# Speculation

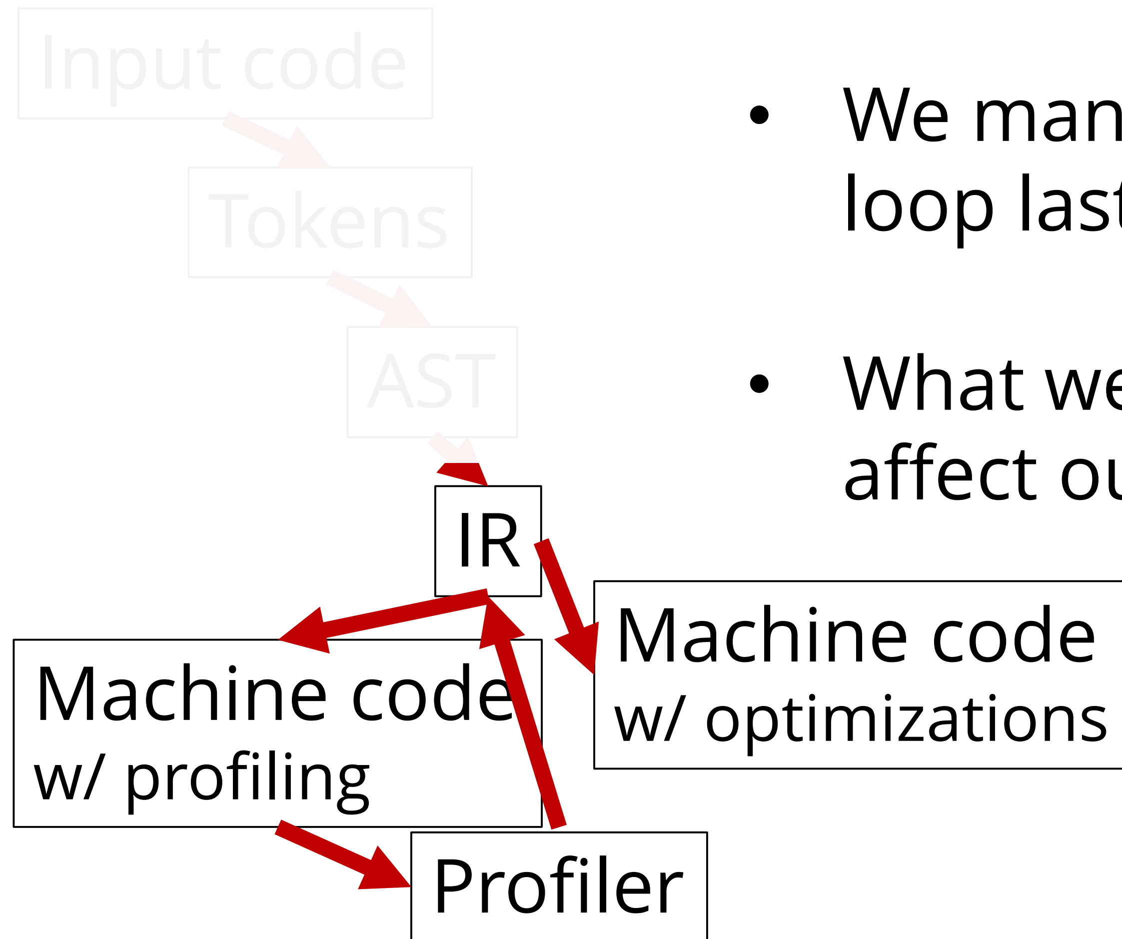
Or at least, so I speculate

# Recall inline caching

- Inline caching:
  - Separate object *shape* from *content*
  - *Cache* object shape
  - ~~*Optimize* assuming~~ same shape seen again

**Speculate that**

# Cache more!




- We managed to avoid completing this loop last time by cheating
- What we want to profile: Types that will affect our compilation

```
function badmul(a, b) {
    var ret = a;
    for (var i = 1; i < b; i++)
        ret += a;
}
```

This operator *heavily* depends on the types of `ret` and `a`

We can generate totally generic code, or *specialized* code for different types

```
function insanemul(a, b) {
    var ret = a;
    for (var i = 1; i < b; i++)
        [if type of ret and a are int]
            ret += a; [int-specialized]
        [if type of ret and a are string]
            ret += a; [string-specialized]
        [if type of ret and a are array]
            ret += a; [array-specialized]
        [...]
}
```

(Note: Our generic adder has to do this anyway, so doing this in the JIT isn't even an optimization... yet)

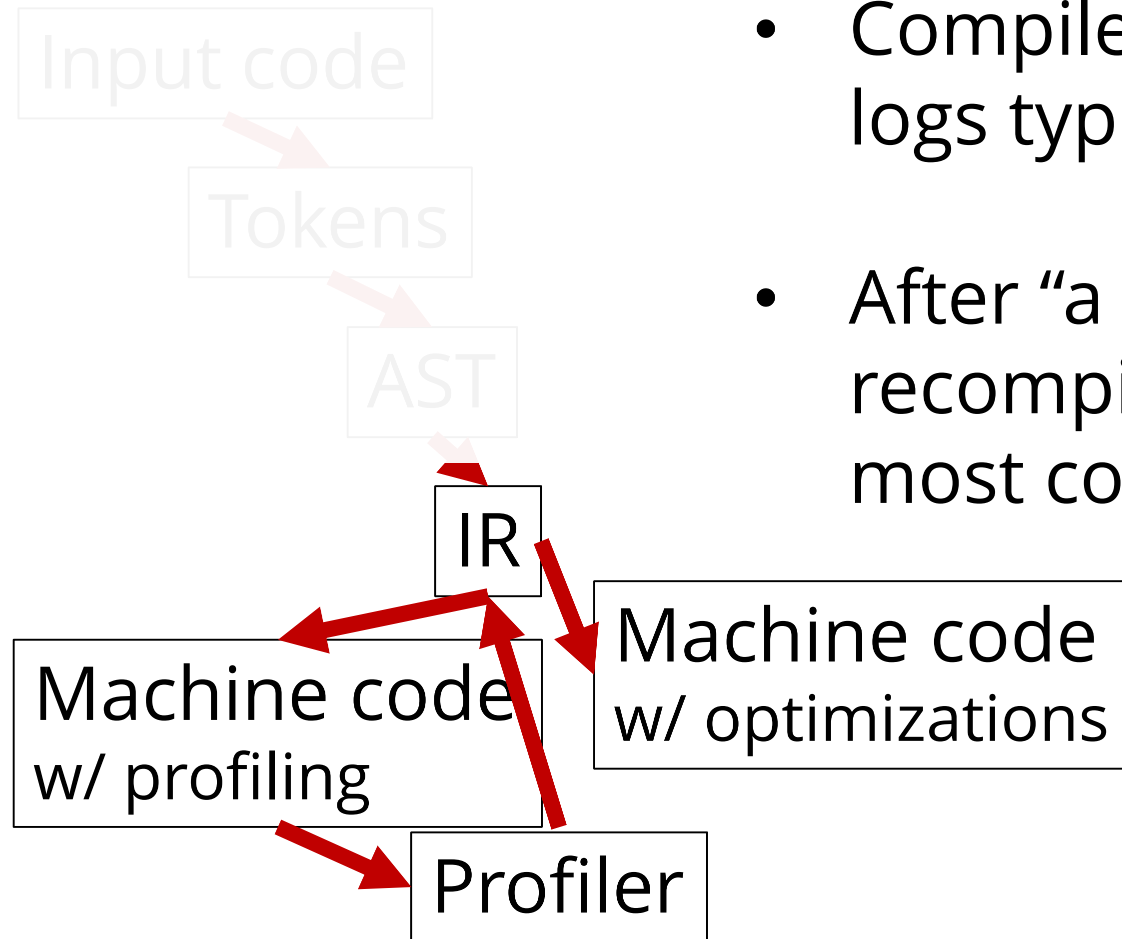
# When to check what

- To make this efficient:
  - Check at the right time, reuse its result
  - Know what to check for
    - Don’t check the whole powerset!
  - Check in the right order

# [and why]

- Check too early: Is this value accessible elsewhere? Could the type change?
- Check too late: Checking repeatedly, redundantly
- Check types that never occur: Wasted cycles
- Check less-common types first: More wasted cycles

# Profiling!




- Compile with un-optimized code that logs types (or other profiling info)
- After “a few” runs of any function, recompile with code specialized for most common types

(The number that qualifies as “a few” is a “brown number”: We pull it out of our ass and hope it’s good enough.)

# Multi-stage JIT

- Most modern JITs are *multi-stage JITs*:
  - Stage 1: Interpreter or template JIT with profiling
  - Stage 2: Type-specialized code with inline caches
  - Stage 3: Advanced optimizations (mostly the same as any ahead-of-time compiler!)

# Multi-stage JIT advantages

- Start-up time: Interpreters and template JITs aren’t fast, but they start fast
- Stage ≥ 2 JIT can run in another thread
- Part of profiling is how often the function runs: Only optimize hot functions!

# Multi-stage JIT disadvantages

- Ossification: It’s hard to go back
- Unpredictability: Performance changes at unknowable times
- Hand-off can be complicated

# e.g. V8

- V8 has changed many times, but right now:
  - Starts with an interpreter+profiler
  - Moves to TurboFan+light profiling
  - Moves to TurboFan+more optimizations

# SDyn

# Escape hatches

- Specialized code needs an *escape hatch*
- Escape hatches can be *complicated!*
  - Jump to unoptimized version?
  - Jump to interpreter (and copy out values)?
  - Re-compile code?

# A typical escape hatch

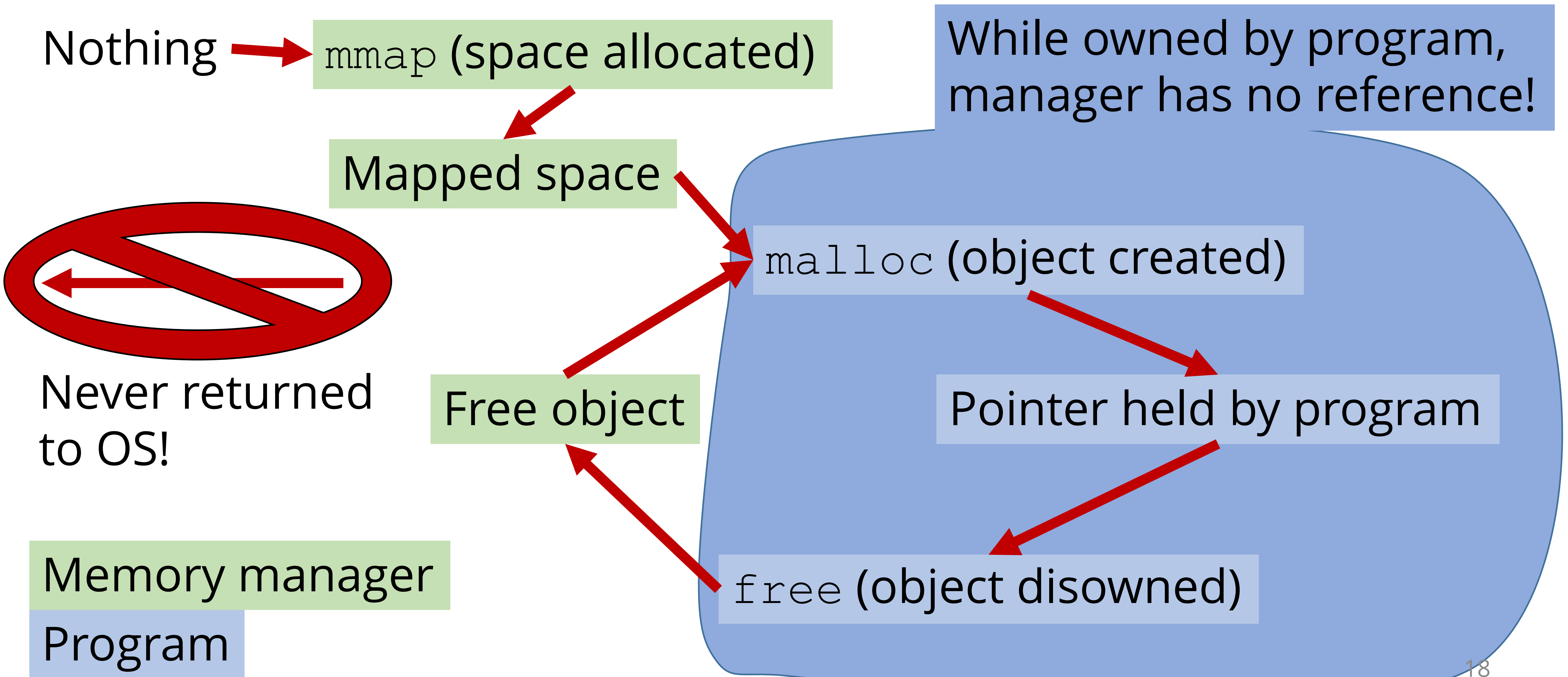
- Escape-to-interpreter:
  - During compilation, create extra code to extract variables from heap
  - Every escape hatch must remember its IR location
  - Escape by storing IR location, jumping to variable-extraction code, *popping stack*, jumping to interpreter
  - We never return to the JIT function!

# Ossification

- [demo]
- A constant battle: Profile or no?
  - Profile in optimized code: Less optimal! (collecting profiles takes time!)
  - Don’t profile in optimized code: We only see escapes, don’t know relative frequency

# GC basics

# Review



# Automatic memory management

- Defining principle: `free()` is automatic
- Common solution: Garbage collector
  - Part of the runtime does `free()` for you
- Other solutions exist (e.g. type-based)
- We focus on GC

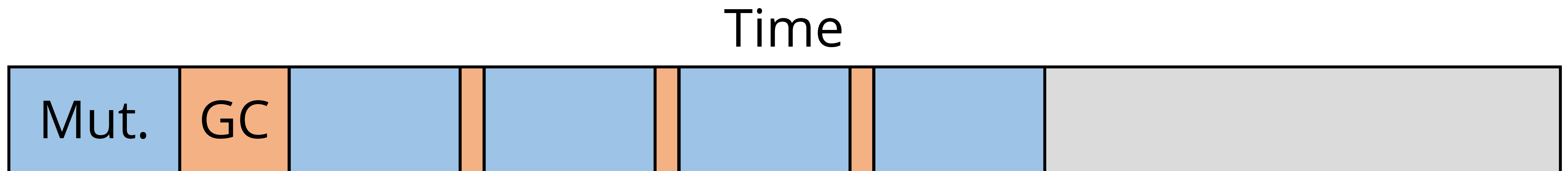
# GC glossary

- Collector
- Mutator
- Heap vs. C heap
- Pool
- Root
- Reference
- Reachable
- Type information
- Stop-the-world
- Pause
- Parallel
- Concurrent

# Performance

Many ways of measuring performance:

- Throughput
- Responsiveness
- Latency
- Resource utilization
- Fairness



# Performance consideration

Manual memory management ain’t free!

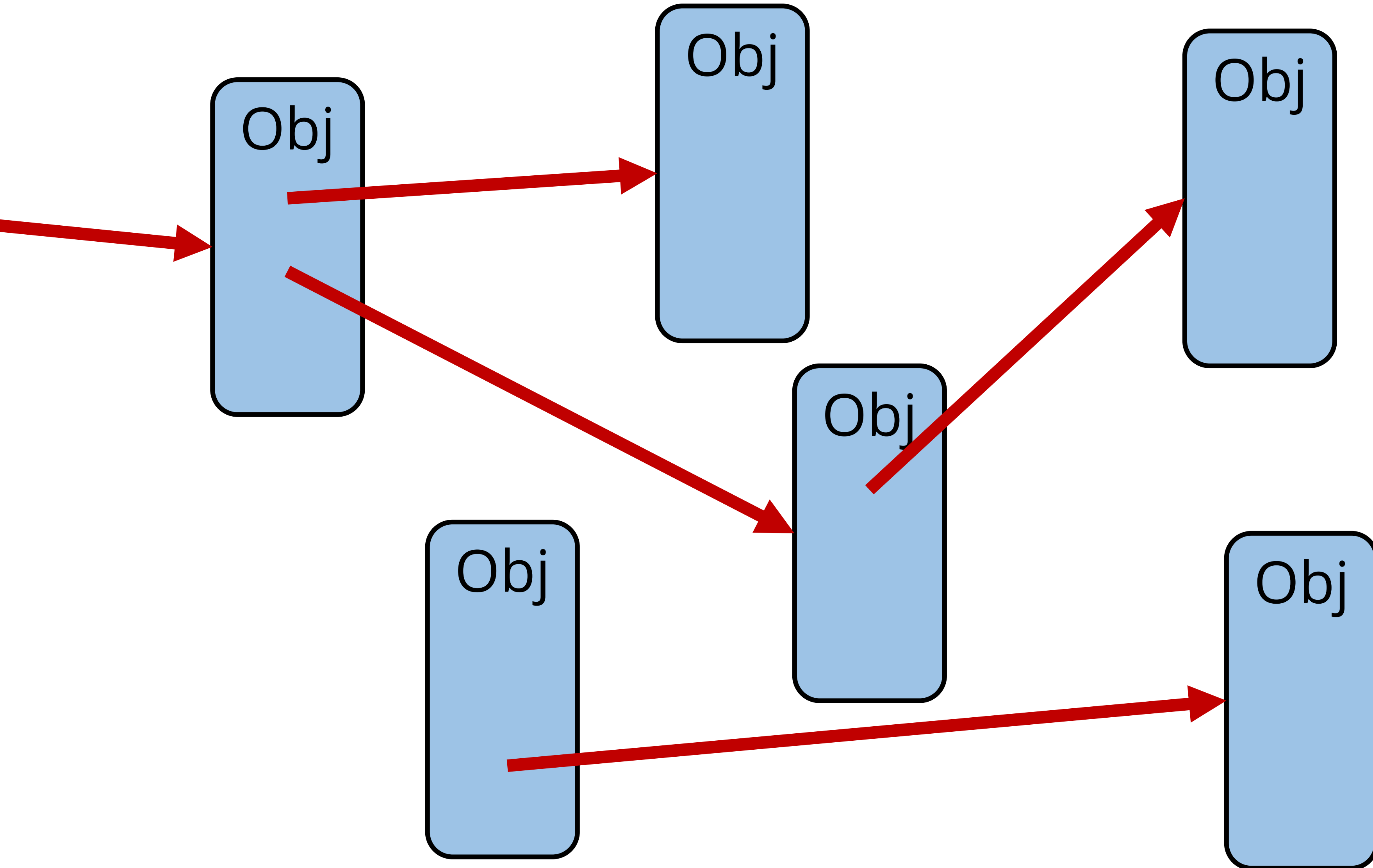
# Performance

- GC technique affects performance
- Application affects performance
- Environment affects performance
- “5% improvement” is the GC mantra
  - (GC should take negative time by now)

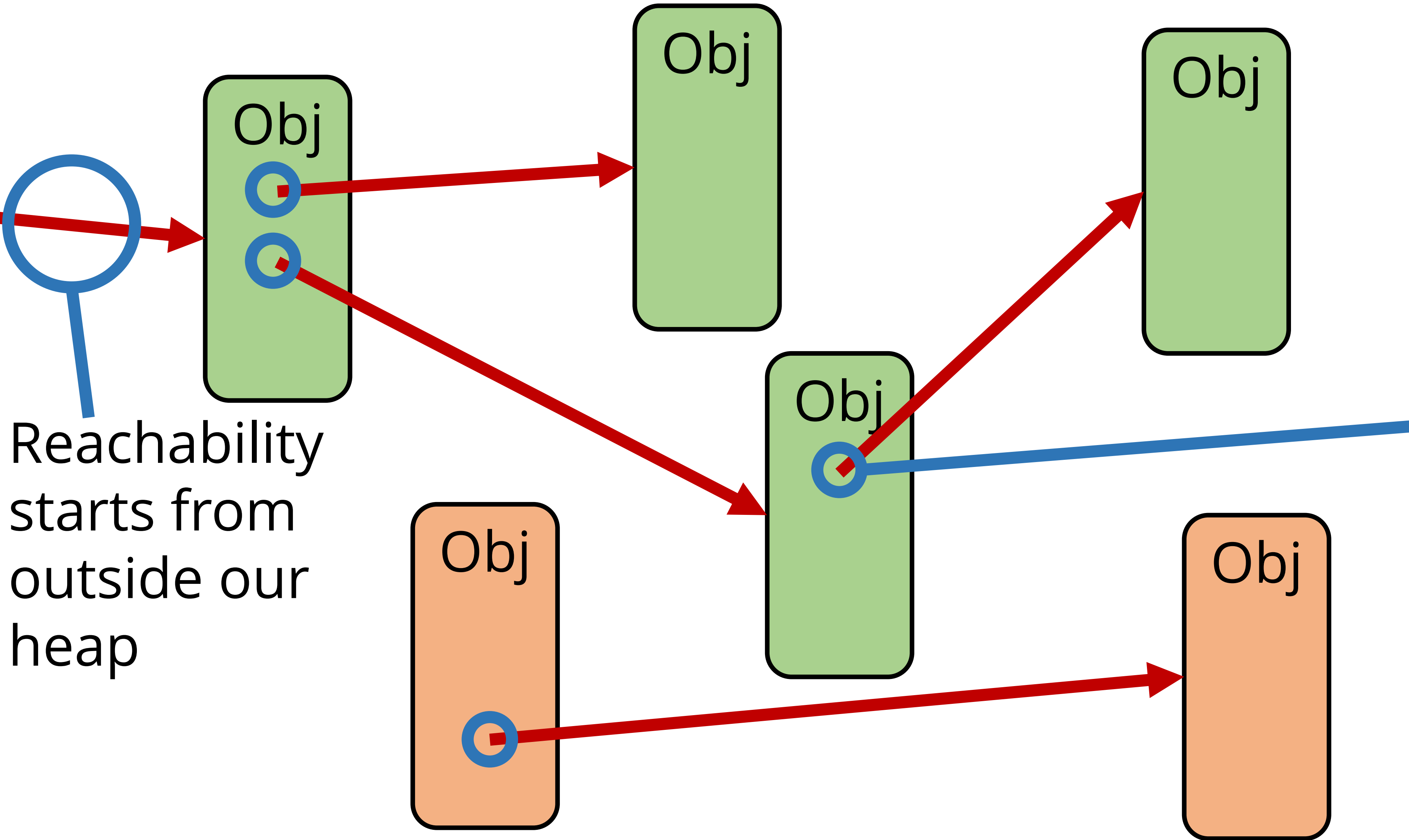
# When to free?

- What we want:
  - Free an object when we’re done with it
- What does “done” mean?
- What we do:
  - Free an object when it’s unreachable

# Reachable?



# Reachable?




Must know where objects have references (pointers into the GC heap)

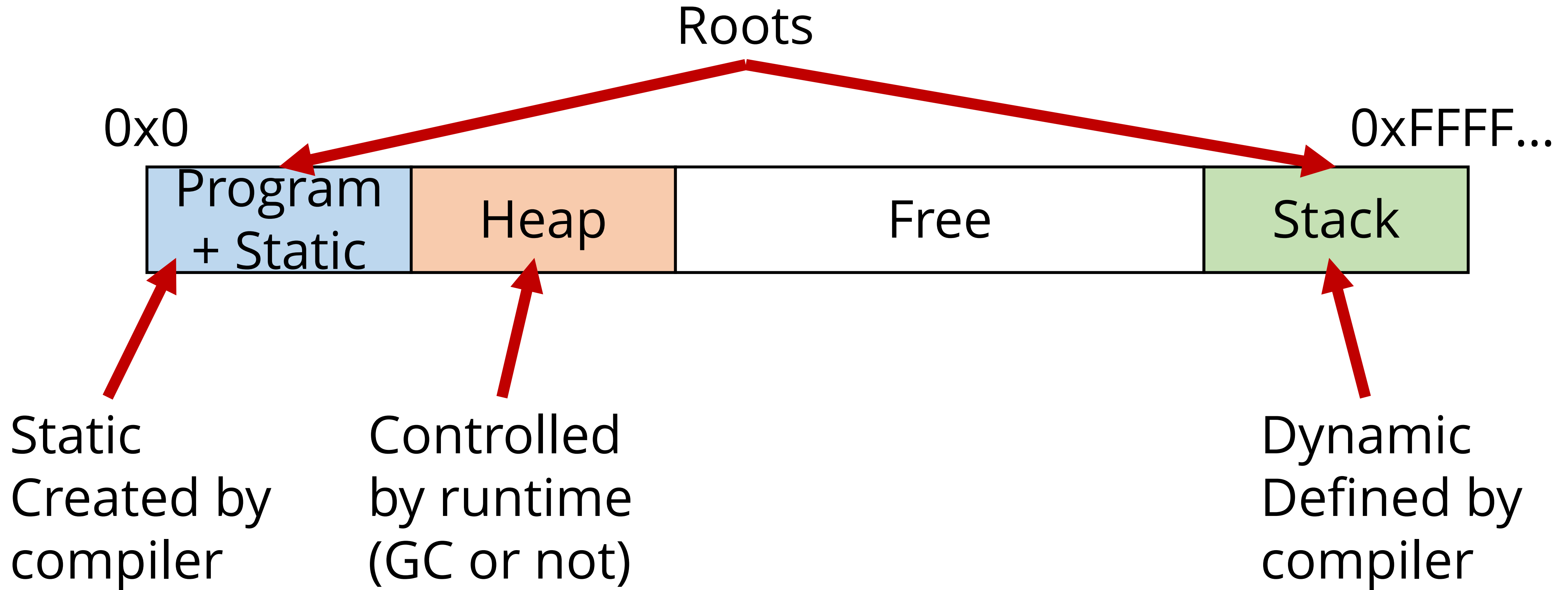
# Aside: Can we do better?

- Easy to leak memory:

```
static List<Object> everyObjectIveEverAllocated;
```

- “No longer in use” → halting problem
- Fear not the halting problem:
  Liveness analysis is very real! But not general.

# Roots



# The compiler conundrum

- Roots are full of stuff
  - (Program code, non-reference variables, unused space…)
- To test reachability, we need *references*!
- Compiler must cooperate with GC: Tell the GC where references exist in roots

# Stack references

Split stack:

```
; need 8 bytes stack
sub $8, %rsp

; and 8 bytes
; “pointer stack”
sub $8, %rbp
```

Marked stack:

```
; need 16 bytes
sub $16, %rsp

; tell GC about
; pointers
mov $8, %rax
call pushGCPointers
```

# Conservativism

- Some languages just won’t play nice (I’m looking at you, C)
- We can at least *guess* where there are pointers

# Heap references

- `malloc(size)` isn't enough!
- We need *references*!
- Need no more type information than that
- Crucial runtime type info: Pointer bitmap
- Extend header with pointer to type info

# Allocation w/ type info

```
struct ObjectHeader {
  struct GCTypeInfo *typeInfo;
};

void *allocate(struct GCTypeInfo *typeInfo) {
  size_t size = typeInfo->size;
  // allocate as usual
  retHeader->typeInfo = typeInfo;
  return ret;
}
```

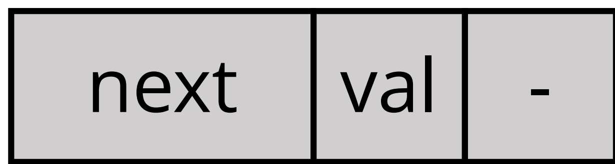
# Aside: Alignment

- Most systems require word-aligned pointers
- It therefore makes sense to word-align objects
- We only care about pointers, so only need one bit per word type info

# Heap references

```
class IntList {
  IntList next;
  int val;
};
```

Compiler




1 word .5w .5w

2 words

```
struct GCTypeInfo {
  size_t size;
  unsigned long pointerMap;
};
```

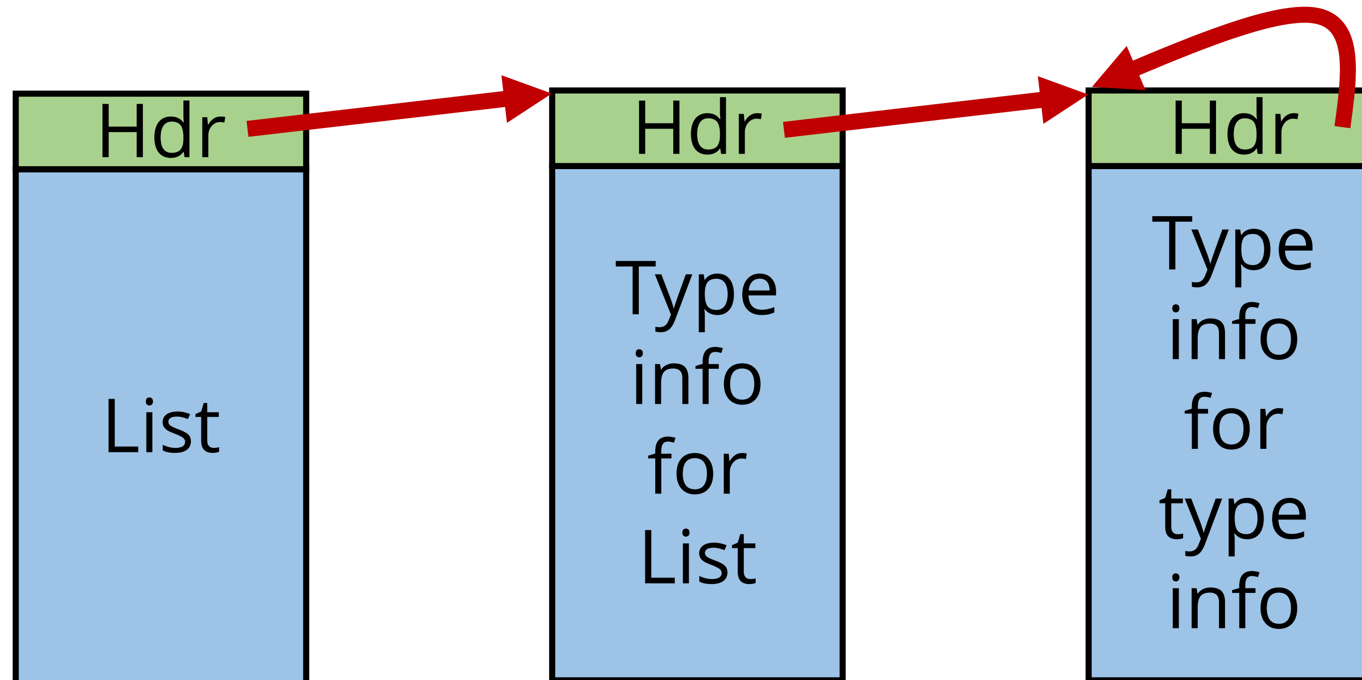
Compiler

```
new GCTypeInfo(16, 0b1000…);
```

# Aside: Dynamic type info

- All types known statically: Static type info
- What about types loaded at runtime?
- Type info object can be stored in the heap!

# Aside: Dynamic type info



# Breather

- Reachability
- Start with roots
- Compiler tells us references from roots
- Type info tells us references from objects

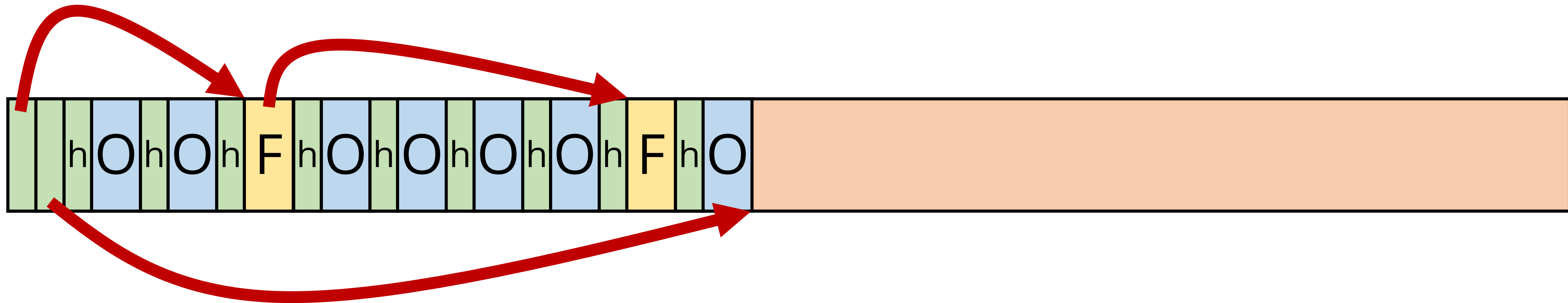
# Garbage collection

- We’ve determined what *is* reachable…
- What *isn’t* reachable is garbage!
- But… it’s not reachable
- So how do we free it?

# Parsable heap

- A heap is parsable if we can walk through all objects in it
- Parsability broken by gaps, lacking info or bugs

# Parsable heap




```
struct Pool {
  struct FreeObject *freeObjects;
  void *freeSpace;
};
```

# Collection

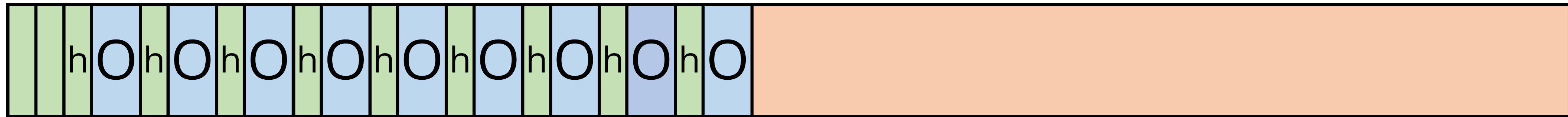
- While parsing the heap, how to tell which objects were unreachable?
- Add a “mark” field to headers, mark objects which are reached
- Unmarked objects are unreachable, so freed.

# Mark and sweep

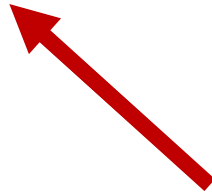
- That’s it! Now we have a mark and sweep garbage collector!

(it would be competitive in 1974)

# Moving



This part takes some effort…



Everything that remains is garbage!

# Moving

- Must be able to update all references
- Compiler tells us about references anyway
- But… isn’t moving memory expensive?

# Lifetime principles

- Most objects die young
- Therefore, more valuable to save time on (plentiful) dead objects than (few) living ones
- Moving can be worth it

# Stop-the-world

- When do we scan?
- If mutator still running, we can miss references
- So, stop the (mutator’s) world before GC
- Compiler implication: Yieldpoints

# Yieldpoint