

## Compiler Construction Lent Term 2015 Lecture 7 (of 16)

- In lecture demo of slang1 compiler
  - [http://www.cl.cam.ac.uk/teaching/1415/CompConstr/slang1\\_compile.tar.gz](http://www.cl.cam.ac.uk/teaching/1415/CompConstr/slang1_compile.tar.gz)
  - Jargon virtual machine
    - Uses static links
  - Lambda lifting
    - Slang.1 to Slang.1 transformation.
    - Does not always work. Why?
    - Static links in Jargon are not used lifted code
    - For tricky bits, see lambda\_lift.ml

Timothy G. Griffin  
[tgg22@cam.ac.uk](mailto:tgg22@cam.ac.uk)  
Computer Laboratory  
University of Cambridge

1

## Compiler Construction Lent Term 2015 Lectures 8 and 9 (of 16)

- **Assorted topics**
  - **Bootstrapping**
  - **Garbage collection**

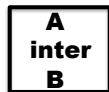
Timothy G. Griffin  
[tgg22@cam.ac.uk](mailto:tgg22@cam.ac.uk)  
Computer Laboratory  
University of Cambridge

2

## Bootstrapping. We need some notation . . .



An application called **app** written in language **A**

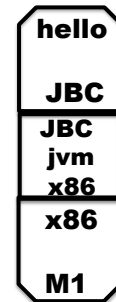
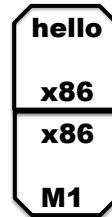


An interpreter or VM for language **A**  
Written in language **B**

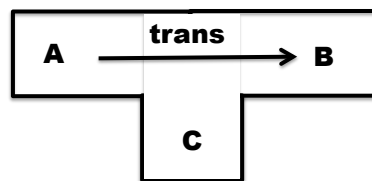


A machine called **mch** running language **A** natively.

### Simple Examples

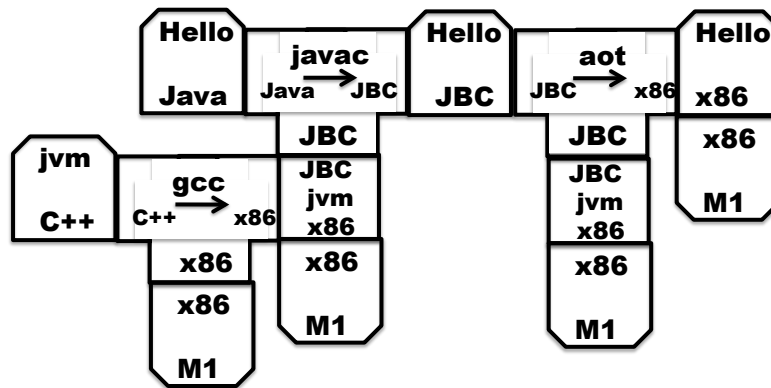


## Tombstones



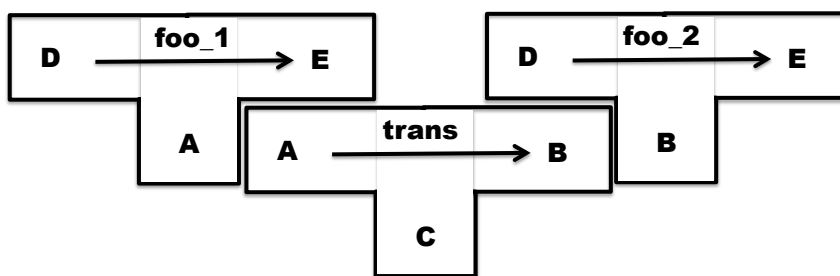
This is an application called **trans** that translates programs in language **A** into programs in language **B**, and it is written in language **C**.

## Ahead-of-time compilation



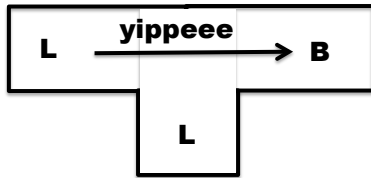
Thanks to David Greaves for the example.

## Of course translators can be translated

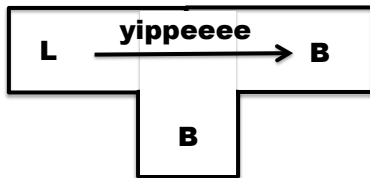


Translator **foo\_2** is produced as output from **trans** when given **foo\_1** as input.

## Our seemingly impossible task



We have just invented a really great new language **L** (in fact we claim that “**L** is far superior to C++”). To prove how great **L** is we write a compiler for **L** in **L** (of course!). This compiler produces machine code **B** for a widely used instruction set (say **B** = x86).



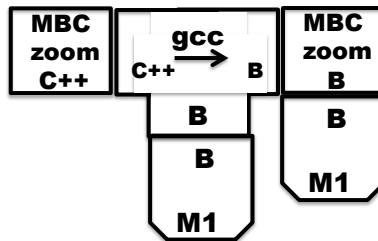
Furthermore, we want to compile our compiler so that it can run on a machine running **B**.

**How can we compile our compiler?**

There are many many ways we could go about this task. The following slides simply sketch out one plausible route to fame and fortune.

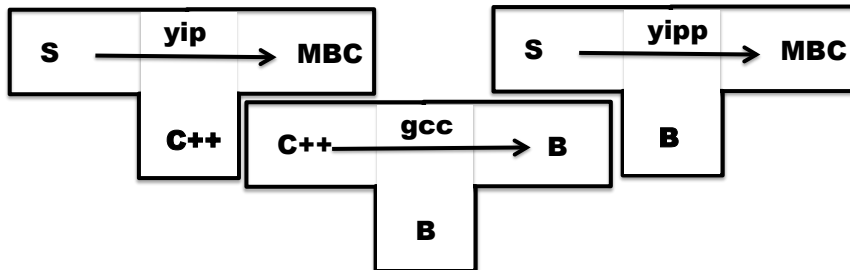
## Step 1 Write a small interpreter (VM) for a small language of byte codes

**MBC** = My Byte Codes



The **zoom** machine!

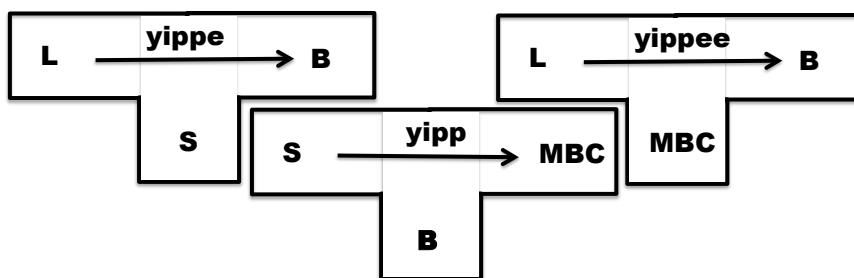
**Step 2**  
**Pick a small subset S of L and**  
**write a translator from S to MBC**



Write **yip** by hand. (It sure would be nice if we could hide the fact that this is written in C++.)

Translator **yipp** is produced as output from **gcc** when **yip** is given as input.

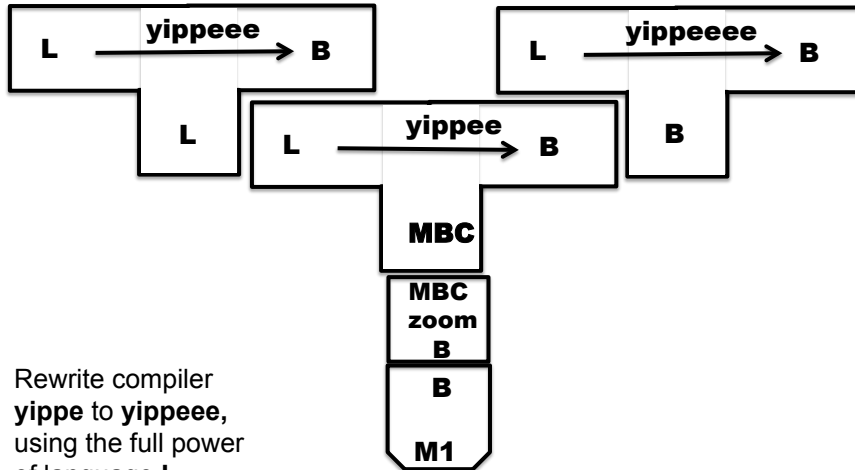
**Step 3**  
**Write a compiler for L in S**



Write a compiler **yippe** for the full language **L**, but written only in the sub-language **S**.

Compile **yippe** using **yipp** to produce **yippee**

### Step 4 Write a compiler for L in L

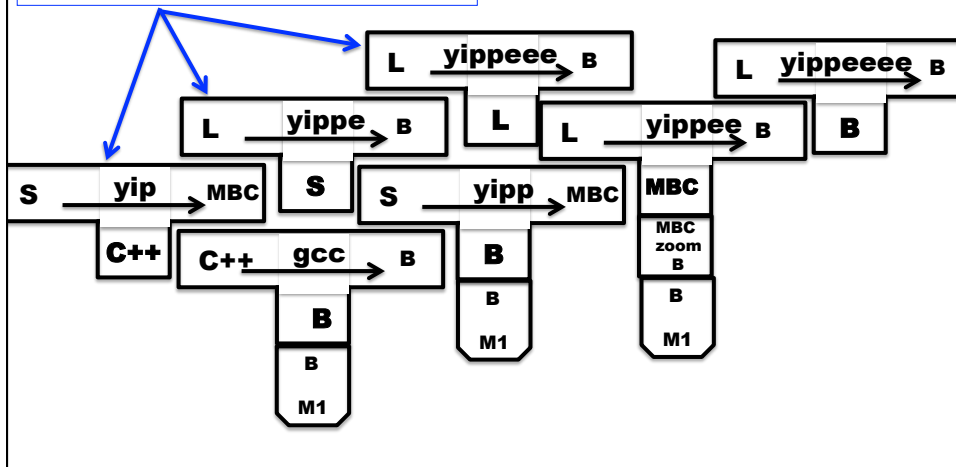


Rewrite compiler **yippe** to **yippeeee**, using the full power of language L.

Now compile this using **yippe** to obtain our goal!

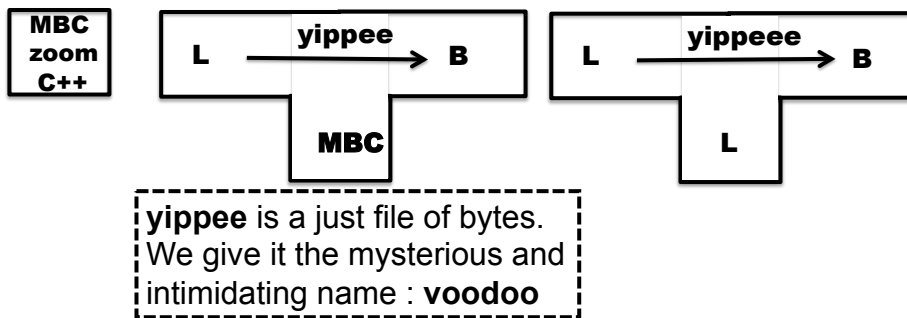
### Putting it all together

We wrote only these compilers and the MBC VM.



## Step 5 : Cover our tracks and leave the world mystified and amazed

Our L compiler download site contains only three components:



Our instructions:

1. Use **gcc** to compile the **zoom** interpreter
2. Use **zoom** to run **voodoo** with input **yippee** to output the compiler **yippeeee**

Shhhh! Don't tell anyone that we wrote the first compiler in C++

## New topic : Automating run-time memory management

- Managing the heap
- Garbage collection
  - Reference counting
  - Mark and sweep
  - Copy collection
  - Generational collection

**Read related chapter of Appel**

## Memory Management

- Modern programming languages allow programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Memory could easily be exhausted without some method of reclaiming and recycling the storage that will no longer be used.
  - Let programmer worry about it (use **malloc** and **free** in C...)
  - Automatic “garbage collection”

15

## Solutions

- Let programmer worry about it (use **malloc** and **free** in C...)
- Do nothing
- Automatic memory management (“garbage collection”)
  - Reference Counting
  - Mark and Sweep
  - Copy Collection
  - Generational Collection
  - ... there are many other GC techniques ...

In general, we must approximate since determining exactly what objects will never be used again is **not decidable**.

16



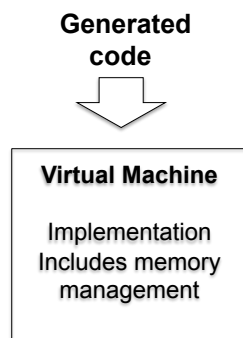
## Explicit Memory Management

- User library manages memory; programmer decides when and where to allocate and de-allocate
  - `void* malloc(long n)`
  - `void free(void *addr)`
  - Library calls OS for more pages when necessary
  - **Advantage:** Gives programmer a lot of control.
  - **Disadvantage:** people too clever and make mistakes. Getting it right can be costly. And don't we want to automate-away tedium?
  - **Advantage:** With these procedures we can implement memory management for "higher level" languages ;-)

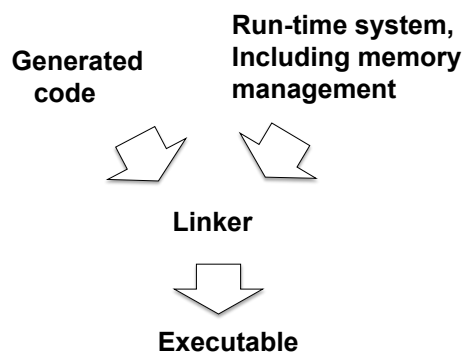
17

## Automatic Memory Management

### Targeting a VM



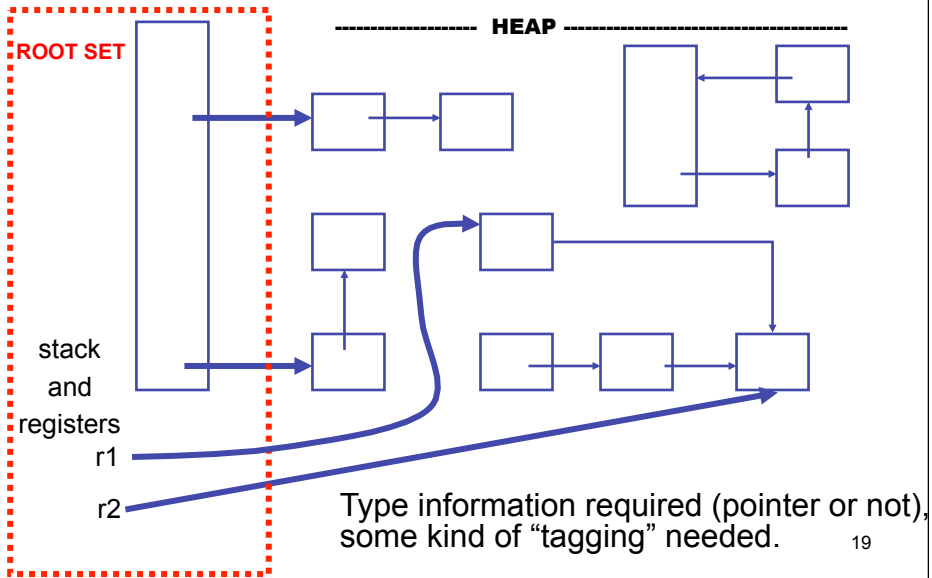
### Targeting a platform



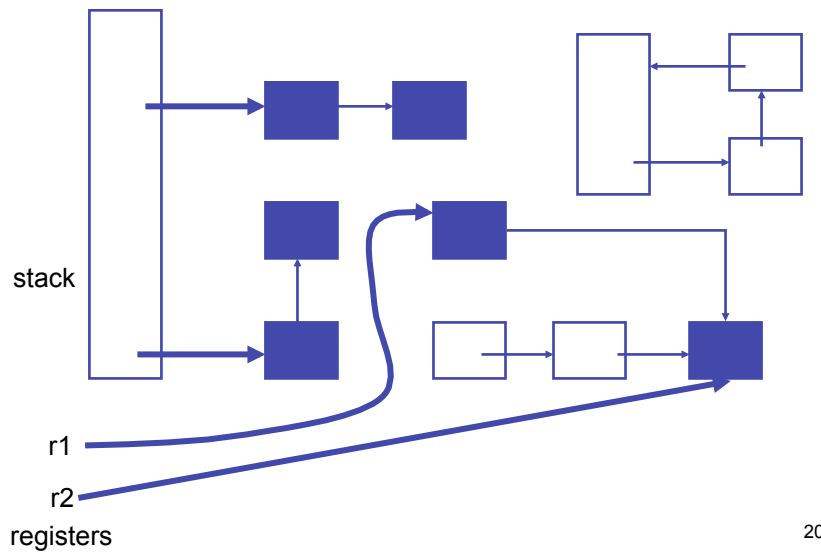
- **When to invoke collection?**
  - **When out of memory?**
  - **When to allocate more space?**
  - ...

18

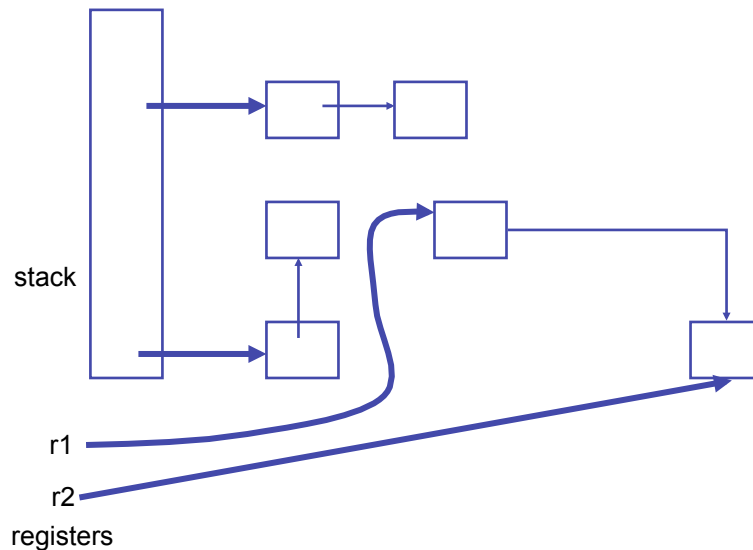
**Automation is based on an approximation : if data can be reached from a root set, then it is not “garbage”**



**... Identify Cells Reachable From Root Set...**



### ... reclaim unreachable cells



21

### But How? Two basic techniques, and many variations

- **Reference counting** : Keep a reference count with each object that represents the number of pointers to it. Is garbage when count is 0.
- **Tracing** : find all objects reachable from root set. Basically transitive close of pointer graph.

For a very interesting (non-examinable) treatment of this subject see

**A Unified Theory of Garbage Collection.**  
David F. Bacon, Perry Cheng, V.T. Rajan.  
OOPSLA 2004.

In that paper reference counting and tracing are presented as “dual” approaches, and other techniques are hybrids of the two.

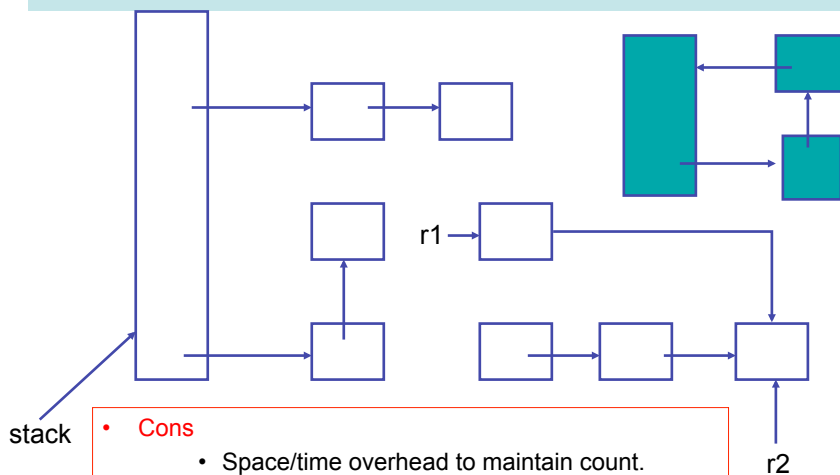
22

## Reference Counting, basic idea:

- Keep track of the number of pointers to each object (the **reference count**).
- When Object is created, set count to 1.
- Every time a new pointer to the object is created, increment the count.
- Every time an existing pointer to an object is destroyed, decrement the count
- When the reference count goes to 0, the object is unreachable garbage

23

## Reference counting can't detect cycles!



- **Cons**
  - Space/time overhead to maintain count.
  - Memory leakage when have cycles in data.
- **Pros**
  - Incremental (no long pauses to collect...)

24

## Mark and Sweep

- A two-phase algorithm
  - **Mark phase:** Depth first traversal of object graph from the roots to mark live data
  - **Sweep phase:** iterate over entire heap, adding the unmarked data back onto the free list

25

## Cost of Mark Sweep (somewhat crude)

- Cost of mark phase:
  - $O(R)$  where  $R$  is the # of reachable words
  - Assume cost is  $c_1 * R$  ( $c_1$  may be 10 instr' s)
- Cost of sweep phase:
  - $O(H)$  where  $H$  is the # of words in entire heap
  - Assume cost is  $c_2 * H$  ( $c_2$  may be 3 instr' s)
- Analysis
  - The “good” = each collection returns  $H - R$  words reclaimed
  - Amortized cost = time-collecting/amount-reclaimed
    - $((c_1 * R) + (c_2 * H)) / (H - R)$
    - If  $R$  is close to  $H$ , then each collection reclaims little space..
  - $R / H$  must be sufficiently small or GC cost is high.  
Could dynamically adjust. Say, if  $R / H$  is larger than .5, increase heap size

26

## Other Problems

- Depth-first search is usually implemented as a recursive algorithm
  - Uses stack space proportional to the longest path in the graph of reachable objects
    - one activation record/node in the path
    - activation records are big
  - If the heap is one long linked list, the stack space used in the algorithm will be greater than the heap size!!
  - What do we do? Pointer reversal [See Appel]
- Fragmentation

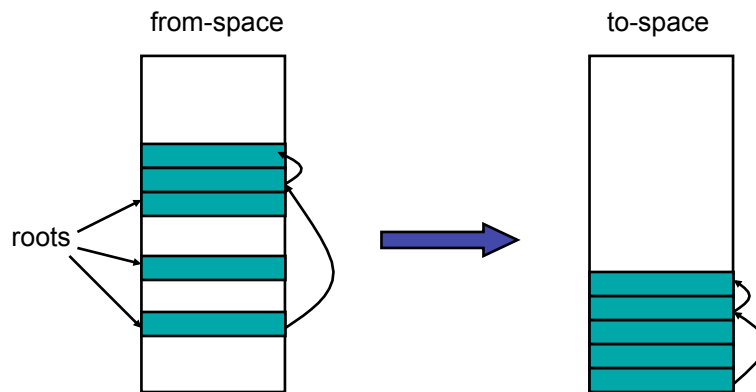
27

## Copying Collection

- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

28

## Copying Collection



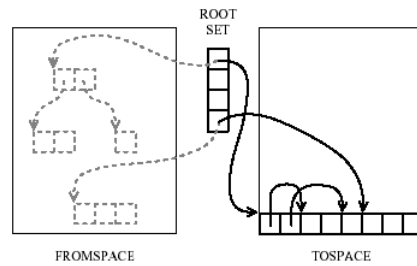
29

## Copying GC

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
- Cons
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps

30

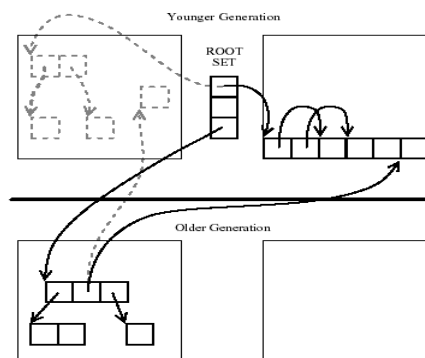
## OBSERVATION: for a copying garbage collector



- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It's inefficient that long-lived objects be copied over and over.

31

## IDEA: Generational garbage collection



**Segregate objects into multiple areas by age, and collect areas containing older objects less often than the younger ones.**

32



## Other issues...

- **When do we promote** objects from young generation to old generation
  - Usually after an object survives a collection, it will be promoted
- **Need to keep track of older objects pointing to newer ones!**
- **How big should the generations be?**
  - Appel says each should be exponentially larger than the last
- **When do we collect the old generation?**
  - After several **minor collections**, we do a **major collection**
- **Sometimes different GC algorithms are used for the new and older generations.**
  - **Why? Because they have different characteristics**
  - **Copying collection for the new**
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - **Mark-sweep for the old**

33