5118014 Programming Language Theory

Ch 13. Garbage Collection

Shin Hong

Memory Management

- Operational semantics defines the functional behaviors of a programming language, but does not reflect the execution on real machines
  - the memory capacity of a real machine is finite, not infinite

- The notion of memory management of a programming language defines how memory addresses of boxes are given, and how they are reused in executions

Stack and Heap

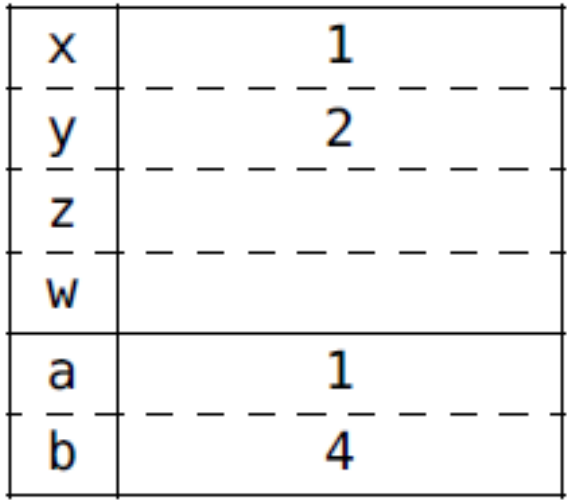
- Stack
  - a part of memory to store local variables of functions
    - when a function call is invoked, a stack frame is created and pushed onto the stack
    - a local variable is created when a function call starts, and it is removed when the function call finishes and returns
  - similar to an environment in MFAE

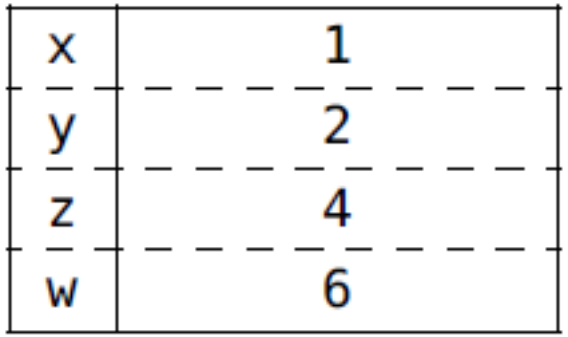
- Heap
  - a part of memory to store variables whose lifetime is not the same as a function call
  - similar to a store in MFAE

Example

```
def f() = {
  val x = 1
  val y = 2
  val z = g(x)
  val w = y + z
}

def g(a: Int): Int = {
  val b = a + 3
  return b
}
```





Heap

- The purpose of heap memory
  - to store large-size data objects
  - to store data objects that must survive across different function call executions

- A memory manager is a run-time system for managing heap memory
  - when a program requests, the memory manager allocates free space and place the value, and then deallocates the allocate spaces when they become useless
  - each language has its own mechanism to choose whether a variable is stored on stack or heap
    - by programmer's choice: C, C++, and Rust
    - by pre-defined rules: Java, Scala

Garbage

- Useless objects that just take up memory spaces are called garbage
  - the presence of a garbage object after a certain moment in an execution does not affect the execution
  - With too many garbage objects, the program may fail since it cannot allocate memory spaces for new heap objects

- The memory manager deallocates garbage to recover the memory space
  - manual memory management
  - automatic memory management

```
def f() = {
  for (i <- 1 to 300) {
    g(i)
    ...
  }
}

def g(a: Int) = {
  val b = C(1, 2, 3, a)
  ... // do something with `b`
}
```

Manual Memory Management

- Heap deallocation (e.g., free()) is programmed by developers
  - e.g., C, C++

- Pros
  - memory space can be efficiently managed without much runtime overhead
- Cons
  - Incorrect programming may introduce memory errors
    - a use of dangling pointers may result in Use-After-Free (UAF)
    - memory leaks happen when deallocation is missing out

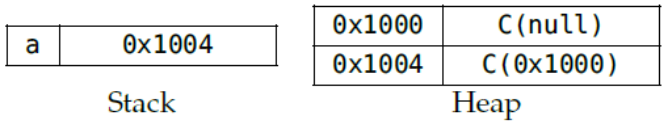
Automatic Memory Management

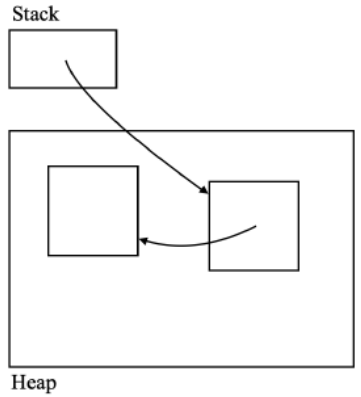
- Garbage Collector (GC) finds and deallocates garbage objects automatically
- GC determines a certain object is garbage or not by checking reachability
  - an object is reachable if a program can reach the object from the stack by following references
  - GC can prevent UAFs completely, but cannot prevent memory leak completely
- Example

```
case class C(x: Int)

var a = C(0)
a = C(1)

... // never use `a`
```



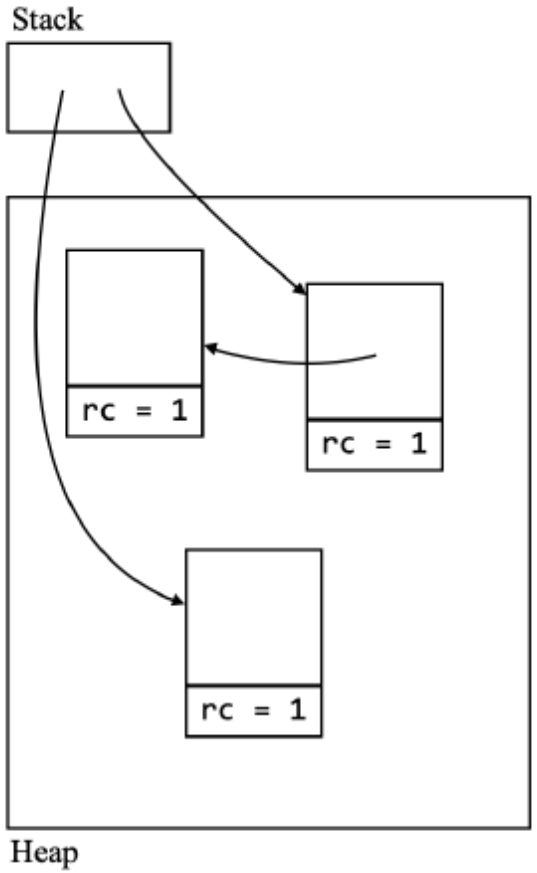


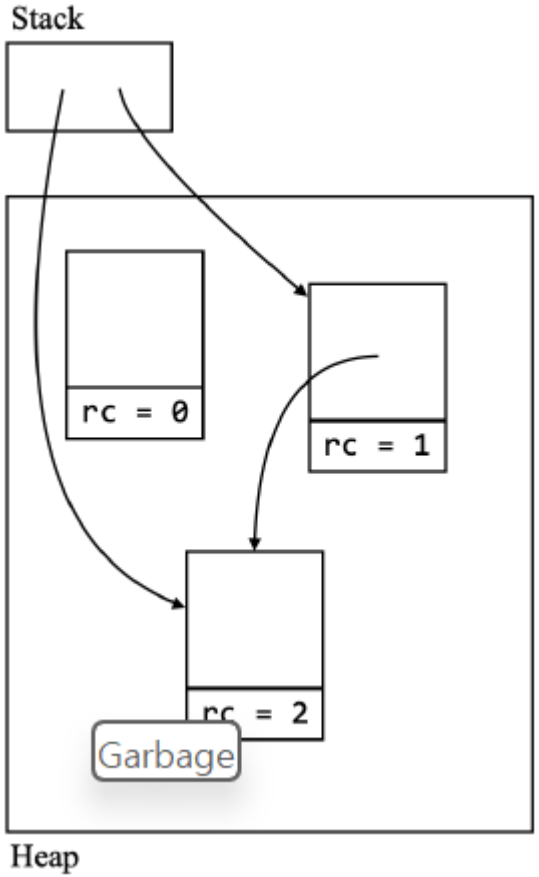
Reference Counting

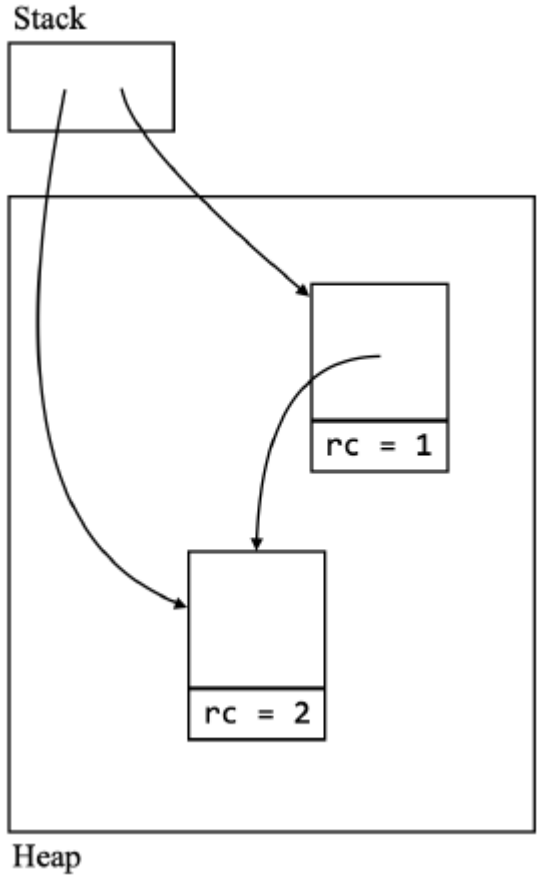
- Each object has a reference count which holds the number of references referring the object at a moment
  - when an object is created, its reference count is set to be zero
  - each time a reference to the object is created/destroyed, the count increases/decreases by one

- The object is considered as unreachable when its reference count is zero, thus deallocated immediately

Example


Stack
rc = 1
rc = 1
rc = 1
Heap


Stack
rc = 0
rc = 1
rc = 2
Garbage
Heap


Stack
rc = 1
rc = 2
Heap

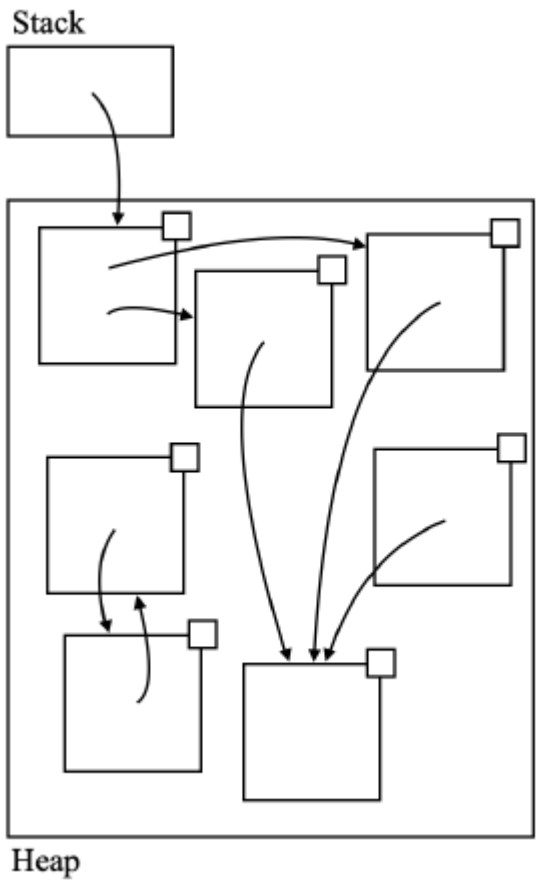
Reference Counting: Pros and Cons

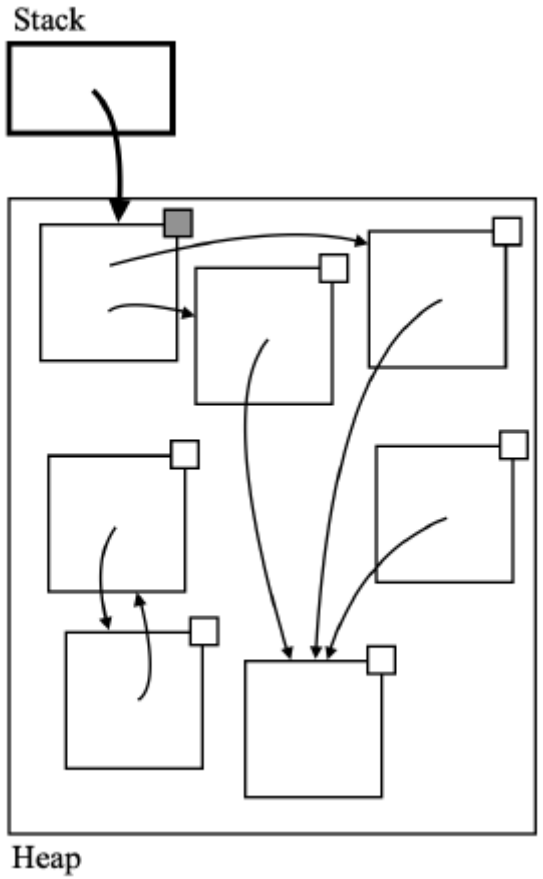
- Pros
  - Memory reclamation is immediate and takes a short time
- Cons
  - Cannot handle memory leaks by cyclic references
  - Requires free lists and suffers from external fragmentation

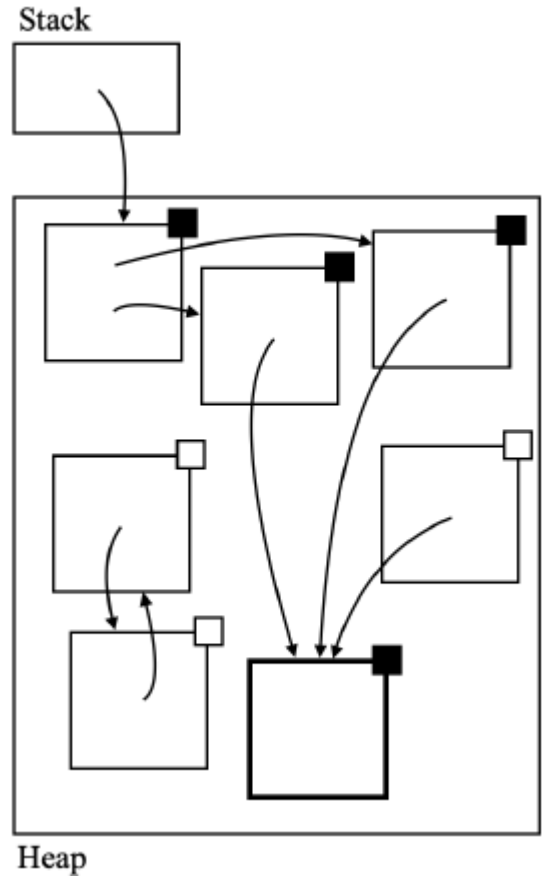
Mark-and-Sweep GC

- Marking: the memory manager marks all reachable objects
  - three possible states of an object
    - Unreached: This object has not been explored yet. It may be unreachable and is a candidate for deallocation. However, the state may change.
    - Unscanned: This object is reachable and will not be deallocated. It has a pointer to an Unreached object, so a further "scan" is required.
    - Scanned: This object is reachable and will not be deallocated. It does not have any pointers to Unreached objects.
- Sweeping: the memory manager deallocates all marked objects
- Mark-and-Sweep GC finds and deallocates all unreachable objects at once only when GC is triggered (e.g., when the heap is full)

Example







Mark-and-Sweep GC: Pros and Cons

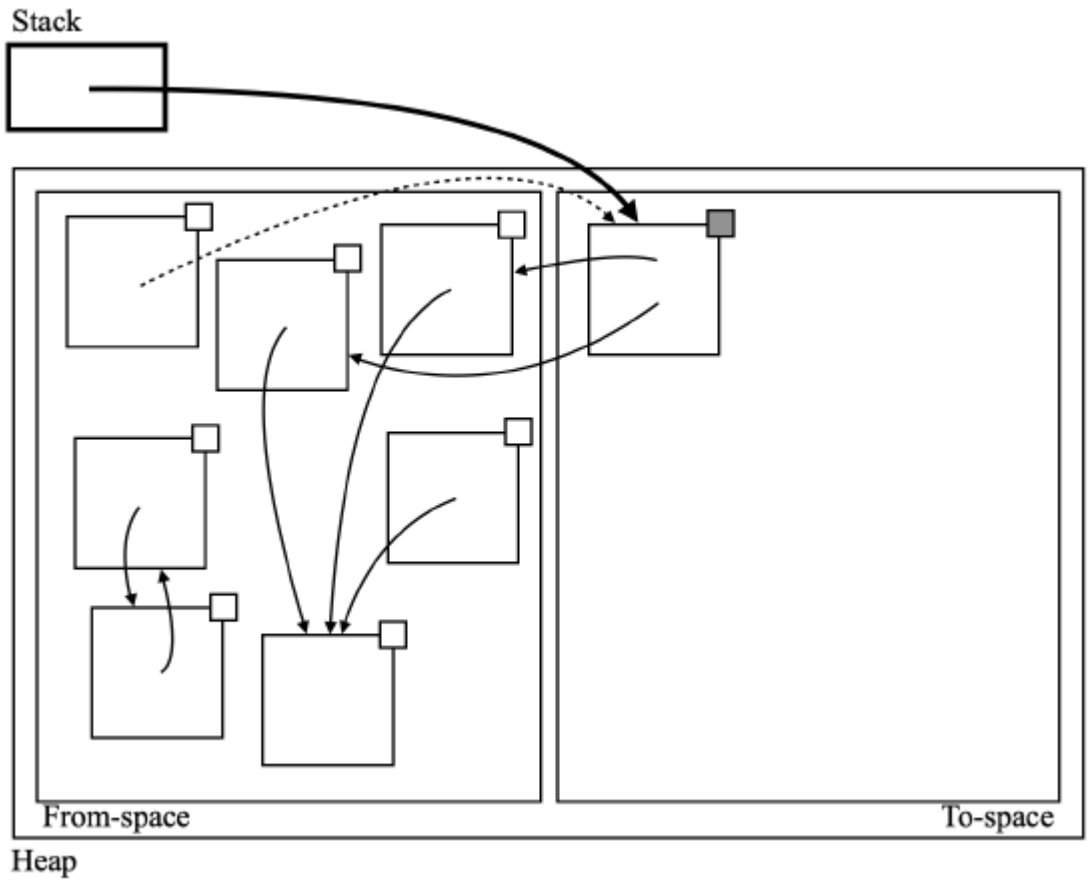
- Pros
  - GC can handle cyclic structures

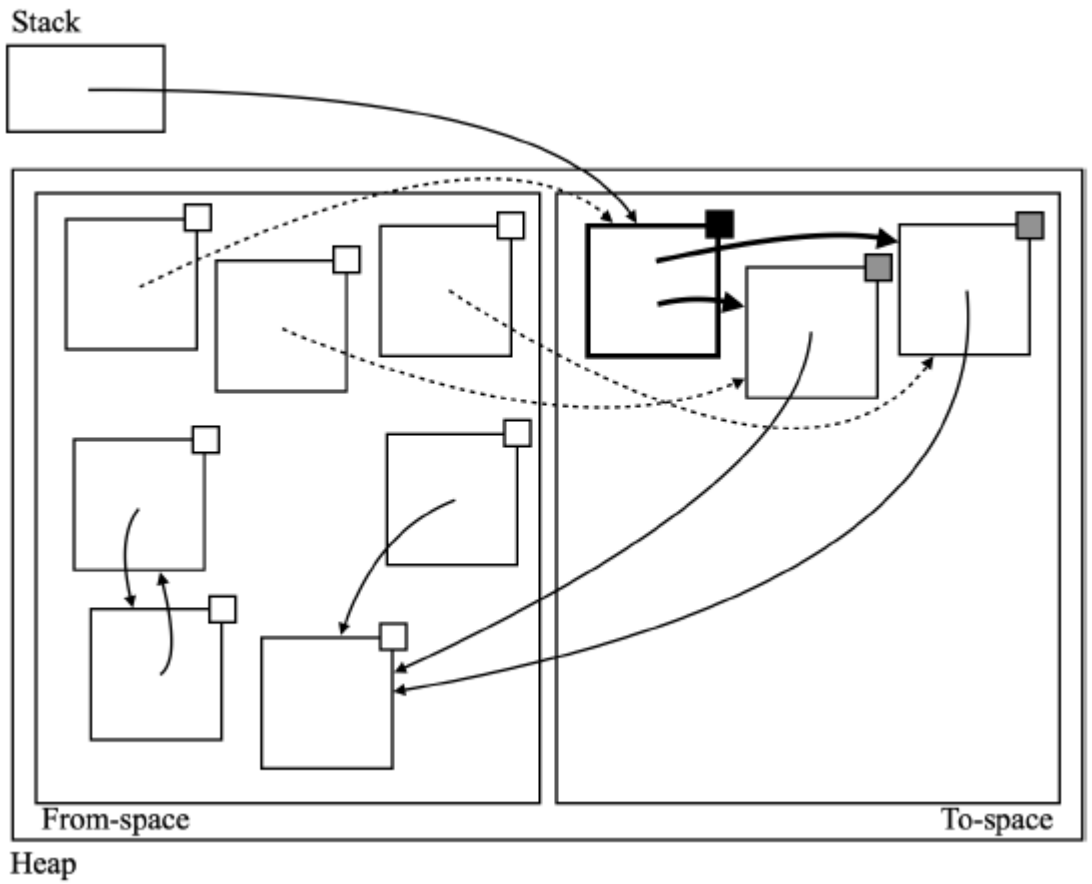
- Cons
  - Execution pauses during GC
  - Requires free lists and suffer from external fragmentation

Copying GC

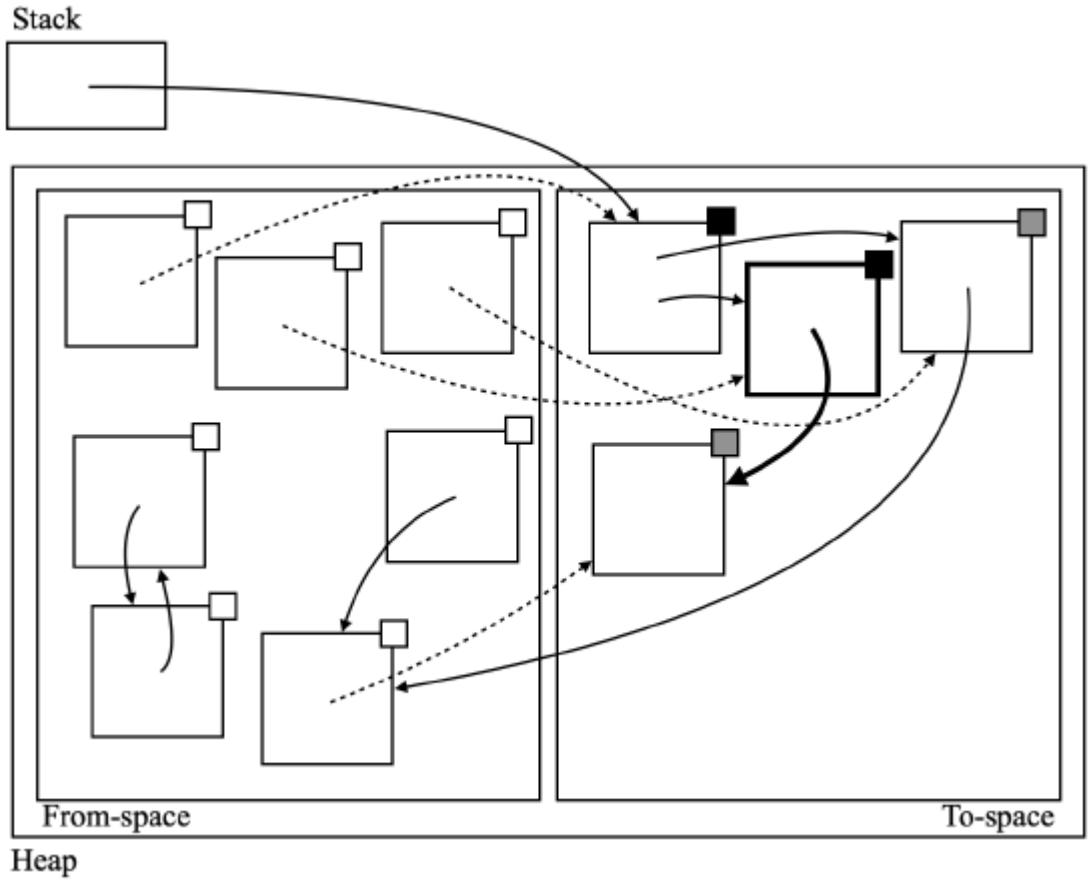
- When it is triggered, GC finds all reachable objects and reorganize them in a compact layout, and deallocates all unreachable objects
- Divide the heap into two parts: from-space and to-space
  - every allocation happens in the from-space
  - at the marking phase, GC marks a reachable object as Unscanned and copies the object from the from-space to the to-space
  - when copying an object, all existing references to the object are updated
  - If there is no Unscanned object, GC defines the from-space as the to-space, and the to-space as the from-space

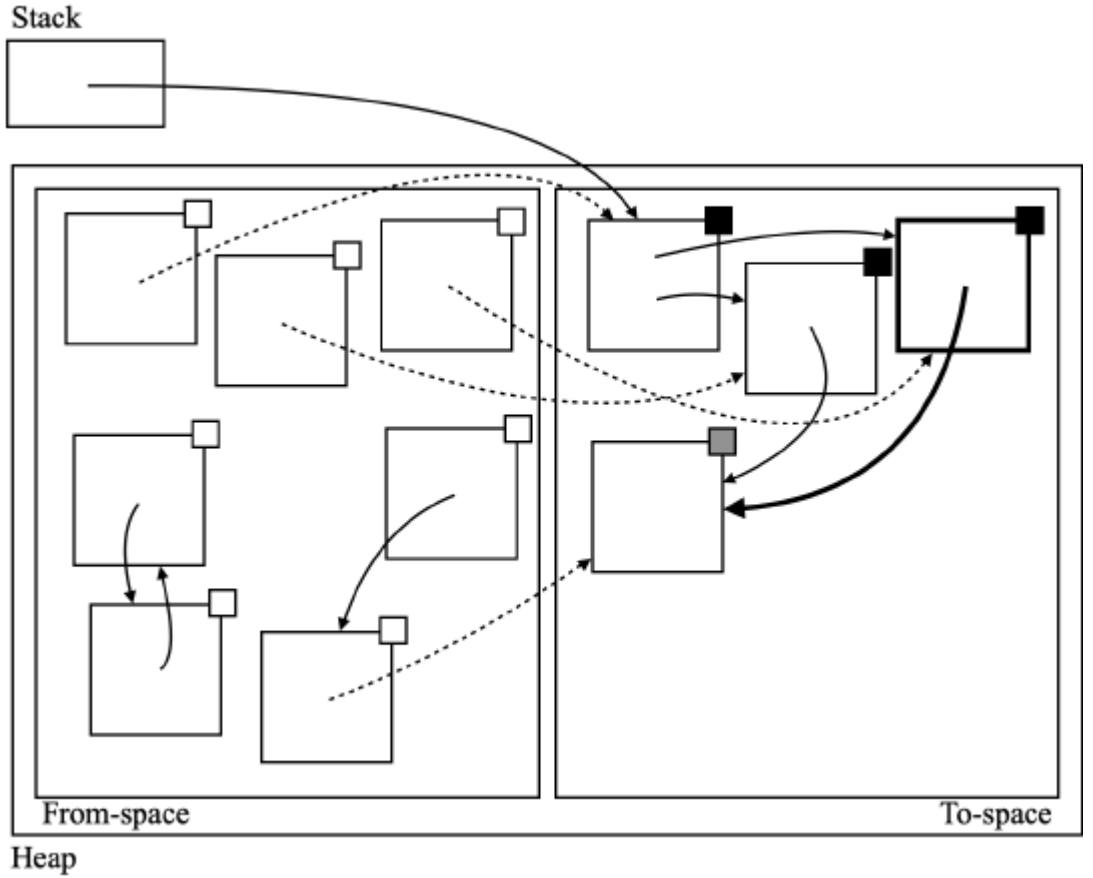
Example


Stack
From-space
To-space
Heap

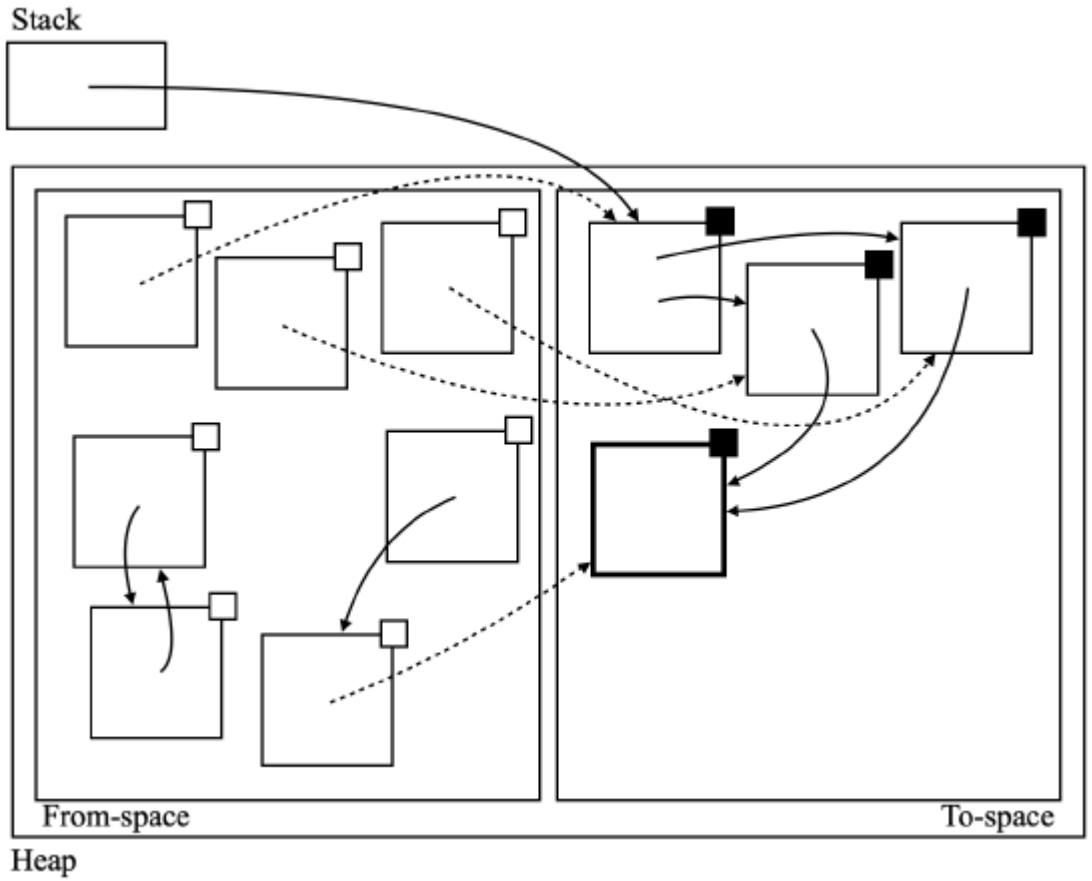

Stack
From-space
To-space
Heap

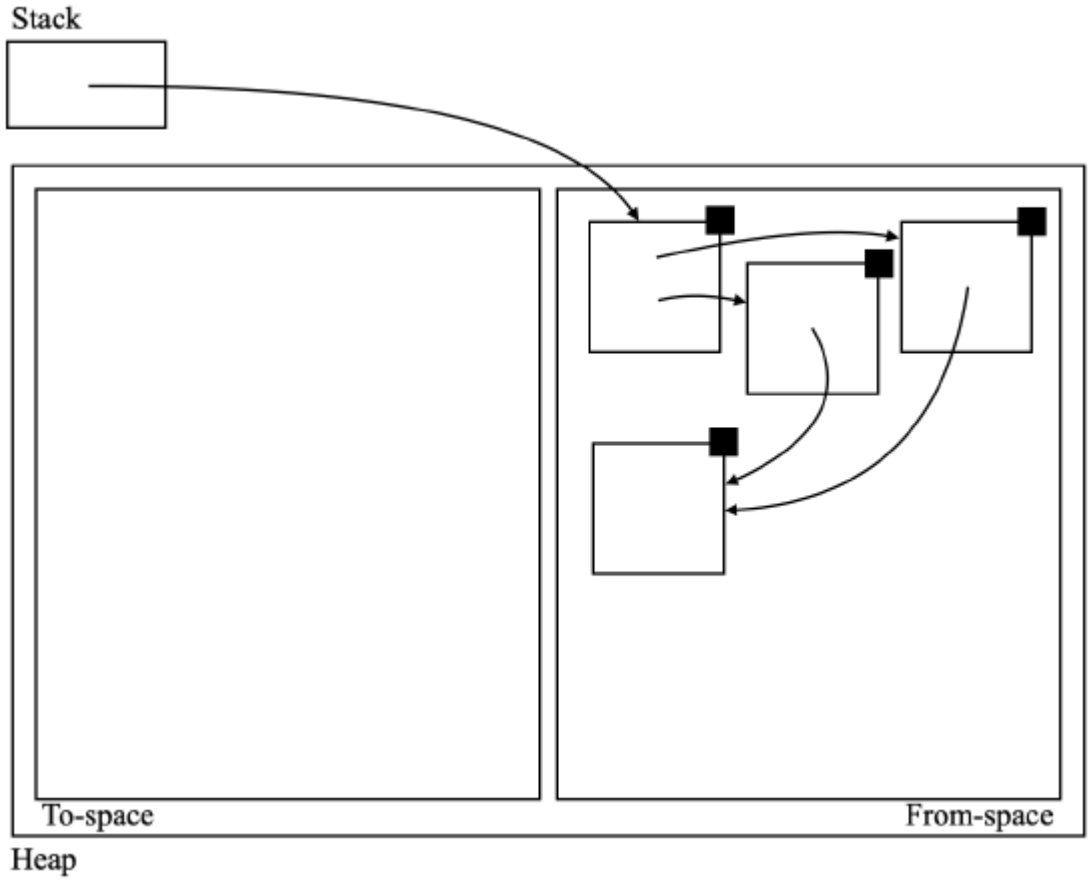
Example


Stack
From-space
To-space
Heap


Stack
From-space
To-space
Heap

Example


Stack
From-space
To-space
Heap

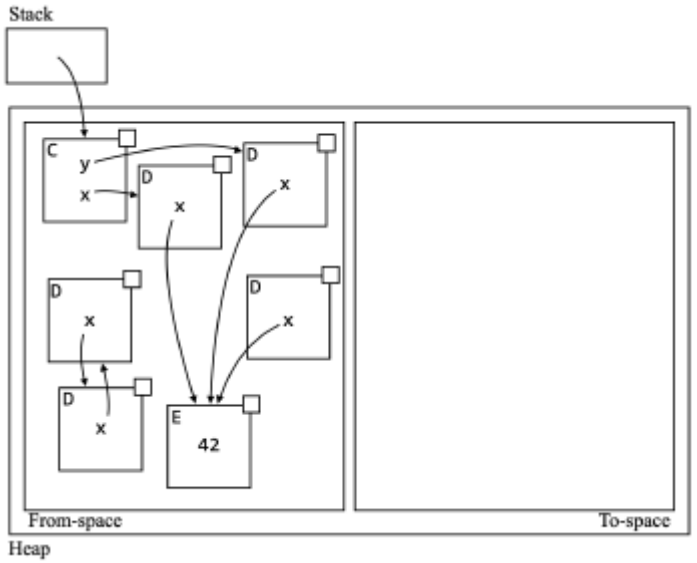

Stack
To-space
From-space
Heap

Cheney’s Algorithm

- copy reachable objects starting from the stack in BFS manner
  - copy an object reachable from stack or an object in the to-space, and update the reference

- maintain two pointers, scan and free in to-space
  - scan: the location of the first Unscanned object in the to-space
  - free: the first location of freely available memory in the to-space

```
case class C(x: AnyRef, y: AnyRef)
case class D(x: AnyRef)
case class E(x: Int)
```



Stack

0x02

From-space

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
D	0x07	C	0x05	0x0d	D	0x0b	D	0x00	D	0x0b	E	42	D	0x0b	

To-space

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
scan free															

Stack

0x10

From-space

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
D	0x07	F	0x10	0x0d	D	0x0b	D	0x00	D	0x0b	E	42	D	0x0b	

To-space

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
C	0x05	0x0d													
scan			free												

Stack

0x10

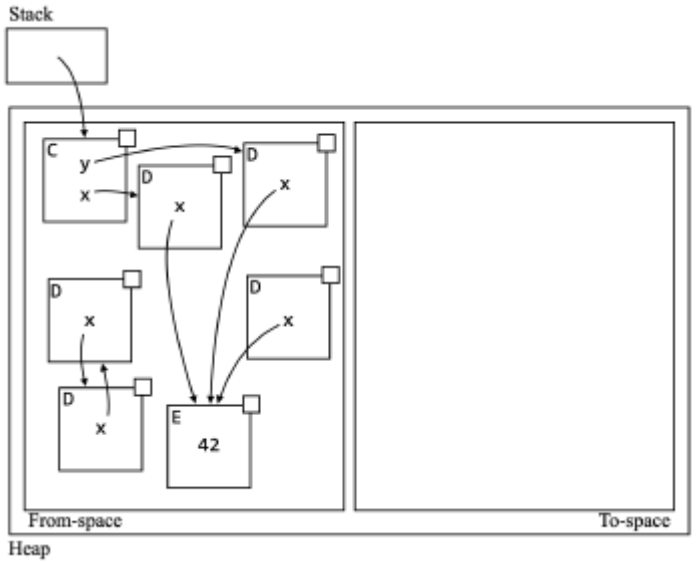
From-space

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
D	0x07	F	0x10	0x0d	F	0x13	D	0x00	D	0x0b	E	42	F	0x15	

To-space

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
C	0x13	0x15	D	0x0b	D	0x0b									
			scan				free								

```
case class C(x: AnyRef, y: AnyRef)
case class D(x: AnyRef)
case class E(x: Int)
```



Stack

0x10

From-space

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
D	0x07	F	0x10	0x0d	F	0x13	D	0x00	D	0x0b	F	0x17	F	0x15	

To-space

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
C	0x13	0x15	D	0x17	D	0x0b	E	42							
					scan				free						

Stack

0x10

From-space

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
D	0x07	F	0x10	0x0d	F	0x13	D	0x00	D	0x0b	F	0x17	F	0x15	

To-space

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
C	0x13	0x15	D	0x17	D	0x17	E	42							
							scan		free						

Stack

0x10

From-space

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
D	0x07	F	0x10	0x0d	F	0x13	D	0x00	D	0x0b	F	0x17	F	0x15	

To-space

0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
C	0x13	0x15	D	0x17	D	0x17	E	42							
									scan free						

Copying GC

- Pros
  - Handle cyclic structures and prevent memory leaks
  - Do not require free lists; do not suffer from external fragmentation
- Cons
  - Execution stops during GC
  - Copying incurs runtime costs
  - Only a half of the heap space can store objects