Administration

• Minijava project
  – Phase 6 (Register allocation)
    • due Wednesday April 4\textsuperscript{th}
    • There is a contest 😊
    • Link is on the course homepage
  – Overall
    • Due Wednesday April 11\textsuperscript{th} 11:59pm
    • A few more days to work out any issues
The Plan

- Dataflow analysis
- Optimization
- A Few final words on optimization
- Garbage collection
  - A few words on techniques
  - Integration with the compiler
- Other compiler-like things
- JIT
Back to inter-procedural optimization

• We said previously that we were doing intra-procedural optimization
• But we can do one simple form of inter-procedural optimization …
Inlining

- Replace a call to a method/function with the (slightly adjusted to deal with arguments) body of the method/function

- When is this a good idea?
- When is it not correct?
- When is it not an improvement?
Performance

• Run the compiler on BinarySearch.java
  – sim
  – simfinal - register allocation
  – osimfinal - constants, dead code, copy, peephole
  – xosimfinal - inlining, plus the above
The Plan

- Dataflow analysis
- Optimization
- A Few final words on optimization
  - Garbage collection
    - A few words on techniques
    - Integration with the compiler
- Other compiler-like things
- JIT
Why do languages add automatic garbage collection (GC) instead of manual memory management?
Garbage collection 101

• Why do languages add automatic garbage collection (GC) instead of manual memory management?

• Manual memory management:
  – is complex and error prone
  – alters program structure:
    • tends to destroy “tail call structure”
  – makes it more difficult to make simple abstractions (version explosion)
  – makes it hard to decide who is responsible for reclaiming “shared” data structures
Garbage collection 101

• What does an automatic garbage collector (GC) do?
Garbage collection 101

• What does an automatic garbage collector (GC) do?
  – It predicts the future!
  – Difference between truth and provability

• Garbage collectors must have
  – utility: gc must reclaim enough memory to help your program continue.
  – soundness: gc must not reclaim memory that may still be used.
  – efficiency: gc must work fast and not use too much memory!
We will look at some details of how GC is implemented and how it interfaces with the compiler.

Algorithms “inside” the GC
- intro (general ideas)
- mark and sweep collector
- copying collector
- reference counting
- generational

Interface between Compiler and GC
The general GC idea

- We want to reclaim any memory that is no longer dynamically live
- This is impossible!
  - Why?
- What is our solution (as always)?
  - Conservative approximation!
  - What approximation do we take this time?
Approximating dynamic liveness

- Any object (memory) that is live must be reachable from a well-defined set of “root” pointers.
- Non-reachable objects are dead and can be reclaimed.
- But ...
  - Some dead objects may be reachable, and won’t be reclaimed!
- How do we know that all live objects are reachable?
- How do we know the set of roots?
Mark and sweep GC

1. Mark Phase
   - Traverse object graph from roots
   - Mark all reachable objects

2. Sweep Phase
   - Scan memory for unmarked objects
   - Collect all unmarked objects onto a free list
Mark Phase

Uses depth-first search

for each root $r$

dfs($r$)

function dfs($p$)

if $p$ is a valid pointer into the heap

if the object at $*p$ is not marked

mark the object at $*p$

for each pointer field $f$ of the object at $*p$

dfs($f$)
Sweep Phase

Linear scan over the heap

\( p = \text{first address in the heap} \)

while \( p \neq \text{last address in the heap} \)

if *\( p \) is marked

    unmark *\( p \)

else

    add *\( p \) to the free list

    \( p.f = \text{freelist} \)

    \( \text{freelist} = p \)

\( p = p + \text{sizeof} \,* p \)
Cost of GC

- Let $H$ be the size of the heap
- Let $R$ be the amount of reachable memory in the heap
- We would like to estimate the average cost of memory reclamation per unit of memory allocated in terms of $H$ and $R$
Cost of Mark and Sweep

- Let $H$ be the size of the heap
- Let $R$ be the amount of reachable memory in the heap
- Let $c_1$ be the number of instructions (or memory references) for each Mark phase iteration (say 10)
- Let $c_2$ be the number of instructions (or memory references) for each Sweep phase iteration (say 5)

\[
\frac{c_1 \times R + c_2 \times H}{H - R}
\]
Problem with the Mark Phase

Uses recursive depth-first search, which uses lots of stack space just when we have run out of memory!

for each root r
  dfs(r)

function dfs(p)
  if p is a valid pointer into the heap
    if the object at *p is not marked
      mark the object at *p
      for each pointer field f of the object at *p
        dfs(f)
Copying GC

1. Divide memory into two equal sized pieces
   - from space and to space
2. Allocate in from space
3. When from space fills up
   - Copy every live object into to space, bringing them all together at one end
   - You have to adjust every pointer since every object moves
4. Swap from space and to space and go to step 2