HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 25:
Garbage Collection

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Greg Morrisett

Announcements

- Embedded EthiCS assignment
  - Due: Friday Dec 29
  - Posted on Piazza
- HW6: Optimization and Data Analysis
  - Due: Tue Dec 3

Announcements: Upcoming Lectures

- Tuesday Dec 3: The Economics of Programming Languages
  - Evan Czaplicki '12, creator of the Elm programming language
    - https://elm-lang.org/

Today

- Garbage collection
  - Key idea
  - Mark and sweep
  - Stop and copy
  - Generational collection
  - Reference counting
  - Incremental collection, concurrent collection
  - Boehm collector

Runtime System

- Runtime system: all the stuff that the language implicitly assumes and that is not described in the program
  - Handling of POSIX signals
    - POSIX = Portable Operating System Interface
    - IEEE Computer Society standards for OS compatibility
  - Automated memory management (garbage collection)
  - Automated core management (work stealing)
  - Virtual machine execution (just-in-time compilation)
  - Class loading
  - …
- Also known as “language runtime” or just “runtime”

Automated Memory Management

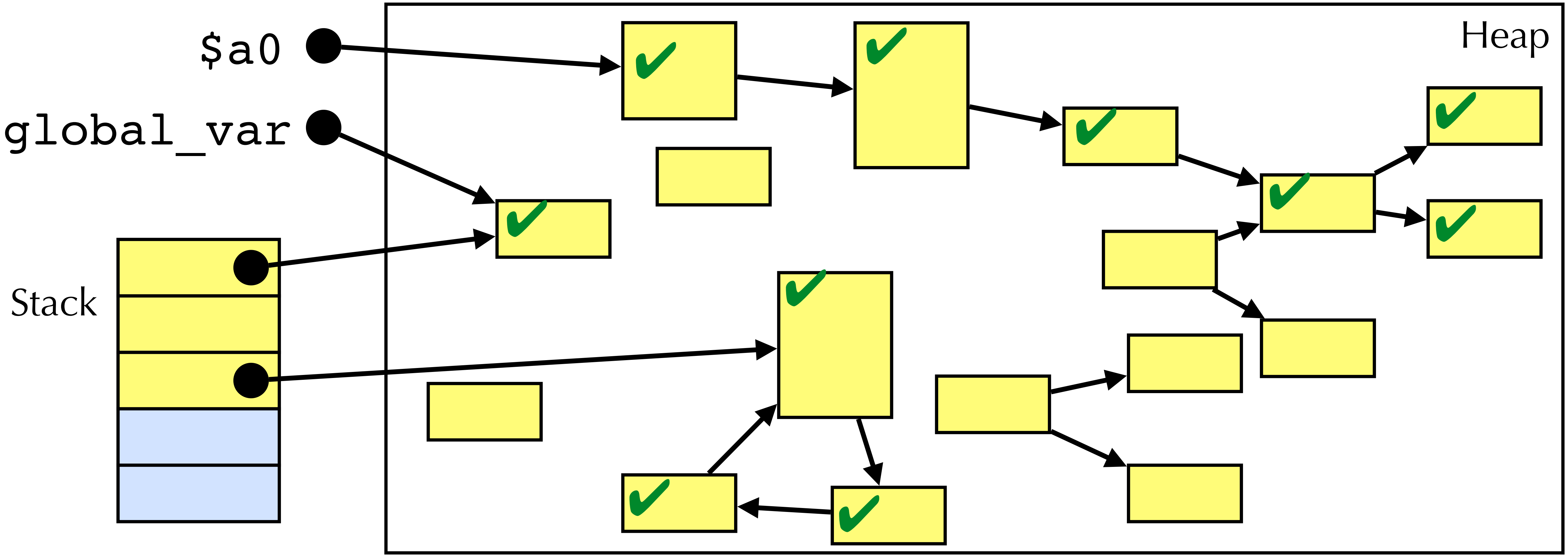
- Manual memory management: programmers explicitly call `malloc()` and `free()`
- Automatic memory management: runtime system looks after allocation and **garbage collection**
  - Garbage collection: free memory that is no longer in use

Garbage Collection

- Runtime frees heap memory that is no longer in use
- How do we determine what is no longer in use?
- Ideally: any piece of memory that will not be used in the future of the computation
- In practice: any piece of memory that is not **reachable**
  - Reachable = can be accessed through some chain of pointers starting from program variables
  - This is a subset of the memory that will not be used in the future

Garbage Collection: Basic Idea

- Start from stack, registers, & globals (roots) and follow pointers to determine which objects in heap are reachable
- Reclaim any object that isn't reachable



- Problem: How do we know which values are pointers and which are non-pointers (e.g., ints)?

Identifying pointers

- OCaml uses the low bit: 1 it's a scalar, 0 it's a pointer
  - Why the low bit? Why not the high bit?
- In Java, we put tag bits in the meta-data
- In C (e.g., Boehm collector), typically use heuristics
  - If value doesn't point into an allocated object, it's not a pointer

Mark and Sweep Collector

- Reserve a mark-bit for each object.
- Mark phase

```
For each root v:
    DFS(v)
```

```
function DFS(x):
    if x is a pointer into heap
        if record x is not marked
            mark x
            for each field f_i of record x
                DFS(x.f_i)
```

- Sweep phase

```
p ← first address in heap
while p < last address in heap
    if record p is marked
        unmark p
    else let f1 be the first field in p
        p.f1 ← freelist
        freelist ← p
    p ← p + (size of record p)
```

Explicit Stack

- DFS is recursive function
  - Stack frame for each invocation!
- Use explicit stack instead...

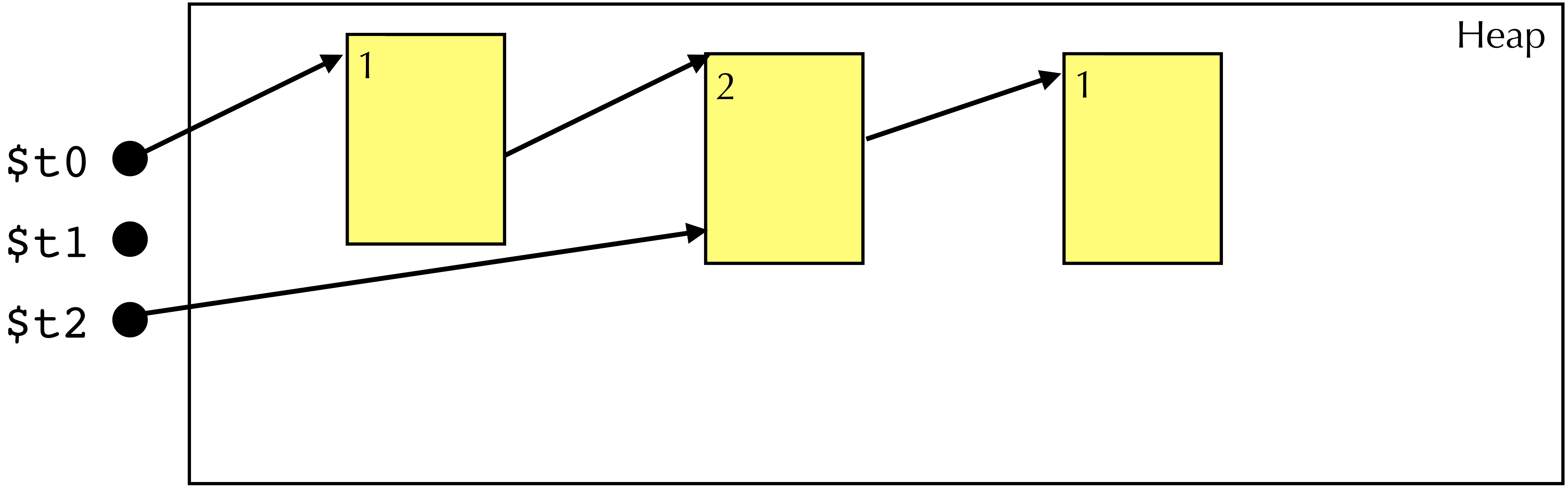
```
function DFS(x):
    if x is a pointer into heap and x not marked
        t ← 1
        stack[t] ← x
        while t > 0:
            x ← stack[t]; t ← t – 1
            for each field f_i of record x
                if x.f_i is a pointer into heap and x.f_i not marked:
                    mark x.f_i
                    t ← t + 1; stack[t] ← x.f_i
```

How Big Can the Stack Get?

- Worst case: stack can be as big as the heap!
- Trick: pointer reversal
  - Don't use explicit stack
  - Instead, when visiting $x.f_i$, use $x.f_i$ to store element of stack!
    - Specifically, store x in $x.f_i$
  - When stack is popped, restore original value of $x.f_i$
-

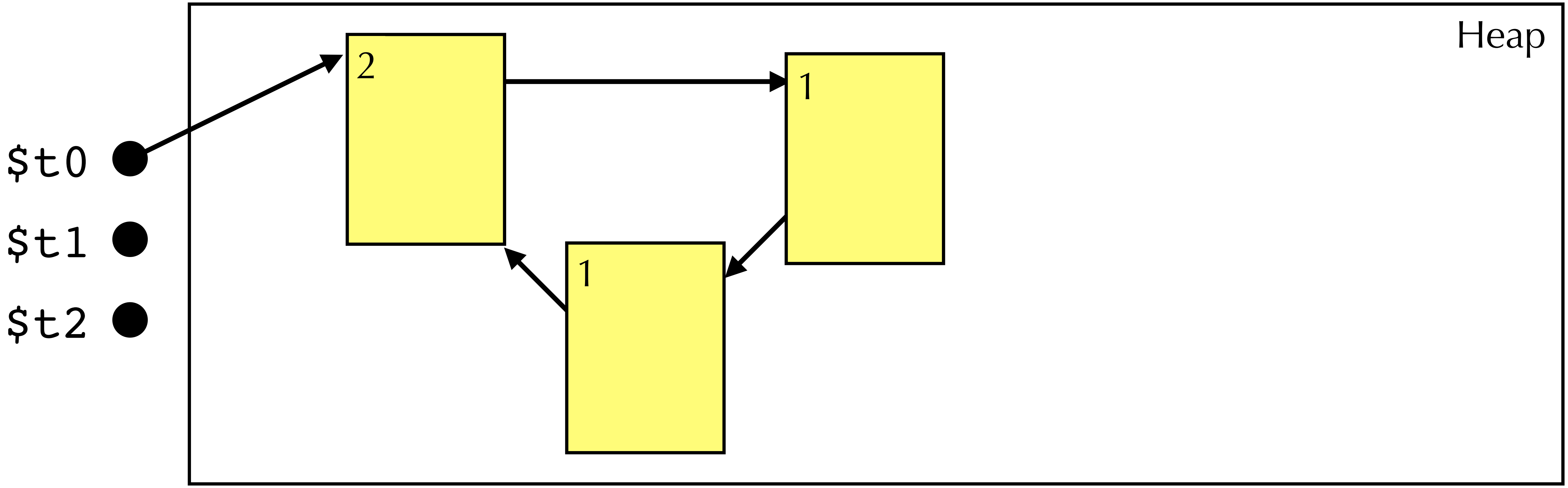
Reference Counting

- Key idea: track how many pointers point to each object
  - The **reference count** of the object, stored with object
  - Compiler modifies stores to increment/decrement reference counts
  - If reference count reaches 0, free object!



Reference Counting

- Any problems?
- What about cycles of garbage?
  - Require programmer to break cycles
  - Or do occasional mark-sweep collection

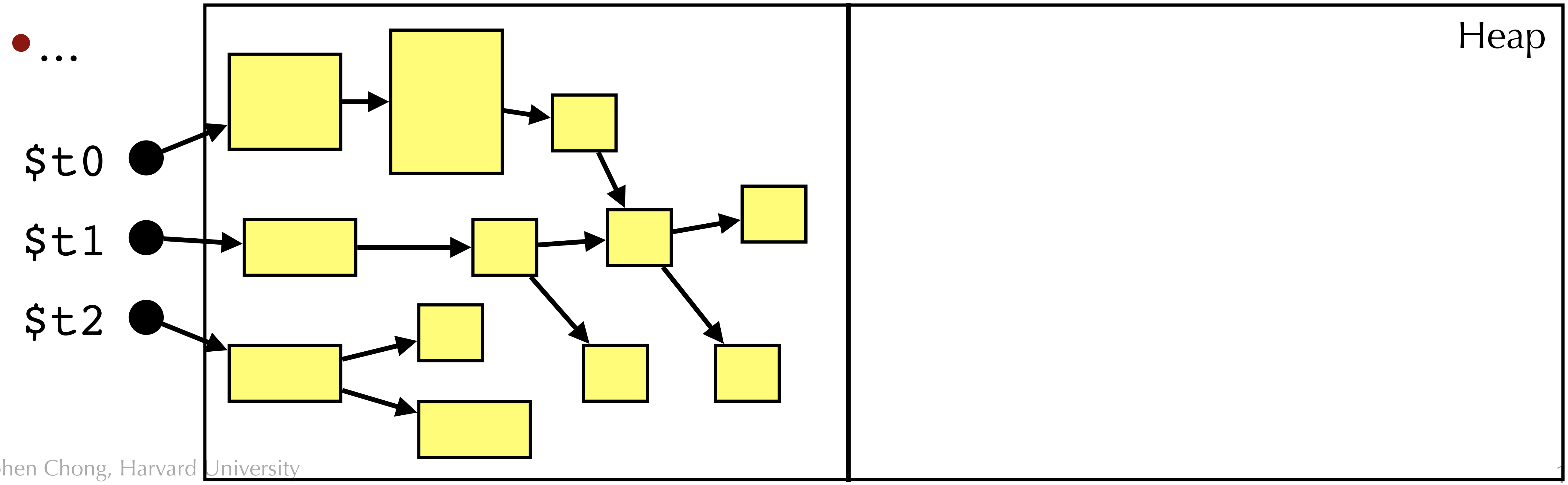


Costs of Reference Counting

- Whenever program wants to $x.f_i \leftarrow p$
- Must execute
  - $z \leftarrow x.f_i$
    $c \leftarrow z.count$
    $c \leftarrow c - 1$
    $z.count \leftarrow c$
    if c = 0 then call putOnFreelist
    $x.f_i \leftarrow p$
    $c \leftarrow p.count$
    $c \leftarrow c + 1$
    $p.count \leftarrow c$
- Dataflow analysis can reduce costs by aggregating updates
- But still expensive and not generally used

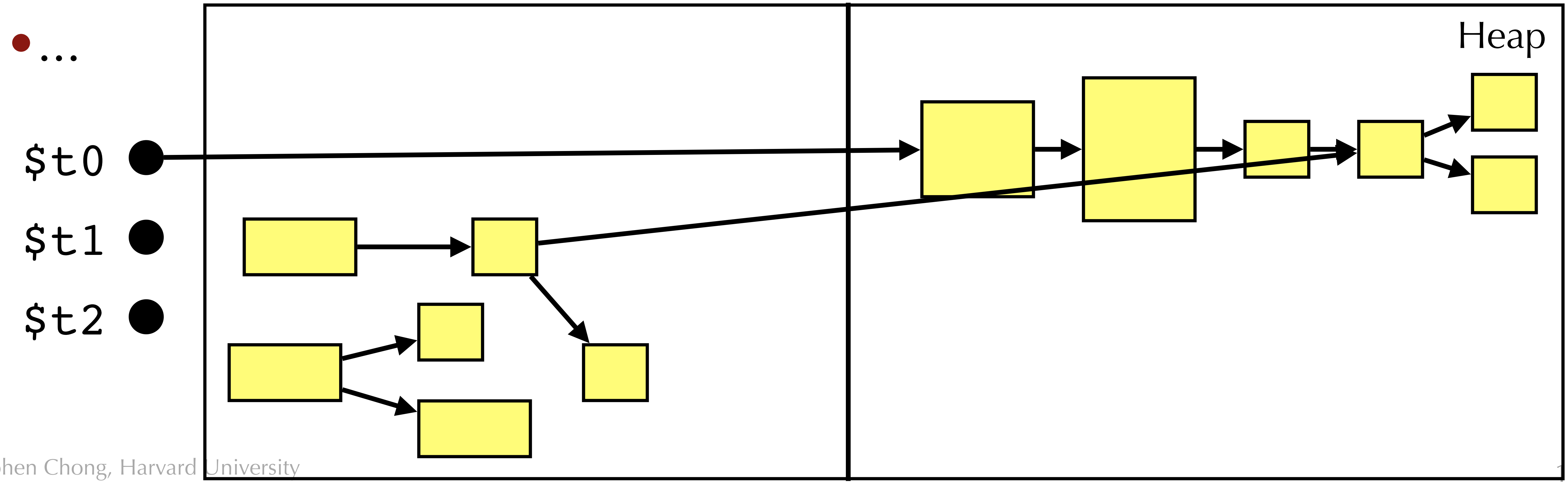
Stop and Copy Collector

- Split the heap into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.
- Can now reclaim all of the 1st piece!
- Allocate in 2nd piece until it fills up
- ...



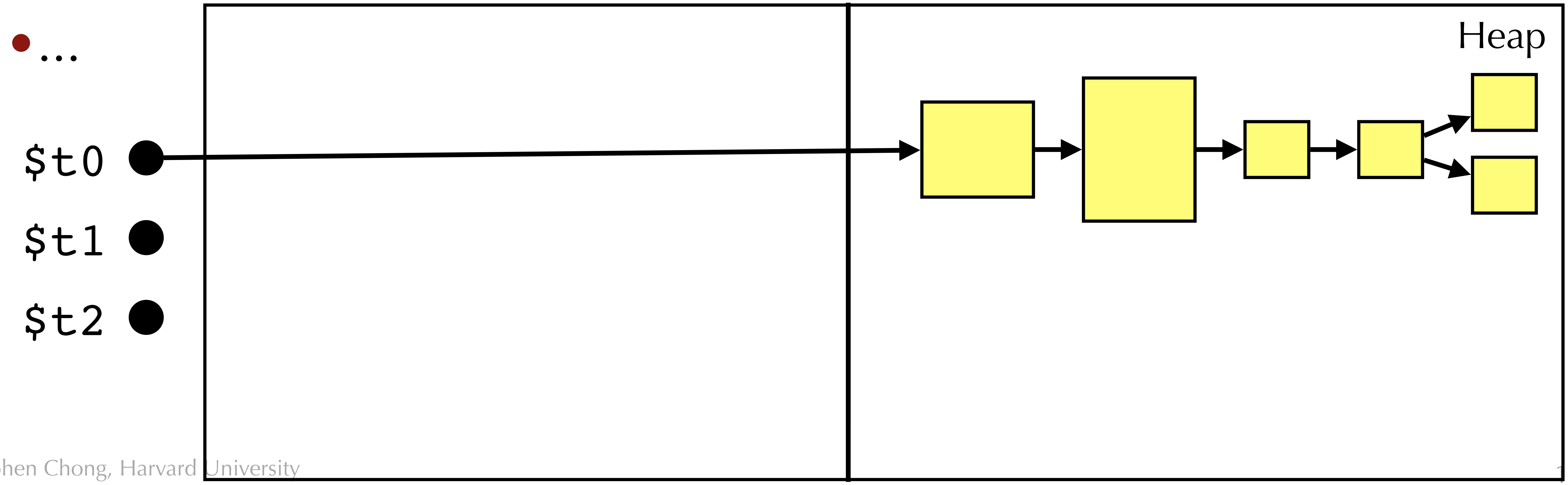
Stop and Copy Collector

- Split the heap into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.
- Can now reclaim all of the 1st piece!
- Allocate in 2nd piece until it fills up
- ...



Stop and Copy Collector

- Split the heap into two pieces.
- Allocate in 1st piece until it fills up.
- Copy the reachable data into the 2nd area, compressing out the holes corresponding to garbage objects.
- Can now reclaim all of the 1st piece!
- Allocate in 2nd piece until it fills up
- ...



Generational Collection

- In many programs, newly created objects are likely to die soon
- Conversely, objects that survive many collections will probably survive many more collections
- So: collector should concentrate effort on "young" data (where there is higher proportion of garbage)
- Key idea: Divide heap into **generations**
  - Allocate new objects into generation G_0
  - Collect G_0 frequently, G_1 less frequently, G_2 even less so, ...
  - If object survives 2-3 collections in G_i, copy it into G_{i+1}
- Roots now include pointers from older generations to younger ones
  - Relatively rare
  - But need mechanism to remember them

Incremental Collection
Concurrent Collection

- Collector will occasionally interrupt program for long periods of time for garbage collection
  - Problem for interaction or realtime programs!
- **Incremental collection** performs some work on garbage collection when the program requests it
- **Concurrent collection** performs garbage collection concurrently with program
- Can greatly reduce latency!

Reality

- Large objects (e.g., arrays) can be copied “virtually" without a physical copy.
- Some systems use mix of copying collection and mark/sweep with compaction.
- A real challenge is scaling to server-scale systems with terabytes of memory…
- Interactions with OS matter a lot: cheaper to do GC than to start paging…
- Java has a variety of GCs available with different tradeoffs
  - Default is generational collector that uses multiple threads when it runs
- OCaml uses a generational/incremental collector, invoked only in allocation

Conservative Collectors

- Work without help from the compiler.
  - e.g., legacy C/C++ code.
- Cannot accurately determine which values are pointers.
  - But can rule out some values (e.g., if they don't point into the data segment.)
- So they must conservatively treat anything that looks like a pointer as such.
- What happens if we have a value we aren't sure is a pointer or not?
  - Two bad things: leaks, can't move the object!

The Boehm Collector

- Based on mark/sweep.
  - Performs sweep lazily
- Organizes free lists as we saw earlier.
  - Different lists for different sized objects.
  - Relatively fast (single-threaded) allocation.
- Most of the cleverness is in finding roots:
  - global variables, stack, registers, etc.
- And determining values aren't pointers:
  - e.g., blacklisting (recording values that aren’t pointers but are in vicinity of heap)