Optimization

(a) This one is fine

(b) This one is not sound; the code on the left computes `<top of stack> - x - y`, but the code on the right computes `<top of stack> - (x - y)`. Subtraction is not associative.

(c) This one is not sound; code e.g. in another thread could try to get the value of `foo.x` between the first and second `putstatic` instructions. (Ya, this one is kind of unreasonable...)

(d) Not sound. This is true in general of patterns that just remove instructions, because you could have someething like

```
goto label:
...
label:
ineg
ineg
.end method
```

It's the same issue I mentioned in class with removing `nop`. The typical way to deal with this is to have patterns insert `nop` instead of removing everything, then have one pattern (or a set of patterns) to handle removing `nop`.

(e) This one is fine

Garbage collection

Mark and sweep

1. (i) Two phases; the marking phase does a depth-first search starting from the program roots (variables accessible from the program; in our case we just considered variables on the stack) and marks every reachable record. The sweeping phase basically iterates over the heap and adds unmarked records to a free list (and unmarks marked records).

   (ii) We didn't really see a "primary strength". Actually plain mark-and-sweep is kind of crazy; it was one of the first garbage collection algorithms and at the time sweeping over the heap wasn't unreasonable because heaps tended to be small, which is not so much the case today. Heaps are still relatively small in e.g. embedded systems, or sometimes you use mark-and-sweep for portions of a heap, e.g. a generation in generational collection. An okay answer would be "it's conceptually simple and easy to implement". But ya.

   (iii) The main weakness we saw was fragmentation; after collection, free records are scattered all over memory, so that potentially you could have enough memory to fulfill an allocation but not enough contiguous memory. This is bad. You could deal with this by periodically running some sort of defragmentation algorithm but we didn't see that in class. Another big weakness is that collection time is linear in the size of the heap.

Reference counting

(i) The runtime maintains a reference count for each record; for every assignment the reference count is updated, and if the reference count of a record drops to 0 it's dead, so you can reclaim it right away.

(ii) It catches dead records right away, so it's not as stop-the-world as the other algorithms we saw, although potentially reclaiming a dead record can cause other records to become dead and you could be reclaiming a huge garbage chunk at once. This is kind of lame because you wouldn't expect e.g. an assignment to take unbounded time. You can deal with this by not reclaiming records right away; reclaim the one that just died, and put the others on some list to be reclaimed later, e.g. on the next allocation.

(iii) The main one we saw is that it doesn't deal with cycles of dead records. There are many ways to deal with this. One is to periodically run e.g. mark and sweep to catch cycles of dead records and free them. Many languages also have a concept of "weak" references, which are references that don't count towards the reference count; C# has `System.WeakReference`, Java has `java.lang.ref.WeakReference` (and collections based on it, e.g. `WeakHashMap`), python has the `weakref` module, etc. Clever use of these can avoid reference cycles. Consider e.g. a doubly linked list; if you drop all references to a doubly linked list, each element still has a nonzero reference count. But if the backwards pointers were weak, this issue disappears.

We also saw the overhead of reference counting tends to be too high in practice.

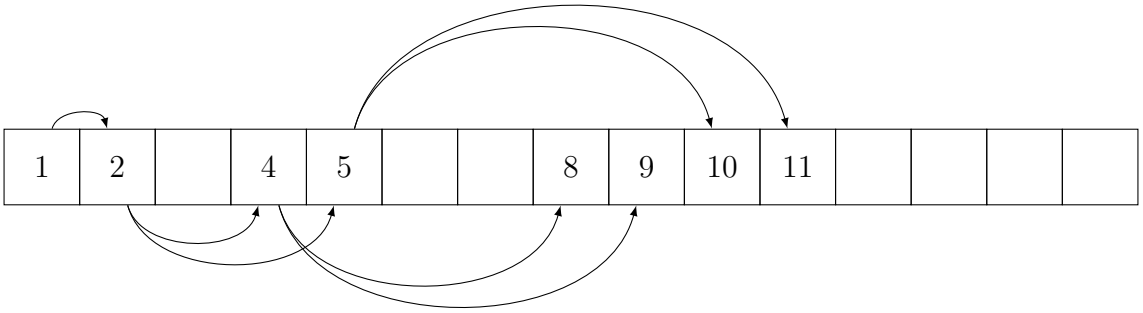
Stop and copy

(i) The heap is divided in two contiguous halves. One half ("from-space") is used to allocate memory, the other ("to-space") remains empty until collection. When a collection is triggered, reachable records in from-space are forwarded to to-space. Two pointers into to-space are kept; `next`, the next available address, and `scan`, which we'll see. Both are initially pointing to the start of to-space. The main operation is forwarding a record; if it's from-space, it's copied to to-space at `next` and a forwarding pointer (also called a broken heart, or a tombstone) is left at its old address pointing to its new address; later, if other pointers are pointing to it, we can check for a forwarding pointer to see where it's been copied to. You start by forwarding the roots. Then you iterate from `scan` to `next`, forwarding fields as you go. You're done when `scan == next`. This is a pretty hairy description; see next question for an example.

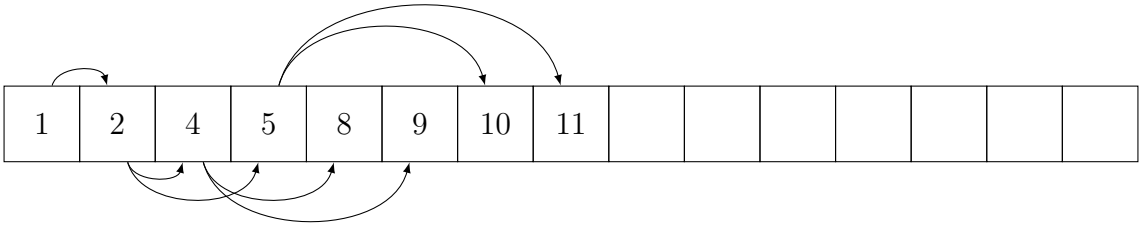
(ii) No fragmentation! Also, collection is linear only in the number of reachable of records, because there's no sweeping phase.

(iii) You only get to use half the heap. Also you get cache locality issues because objects that point to each other will not be next to each other in to-space because we do a breadth-first and not a depth-first search.

2. (a) After mark and sweep (using a blank square to mean something was reclaimed):

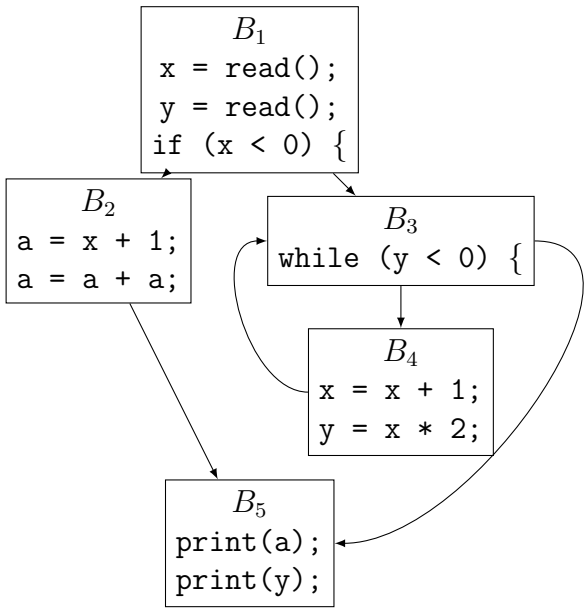


(b) After stop and copy. I'm not drawing out the intermediate steps, but note the order of the records, resulting from a breadth-first, not a depth-first search. A depth-first search would give an order of 1 2 4 8 9 5 10 11. This example is a little contrived because the order of the records doesn't change, just because of the way the heap is constructed. In general it's not that trivial.



3. I mentioned this in class. You'd prefer stop and copy to mark and sweep before 8GB is pretty big and there isn't very much reachable memory at any point so stop-and-copy should be fast.

Dataflow analysis



1. (a) Let's just start by giving each block a name:

Recall the dataflow equation for liveness was

$$in(S) = (out(S) \setminus defs(S)) \cup uses(S) = (\bigcup_{x \in succ(S)} in(x) \setminus defs(S)) \cup uses(S)$$

We didn't really see an example like this in class, where the nodes in the CFG were basic blocks instead of single statements, but it's not difficult; basically for a block B, $in(B) = in(S)$ where S is the first statement in the block, and $succ(B) = succ(U)$, where U is the last statement in the block. So we can get a dataflow equation for B_1, where S is `x = read()`, T is `y = read()`, U is `if (x < 0)`, just by repeatedly applying the dataflow equation:

$$in(B_1) = in(S) = in(T) \setminus \{x\} = in(U) \setminus \{y\} \setminus \{x\} = (\bigcup_{X \in succ(U)} in(X) \cup \{x\}) \setminus \{x\} \setminus \{y\}$$

so that $in(B_1) = \bigcup_{X \in succ(B_1)} in(X) \setminus \{x, y\}$. We can do the same for the other blocks. A complete solution might look like this:

Block	Successors	$\perp$	$f(\perp)$	$f^2(\perp)$	$f^3(\perp)$
B_1	B_2, B_3	$\varnothing$	$\{a\}$	$\{a\}$	$\{a\}$
B_2	B_5	$\varnothing$	$\{x, y\}$	$\{x, y\}$	$\{x, y\}$
B_3	B_4, B_5	$\varnothing$	$\{a, y\}$	$\{a, x, y\}$	$\{a, x, y\}$
B_4	B_3	$\varnothing$	$\{x, a\}$	$\{x, a\}$	$\{x, a\}$
B_5	exit	$\varnothing$	$\{a, y\}$	$\{a, y\}$	$\{a, y\}$

The order in which you visit consider nodes is not important in terms of the solution, but it can affect how many iterations you need. I used B_5, B_2, B_3, B_4, B_1.

Now that we know live-in and live-out sets for every basic block, we can go and propagate the information through each block and get information for each statement. I won't do this but it's straightforward.

(b) (There's a typo in the question; obviously there's no `b` in the program). I didn't really think this question through. In general you can (a) construct the interference graph within a basic block and look for a minimal coloring and (b) use the live-out set of each basic block to avoid spilling registers containing variables that aren't live. I don't think either of these is useful in this case.

2. (i) The information is flowing from the occurrence of the expression backwards; if you see an expression, it's very busy before the occurrence, not after.

(ii) This is a must (intersection, meet) analysis. In the example, `y+z` is very busy because it's computed in both branches; if we merged using union, `2*w` would also be very busy, which it's not.

(iii) If $S =$ `x = y + z;`, then $in(S) = (out(S) \setminus kill(S)) \cup \{y + z\}$ where $kill(S)$ is the set of expressions involving `y`. The set-minus happens before the union because e.g. consider `y = y + z; y + z` is still very busy before this, because `y` is assigned to after it's computed.