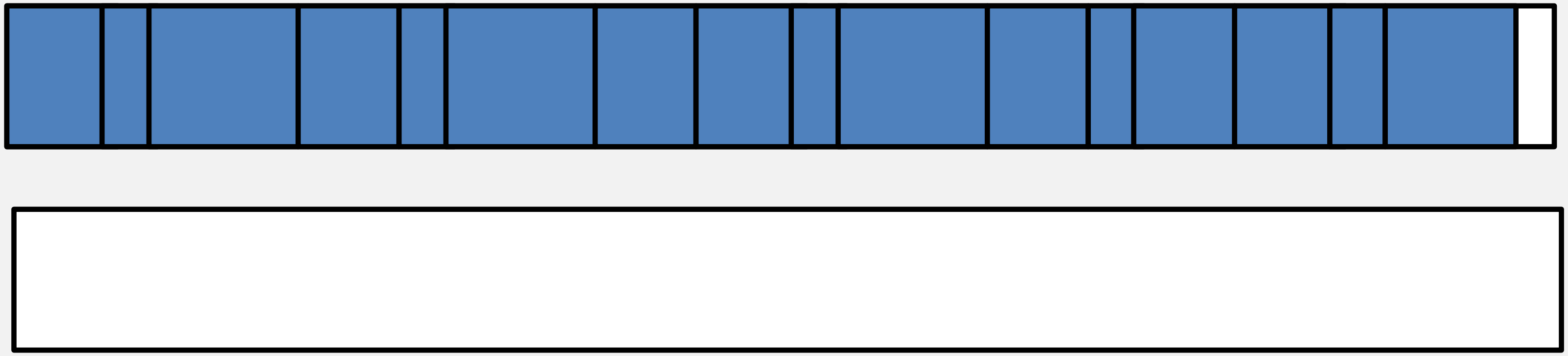
It turns out we can - at a cost

A semi-space collector uses two equally sized regions of memory



Semi-space copying

We use one of the regions to allocate new objects in, and keep allocating until it is full



For this memory block, we can use the nice, cheap, bump-the-pointer allocation

☺

Semi-space copying

The basic idea of the algorithm is to copy each of the reachable objects into the other memory space

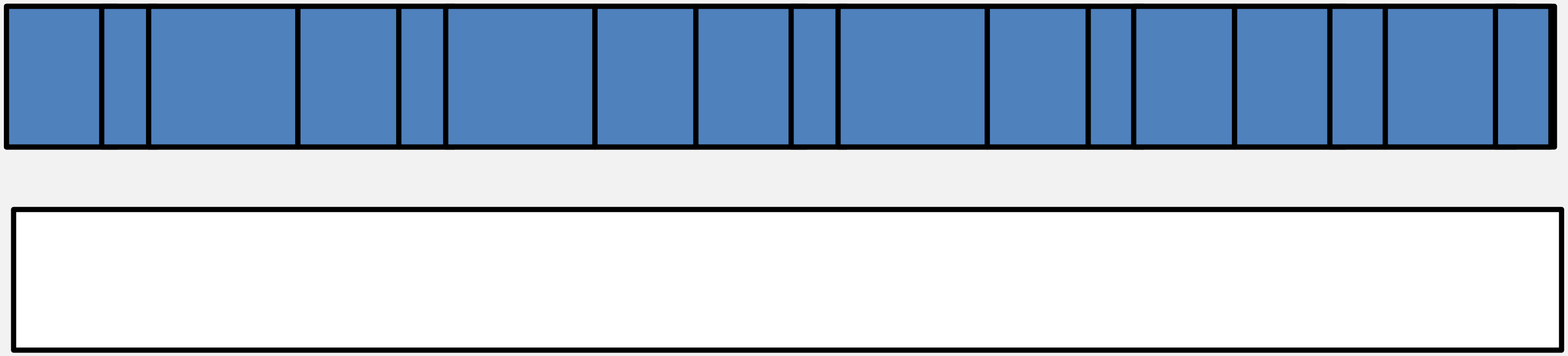
This is a one-pass process. However, we need to store the new address for each object; the easy way is a forwarding pointer in the header

Type Table Pointer
Forwarding pointer
Field 1
Field 2

Semi-space copying

Do reachability analysis, but instead of just marking:

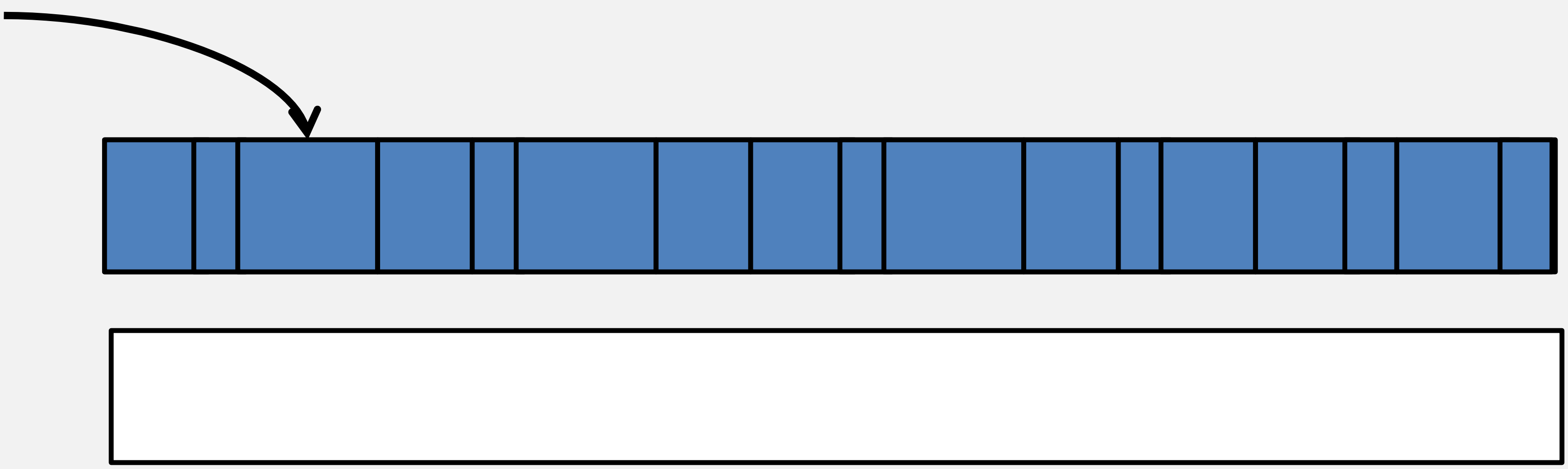
Calculate a new address in the second space
Copy the object to the new address
Write the address into the header



Semi-space copying

Do reachability analysis, but instead of just marking:

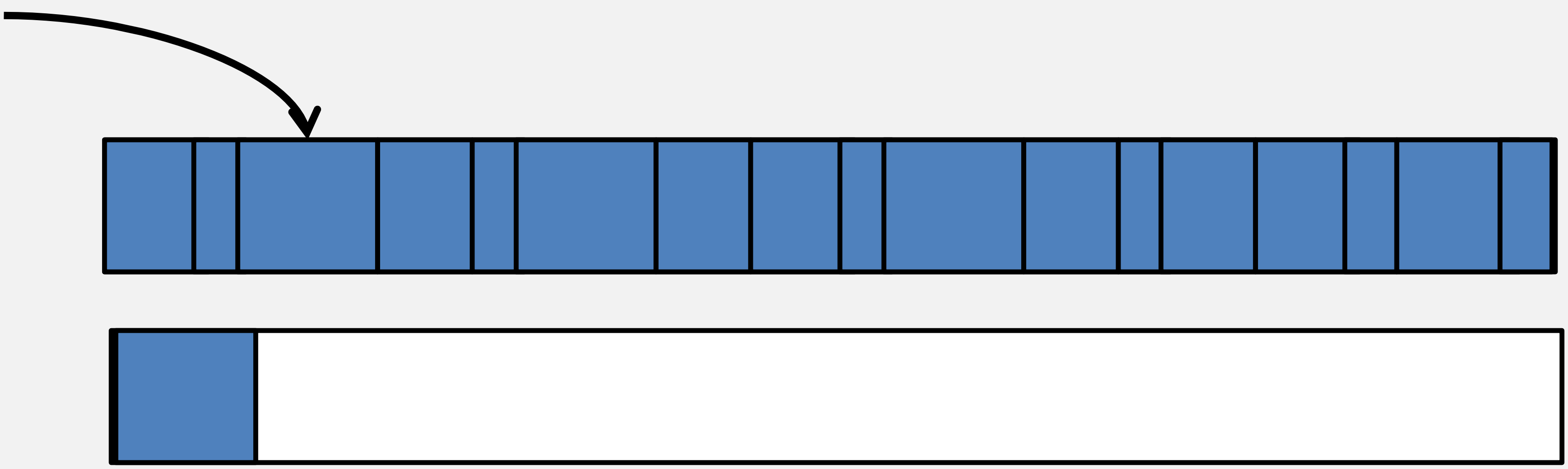
Calculate a new address in the second space
Copy the object to the new address
Write the address into the header



Semi-space copying

Do reachability analysis, but instead of just marking:

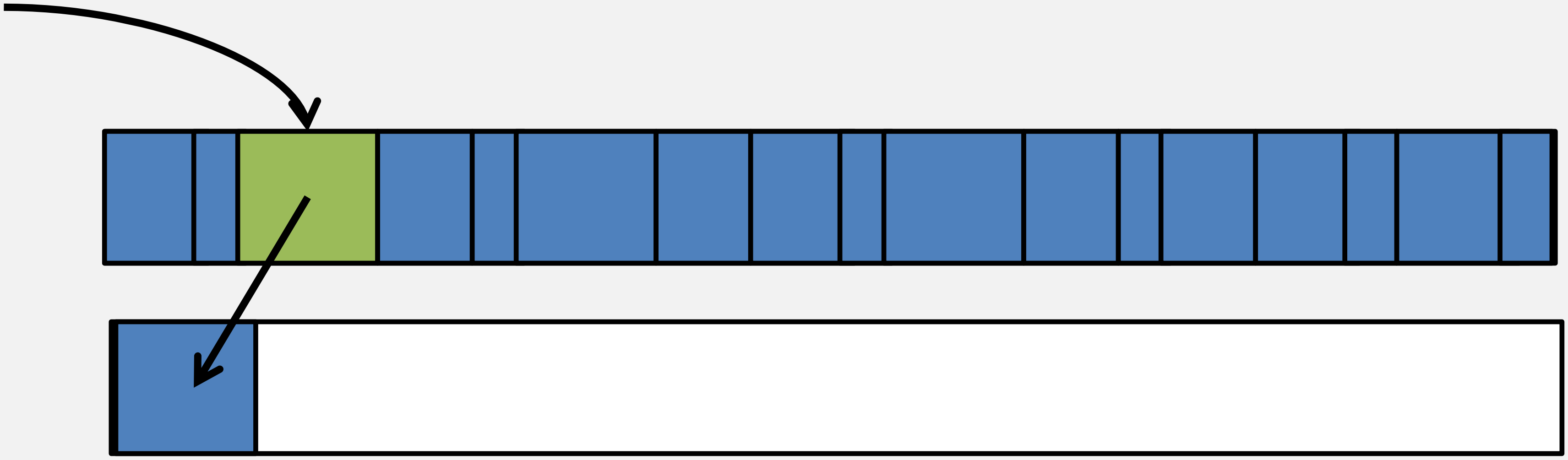
Calculate a new address in the second space
Copy the object to the new address
Write the address into the header



Semi-space copying

Do reachability analysis, but instead of just marking:

Calculate a new address in the second space
Copy the object to the new address
Write the address into the header



Semi-space copying

We update pointers as we go

When we first copy an object, we update the pointer we saw to it immediately with the address that we copied it to

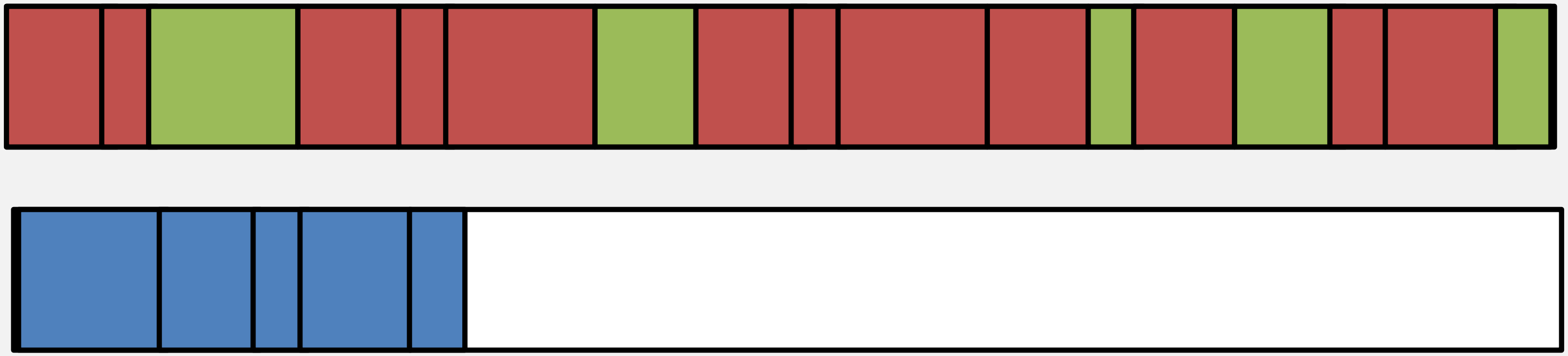
If we see a pointer to an object that has a forwarder, then we already copied it; just update the pointer

If we see a pointer into the new memory region - it's already updated, so ignore it

Semi-space copying

Once we're done, all the reachable objects have been copied into the second semi-space

We now continue allocating objects in there, using bump-the pointer, until it is full. Then the roles flip.



Semi-space copying: pros and cons

Really quite easy to implement

Get cheap, bump-the-pointer, allocation

Very cache friendly, as we only visit each object once, and we recently touched all the memory we copied living objects into, so it should be hot

However, we have to double the memory space - after usual overhead! Surely this can't be practical?

Visit ALL the heap?!

Generational collection

Most objects don't last long. They are allocated, used for a short amount of time, and then become unreferenced. They don't survive a single GC run.

Most objects that survive 1-2 GC runs will likely also survive quite a few more runs.

This is the generational hypothesis. Most objects are short lived or long lived. Additionally, long lived objects are often mutated less, whereas short lived ones are in active use and so are mutated lots.

Generational collection

A generational collector breaks objects up into at least two generations (2-3 is the norm)

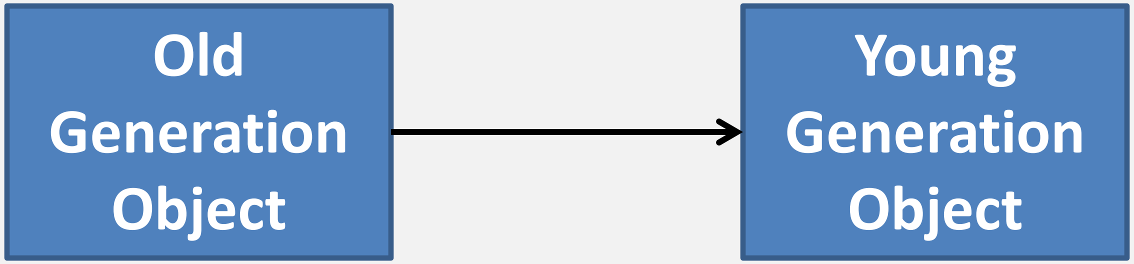
Objects are allocated in the young generation, sometimes known as the nursery

If they survive a certain number of collections, they are promoted to the old generation

The trick is that we only consider the young generation in most garbage collection runs

Generational collection

The thing that makes this difficult is when the only remaining reference to a young generation object is from an old generation object



If we're ignoring old (gen-2) objects, we'll miss it!

```
item_gen2 = item->flags & MVM_CF_SECOND_GEN;
if (item_gen2 && collecting == MVMGCGenerations_Nursery)
    continue;
```

Generational collection

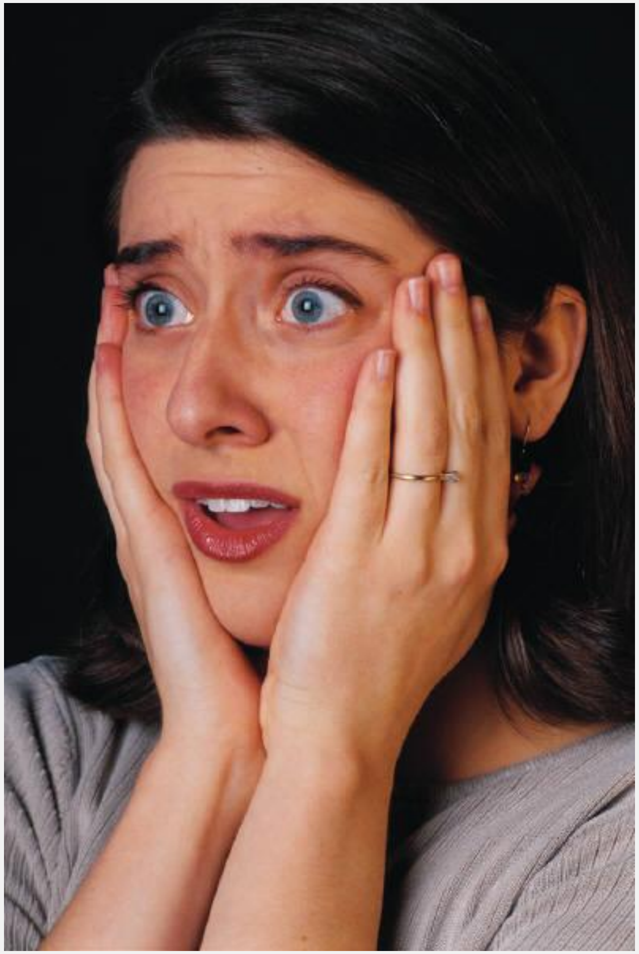
To cope with this, we use a write barrier

Every time we write a pointer to a new object into an old object, then we put the old object into a remembered set, and treat it as a root

```
#define MVM_WB(tc, update_root, referenced) \
    { \
        MVMCollectable *u = (MVMCollectable *)update_root; \
        MVMCollectable *r = (MVMCollectable *)referenced; \
        if (((u->flags & MVM_CF_SECOND_GEN) && r
                && !(r->flags & MVM_CF_SECOND_GEN))) \
            MVM_gc_write_barrier_hit(tc, u); \
    }
```

Generational collection

Isn't the write barrier terribly costly?!



Generational collection

Isn't the write barrier terribly costly?!

No, not really

It uses pointers we'd already have in the CPU register and memory we'd have in cache anyway

Fits well with superscalar CPU architecture

Comes out vastly cheaper than having to consider the entire heap every collection!

Concurrent collection

One problem with all of this is that running the GC involves a reasonable amount of work

If you are building a graphical application or something that needs to feel very responsive to a user, the pauses can become as a UX issue

Therefore, a range of concurrent GC algorithms exist, which run the GC at the same time the program is running, typically on another thread

Concurrent collection: terrifying

We'll not cover concurrent GC algorithms in this session, partly due to lack of time, and partly for our collective sanity

In short, they are *difficult* to implement

Read barriers may be involved. That is, every time you read a memory address, you may need to check that the object didn't move underneath you!

Interesting, but a whole other talk

The pause/throughput trade-off

While a concurrent GC can reduce or practically eliminate pause time, the extra bookkeeping required to implement it comes at a cost

The .Net CLR actually comes with two collectors: a client one and a server one

The client one is a concurrent collector. The server one is not. Why? Because on a server you typically care about overall throughput, not keeping up a certain frame rate

Parallel collection

Actually, much easier

Still stop all threads to do the GC run

Just parallelize the work

Many GC algorithms parallelize quite reasonably

Good enough for now, though once we need to deal with 16+ cores the synchronization overhead may be a killer ➔ may force us to concurrent anyway

So what did I choose?

So what did I choose?

MoarVM has a generational collector

The young objects are managed by a semi-space copying collector, for fast allocation/cleanup

The old objects live in sized pools, and a free list is chained through it

Once in old generation, objects never move

Pinning = allocate right away in the old generation

Takeaways

GCs do all kinds of things behind the scenes

You'll probably not need to implement one, but performance programming in a language with a GC means understanding roughly what it's doing

Also, the JVM offers a choice of collectors, and knowing how each of them basically works may help with choosing an appropriate one

In reality, benchmarking will help you much more

Things to remember

Allocations make work. Reducing allocations helps. C# programmers, learn about when to use `struct`!

Allocating lots of **large objects** may also have a negative impact. Repeated string concatenation or regular collection resizing can be pain points.

Since VMs tend to assume the **generational hypothesis**, it's now something of a performance rule. Avoid mid-life crisis; have short-lived and long-lived objects, but not medium-lived.

Thank you!

Hunt me down...

Email: jonathan@edument.se
Twitter: @jnthnwrthngtn

Questions?

P.S. Think I'd be fun to work with? Edument is hiring. Not for writing GCs...but if you like teaching/mentoring and building quality stuff, come and say hi. kthx. ☺