Runtime interface

Review

- Unreachable objects dead
- Sweep, copying, compacting, ref counting
- Combine with partitioning, generational
- Most objects die young, so partitioning by age universal
- Also useful to partition large objects, threads, executable “objects”, etc

Runtime interface

- Compiler, memory manager and (possibly) programmer must agree
- Compiler motivated by language, so
- memory manager often motivated by language too
- GGGGC's interface is simple by design

Runtime interface

- Important components:
  - Allocation
  - Where references are in roots, objects
  - When collection can occur
  - Barriers on writing to (reading from?) objects

Allocation

Allocation

C `malloc`
"Just give me some bytes."

Memory manager needs no type info.

Object returned full of garbage.

Allocation

C `malloc`
"Just give me some bytes."

Memory manager needs no type info.

Object returned full of garbage.

Haskell allocation
"I promise not to touch!"

Memory manager knows all!

Object returned fully initialized, immutable.

Allocation

C `malloc`
"Just give me some bytes."

Memory manager needs no type info.

Object returned full of garbage.

Java `new`
"Good enough is good enough."

Memory manager needs refs.

Object returned full of zeroes.

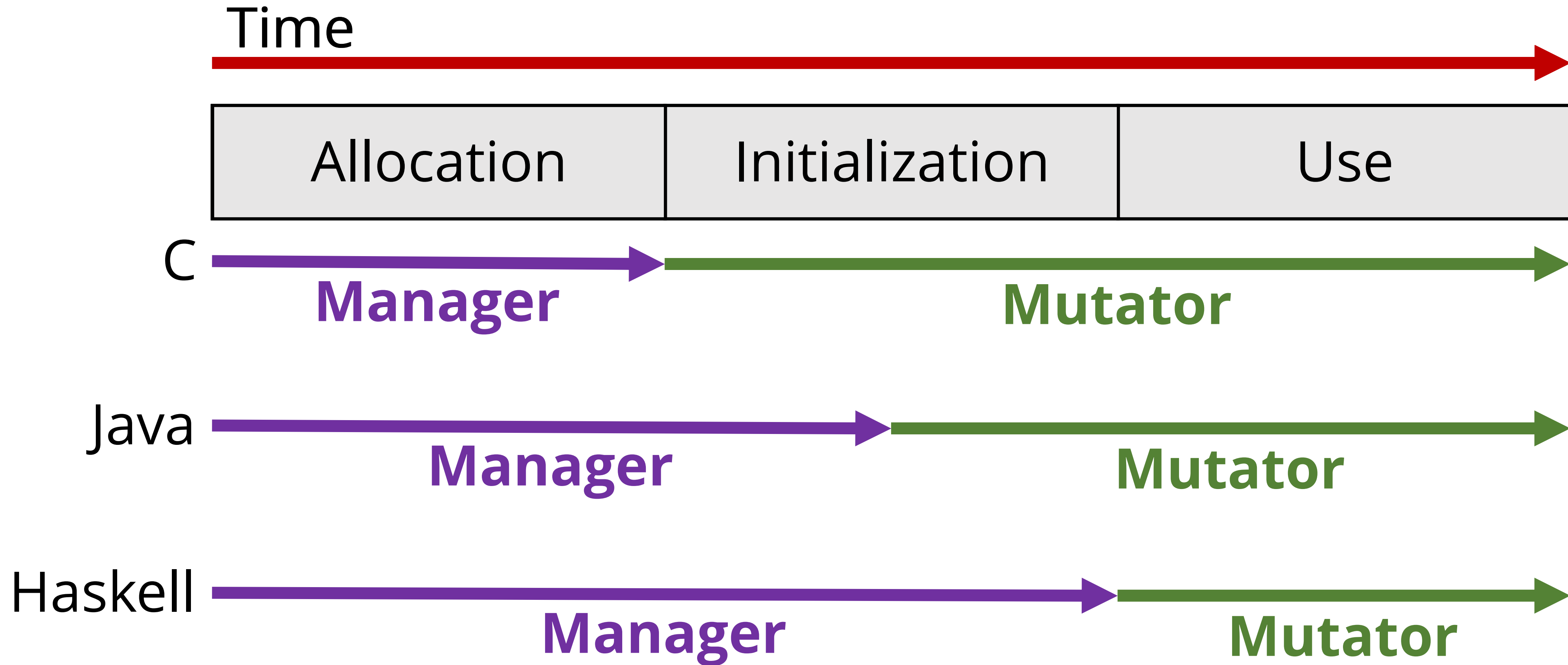
Haskell allocation
"I promise not to touch!"

Memory manager knows all!

Object returned fully initialized, immutable.

Allocation and initialization

Allocation and initialization



Constructor problem

- GC depends on “correct” objects
- “Correct” for us means: References refer to valid objects or **`NULL`**
- If an object scanned mid-initialization, it may not be correct!
- Options: Prevent GC mid-initialization, or initialize ourselves

Mid-initialization GC

- *If* initialization cannot allocate objects, we could prevent collection
- Does not generalize
- Initialization often not guaranteed correct anyway (just user code)
- Generally, manager must initialize

Zeroing

- Simple initialization: All bits zero
- Mundane value for virtually all types
- Simple initialization for manager
- Consequence: Objects initialized twice!
  - (Once by manager, once by program)

When to zero

When to zero

During allocation

Makes allocation slower.

Inefficient to spread work over many allocations.

When to zero

During allocation

Makes allocation slower.

Inefficient to spread work over many allocations.

During collection

Makes collection (much!) slower.

Impossible with free-list.

Efficient.

When to zero

During allocation	Ahead of allocation	During collection
Makes allocation slower.	Zero “chunks” beyond current object during allocation.	Makes collection (much!) slower.
Inefficient to spread work over many allocations.	Choose chunk size wisely (cache line or page) for efficiency.	Impossible with free-list.
		Efficient.

Runtime interface

- Important components:
  - ~~Allocation~~
  - Where references are in roots, objects
  - When collection can occur
  - Barriers on writing to (reading from?) objects

How to identify references

How to identify references

Conservative

“If it looks like it might be a reference, it’s a reference.”

To be discussed later.

How to identify references

Conservative

"If it looks like it might be a reference, it's a reference."

To be discussed later.

Precise

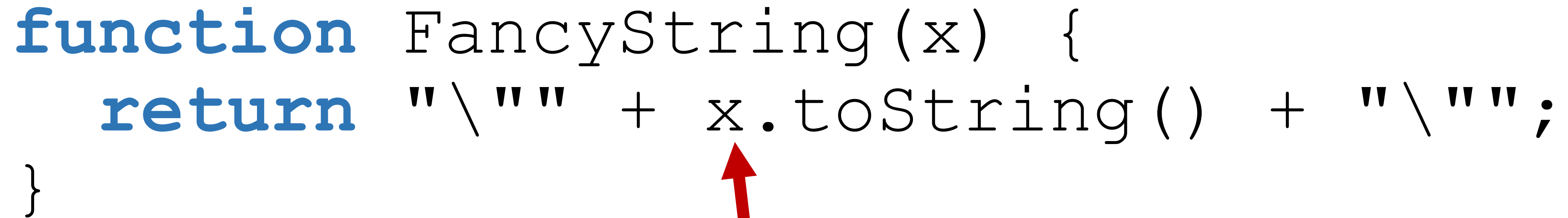
Compiler must know location of all references and tell the GC.

Usual for statically-typed languages.

How to identify references

Conservative	Tagged	Precise
“If it looks like it might be a reference, it’s a reference.”	References stored differently from data. Known only at runtime.	Compiler must know location of all references and tell the GC.
To be discussed later.	Usual for dynamically-typed languages.	Usual for statically-typed languages.

Tagged references

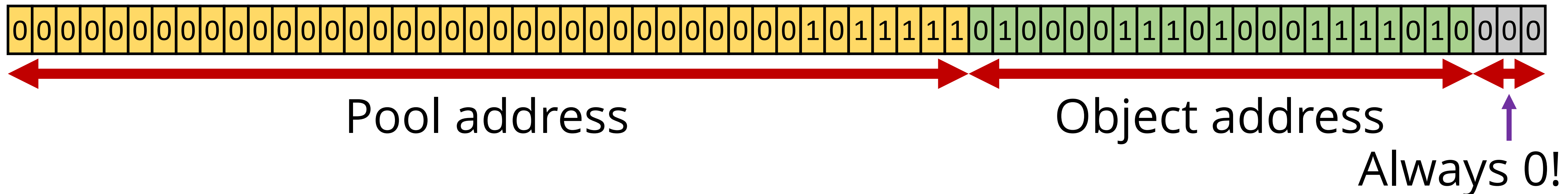


This code works whether x is a reference or not

(That's JavaScript)

Refresher: Bit-sneakiness

- There are three[1] wasted bits in our header



- *All* pointers have this extra space!

[1] On 32-bit systems, two

Tagged references

- Use extra bits as type
- e.g.: References end in 000,
  Integers end in 1,
  Strings end in 010,
  …
- Compiler and GC must agree
- Compiler must untag values to use them

Tagged references

```
trace(obj):
  for loc in obj to obj + (size)
    if (*loc & 0b111) == 0:
      …
      trace(*loc)
      …
```

Precise object references

- You’ve seen plenty of this!
- Bitmaps is pretty much how it’s done
- In some (usually functional) language, object type includes ‘trace’ function:
  - Compiler creates trace function per type
  - GC calls trace function referred to in descriptor

Stack references

- Pointer stacks (ala GGGGC) is *not* how it's usually done
- Options:
  - Secondary reference stack
  - Parsable stack
  - Heap-allocated stack
  - Object-shaped stack

Secondary reference stack

- Normal stack has no references, second stack has only references

Ref stack

Secondary reference stack

- Normal stack has no references, second stack has only references

```
foo()
```

Secondary reference stack

- Normal stack has no references, second stack has only references

`foo() bar()`

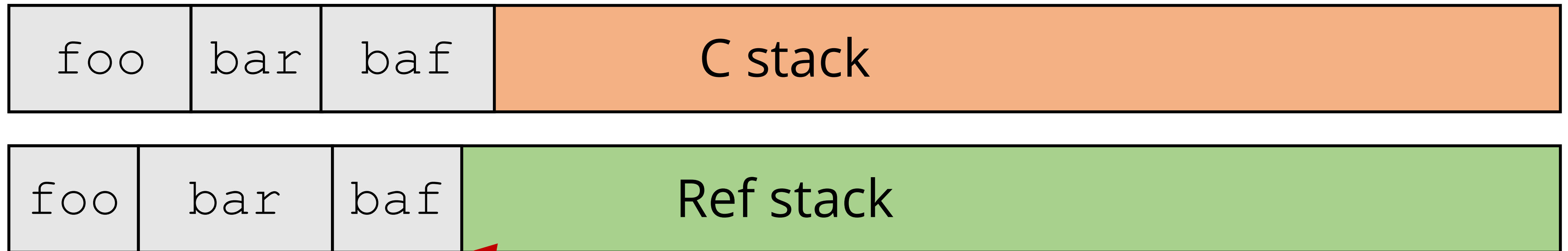
Secondary reference stack

- Normal stack has no references, second stack has only references

`foo() bar() baf()`

Secondary reference stack

- Normal stack has no references, second stack has only references



Reference stack caveats

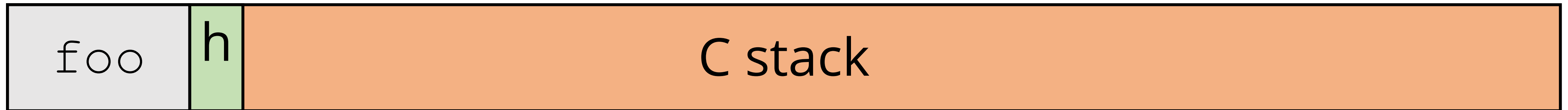
- Usually needs reference stack register
- CPU registers are rare!
- Complicates other parts of compiler, e.g. exception handling

Parsable stack

- Like parsable heap: Make sure GC can find info in stack

Parsable stack

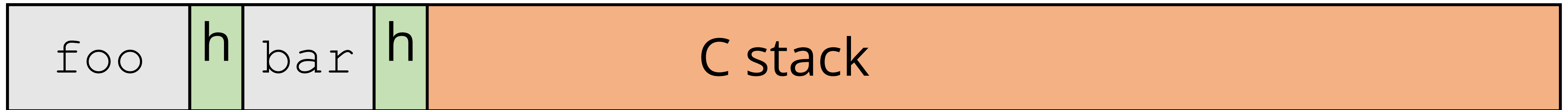
- Like parsable heap: Make sure GC can find info in stack



`foo()`

Parsable stack

- Like parsable heap: Make sure GC can find info in stack



`foo() bar()`

Parsable stack

- Like parsable heap: Make sure GC can find info in stack

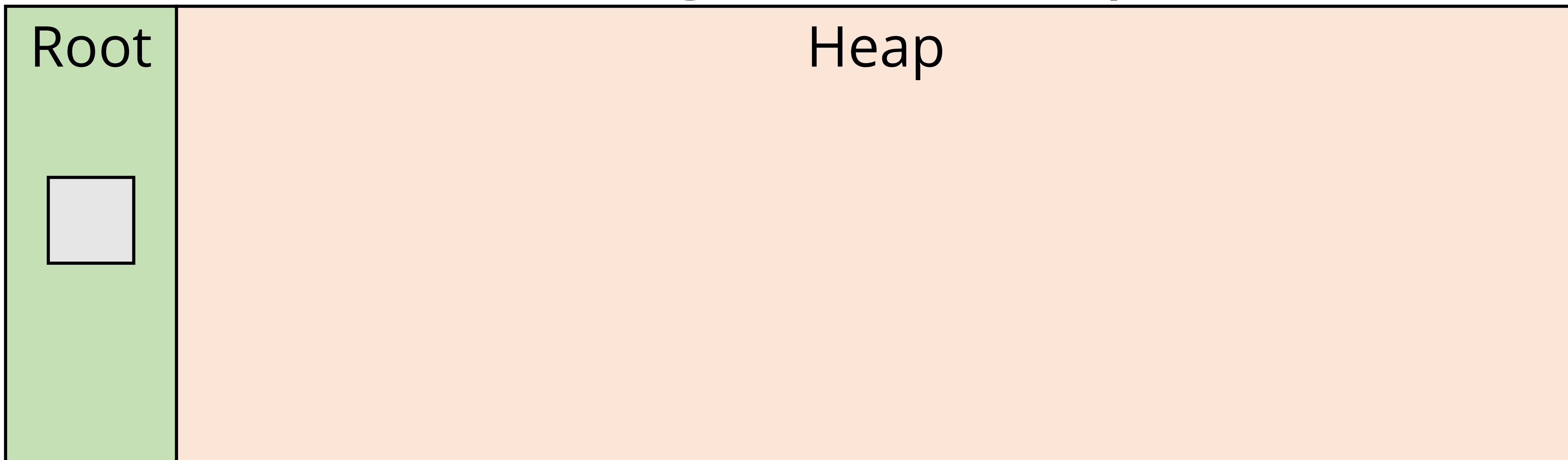
`foo() bar() baf()`

Parsable stack caveats

- Does not play nicely with others (C, C++)

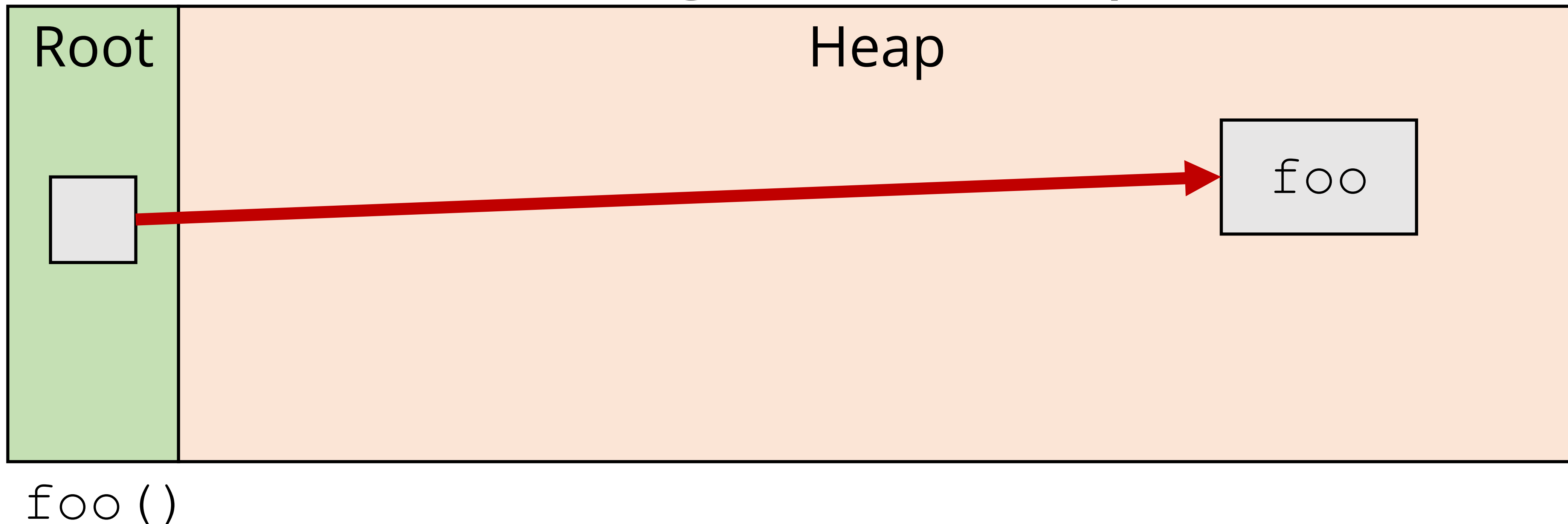
Heap-allocated stack

- Every function allocates a stack “object”, chains those together into pseudo-stack



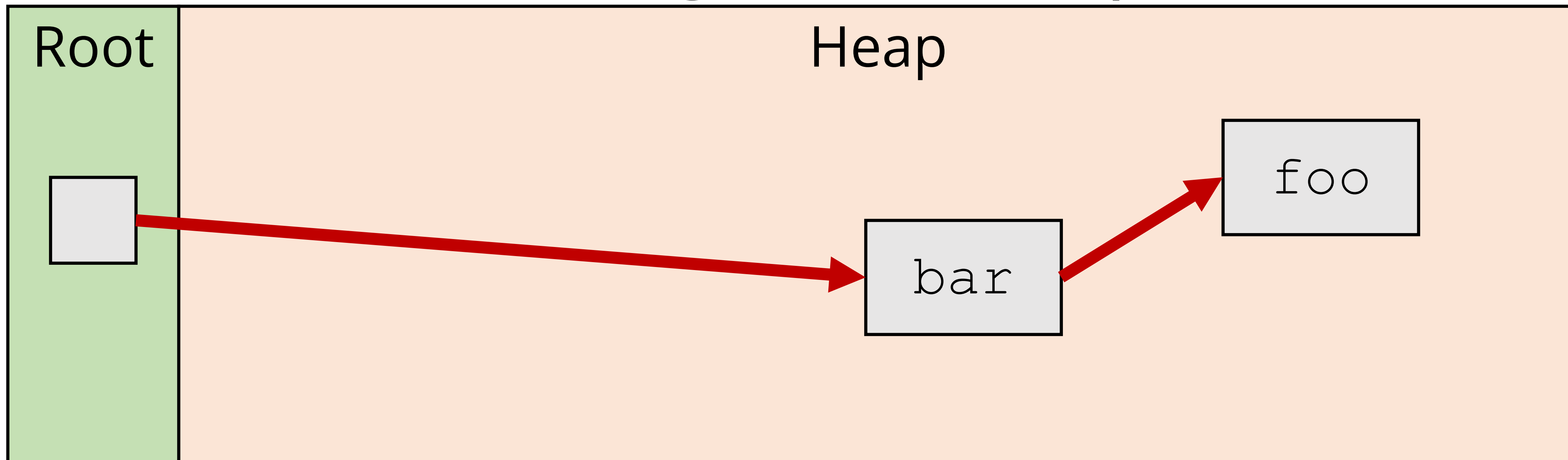
Heap-allocated stack

- Every function allocates a stack “object”, chains those together into pseudo-stack



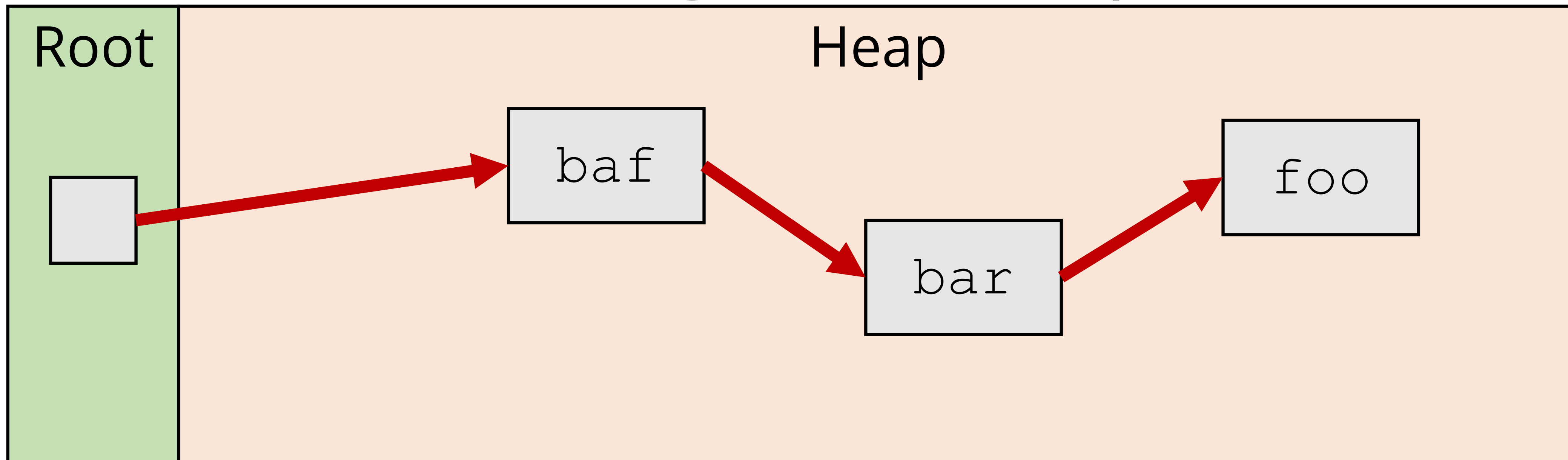
Heap-allocated stack

- Every function allocates a stack “object”, chains those together into pseudo-stack



Heap-allocated stack

- Every function allocates a stack “object”, chains those together into pseudo-stack



Heap-allocated stack caveats

- Stack frames outlive function execution
  - Easy closures!
- Need descriptors for every possible stack frame
- Restrictive to compiler

Object-shaped stack

- Exactly like heap-allocated stack, but allocate stack frame objects on stack
- Caveat: GC must expect objects in stack
- Stack is now a partition

What is a reference?

- Thusfar: Reference is pointer to beginning of object
- Possibilities to consider:
  - Interior pointers
  - Indirect pointers

Interior pointer

```
struct List {
  struct List *next;
  int val;
};

void breakThings(struct List *l) {
  int *nasty = &l->val;
  l = l->next;
  // yield to GC
  printf("%d\n", *nasty);
}
```

Interior pointers

- Usual solution: Not allowed
- When allowed:
  1. Get chunk (e.g. card) associated with interior pointer
  2. Parse chunk to find containing object
  3. If moving, preserve offset

Object tables

- Alternative contract between compiler and memory manager
- Instead of program using (e.g.)

```
struct List *
```

  always use (e.g.)

```
struct List **
```

Object tables

- Accessing object is double-dereference
- User pointer refers to object table entry,
- object table entry refers to object.
- Object table forms its own partition
- Object table entries don’t move

Object table thoughts

- Slows down object access
- Allows *atomic object replacement*
  - i.e., one object may replace another
- Makes moving objects simple (only one ref to update)
- Makes mark-and-compact require only one sweep!

Runtime interface

- Important components:
  - ~~Allocation~~
  - ~~Where references are in roots, objects~~
  - When collection can occur
  - Barriers on writing to (reading from?) objects

Any-time collection

- Threads may be doing anything at any time
- Can we stop mutator at any time to collect?
- Think about stack references: Mutator must never deviate from description
- Very limiting to optimizations

Yieldpoint collection

- Compiler adds “yieldpoints” to allow collection
- Between yieldpoints, stack in inconsistent state
- To stop, wait for all threads to reach yeildpoint

When to yield

- Mandatory: During allocation, at function calls
- Optional: At tight loops or other long-running operations
- Threads must yield frequently to allow GC to occur promptly

How to yield

- Polling:

```
void yield() {
  if (stopTheWorld)
    collect();
}
```

- Patching: Overwrite the yield function with a version that stops the thread

Runtime interface

- Important components:
  - ~~Allocation~~
  - ~~Where references are in roots, objects~~
  - ~~When collection can occur~~
  - Barriers on writing to (reading from?) objects

Write barriers

- Either reference counting or inter-partition remembering
- For inter-partition remembering:
  - Guard (check that ref must be remembered)
  - Remember (e.g. write bit for card)

Write barrier guard

```
write(obj, loc, val):
  if genOf(obj) == old and
     genOf(val) == young:
    …
```

- genOf isn’t cheap (bit math to get pool, read gen from pool header)

Arranging for better barriers

- Sometimes can ask for pools at certain locations in memory
- Approximate generation check with location check
- Imperfect guard means more remembered set writing, but less time in write barrier

Sequential heap guard

```
write(obj, loc, val):
    if obj > val:
        …
```

- Write barrier can trigger improperly, but
- the check is (usually) one CPU operation
- Young gen must have (pointless) remembered set

Unguarded write barrier

```
write(obj, loc, val):
  poolOf(obj)->(remember obj)
  *loc := val
```

- All writes more expensive,
- remembered set less precise,
- but write time is predictable.

Runtime interface

- Summary:
  - Allocation
  - Where references are in roots, objects
  - When collection can occur
  - Barriers on writing to (reading from?) objects