Using type information in cyclic reference counting (single reference count)

- Compiler finds possible cycles between types at compile-time
    - This is meant for languages without subtyping. It would probably be much harder to have guarantees about which types refer to what in a language like Java
    - At least one paper does do a bit of that for C# (Compiler Optimizations for Nondeferred Reference-Counting Garbage Collection, 2006), but the improvement from their static acyclic object RC update specialization was negligible. Their runtime specialization did, however, produce great improvements for some programs
- Contribution: reducing the number of objects that need to be scanned
    - As far as I can tell, in previous papers, when you have a bunch of potential cyclic roots, you have to scan every single object reachable from those

roots (save types known to be completely uninvolved in cycles)

- They do this to avoid rescanning cycles (see)
- But scanning the subgraph of every potential cyclic root isn't necessary if you know the direction in which references will go

For objects with cyclic types, need two extra bits of information in addition to reference count:

- 1 bit to tell whether or not the object is already added to the list of potential cyclic roots
- 1 bit to tell whether or not an object has been visited or not in the mark-scan phases
- Possibly an extra bit to tell if an object is a "critical link," as defined in Cyclic reference counting (Lins, 2008)

For objects whose types are known to not be part of any cycles, everything is the same as standard reference counting.

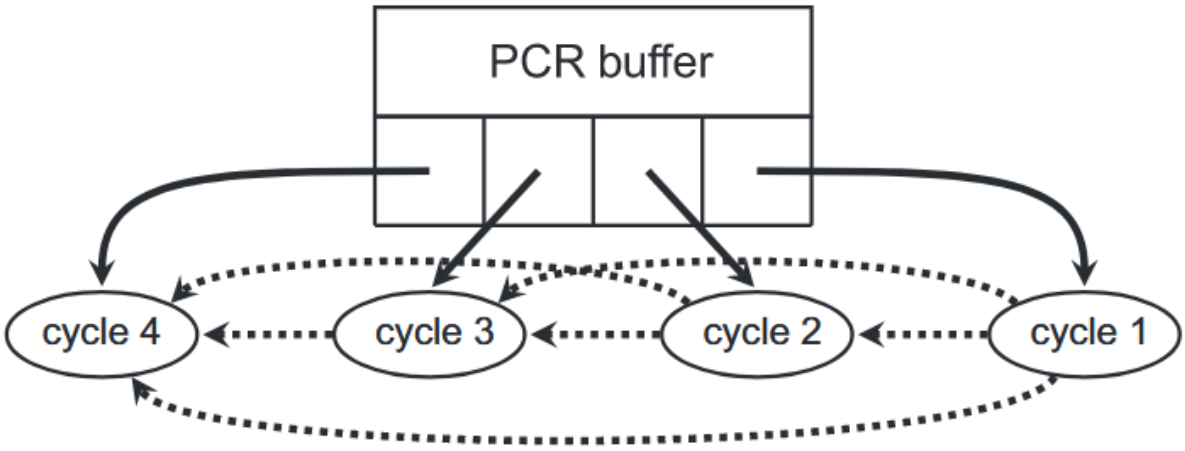
Deleting references to possible member of cycles

Like Lins' lazy mark-scan paper, we can maintain a list of objects that are to be deleted. Every time you remove a reference to an object, if the object's refcount is now 0, it gets freed. Otherwise, if the object might be part of a cycle (based on its type), add it to a list of possible cyclic roots (PCRs). This list will be sorted based on the types of the objects in it (elaborated below).

O(n^2) subgraph scanning issue

In a setup like the one below, where cycles 4, 3, 2, and 1 have been marked as possibly being garbage, you'd scan them in the order 4, 3, 2, 1. But suppose no one's referring to them and they all really are garbage. You'd scan cycle 4, think that it's alive because cycle 3 is referring to it. Then you'd scan cycle 3 and think that it's alive because cycle 2 is referring to it. Same for cycle 2. Finally, when you get to cycle 1, you'd see that it's dead and free that. But now, you need to scan

cycles 4, 3, and 2 again. So you can't maintain a simple stack of objects that are possibly garbage.



To avoid this, the objects will be stored in the list of PCRs so they're sorted according to their type. At compile-time, strongly connected components will be found in the graph of types. Each SCC will be assigned a number according to high in the tree it appears. For example, SCCs that no one refers to would be assigned 0, while SCCs that don't refer to anyone would be assigned the highest number, 9999 or something. SCCs that are unrelated to each other could be assigned the same number, it doesn't really matter. This means that cycle 1 would always be scanned first, then cycle 2, then 3, then 4.

This would be my main contribution. As far as I can tell, previous papers have instead chosen to scan not only cycles 4, 3, 2, and 1, but also all the objects between them. This might be a bit wasteful, depending on how many such objects there are.

Actually collecting garbage

This will work pretty much the same way as the algorithm in Lins' lazy mark-scan paper, except that references known to be acyclic won't be followed. Other improvements have been made to that, but I don't think there's any point in me implementing those too.

One minor change to avoid the quadratic time issue mentioned above: objects in the PCR list are now sorted by which SCC they were in, and consecutive objects

that were in the same SCC will be scanned together, instead of one after the other. This is to avoid an edge case where the quadratic time issue could occur.

- Note: this would hurt how incremental the algorithm can be. You might be forced to scan a thousand deleted PCRs at once just because they're all of the same type or something. You couldn't do a handful of them and then continue program execution.
- Possible alternative: whenever an object's refcount is decreased, even if it's already in the list of PCRs, it's moved up to be processed before the other PCRs that are in the same SCC. This won't solve the quadratic time issue entirely, but might ameliorate it. The "might" needs to be backed by profiling and stuff, though, and that seems like a lot for a semester.

Detecting cycles between types

The compiler will only care about cycles between types if one of the fields involved is mutable.

Somewhat contrived example

Suppose you are working on creating a compiler for a programming language and have the following types:

```
data Context {
  name: str
  mut files: FileList
}

data FileList
  | Nil
  | Cons {
    ctx: Context
    head: File
    tail: FileList
  }

data File {
```

```
  mut exprs: ExprList
}

data ExprList
  | Nil
  | Cons {
    head: Expr
    tail: ExprList
  }

data Expr {
  file: File
  // other stuff here
}
```

Some functions on those types:

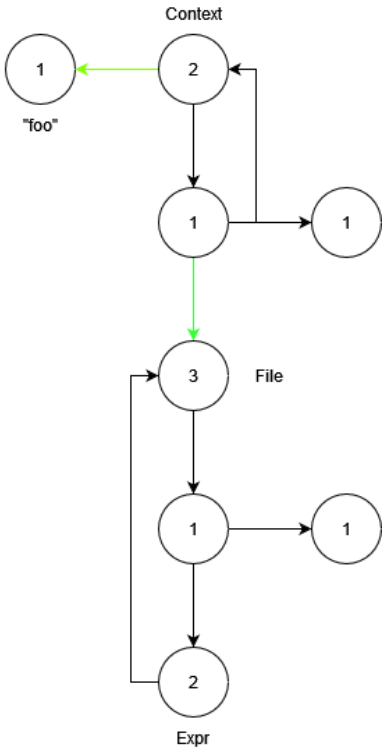
```
fn addFile(ctx: Context, file: File) {
  ctx.files = FileList::Cons { ctx: ctx, head: file, tail: ctx.files }
}
fn addExpr(file: File, expr: Expr) {
  file.exprs = ExprList::Cons { head: expr, tail: file.exprs }
}
```

Creating objects:

```
let ctx = Context { name: "foo", files: FileList::Nil }
let file = File { exprs: ExprList::Nil }
addFile(ctx, file)

let expr = Expr { file: file }
addExpr(file, expr)
```

After running that code, here's what the graph of references looks like (draw.io):



Imagine that the variable `ctx` is assigned to a different value, meaning that the `Context` object above has its refcount decremented to 1 and is added to the list of possible cyclic roots. After that, imagine that the variables referring to the `File` and the `Expr` are reassigned or go out of scope. The `File` object has refcount 2 and is added to the list of possible cyclic roots, and the `Expr` object has refcount 1 and is also added to the list.

At this point, every object in the diagram above is garbage. They will stay alive until the next mark-scan operation because the reference cycles are keeping their refcounts positive.

At some point, the lazy mark-scan algorithm will be invoked. It will go over the list of possible cyclic roots, look for garbage, and free it.

Note: the green edges in the diagram above are references that are known not to introduce any cycles because of the types of the objects involved. Therefore, when doing mark-scan, they won't be followed. There are other references that don't cause cycles in there, but they're not known to be acyclic at compile-time.

What other implementations do

If the list of possible cyclic roots had been just a stack/linked list, then it would look like `<Expr object> :: <File object> :: <Context object> :: nil`. This would mean that we'd first check if the `Expr` object is garbage and think that it isn't even though it will have to be freed later, when the `Context` object is freed. Then we'd check if the `File` object is garbage, and again think that it isn't. Only when we get to the `Context` object would we realize that the `Context` object is garbage. But the `Expr` and `File` objects would remain in the list of possible cyclic roots until the next time that mark-scan is invoked (unless you repeatedly traverse the list of possible cyclic roots until there are no changes, which introduces quadratic complexity, as mentioned above).

To avoid this, current implementations do two things:

1. Do the mark-scan thing starting from all the possible cyclic roots at once. That is, they wouldn't process the `Expr` object first, then the `File` object, and so on. They would process all the objects at once. As far as I can tell, this means you can't incrementally process potential cyclic roots, but I couldn't find papers that talked about this being a problem. Maybe I'm wrong, or maybe it's not a big deal.
2. They also have to trace (more or less) every object that can be referred to by each cyclic root. This is a bigger problem. It means unnecessarily following the green edges in the diagram above. In this particular example, that's fine, but it won't be in other scenarios. Suppose the `Context` object has a reference removed and is therefore added to the list of possible cyclic roots. However, the `File` and `Expr` objects are still referred to by local variables and therefore do not need to be scanned. However, while scanning the `Context` object, the `File` and `Expr` objects would also be scanned.

What my implementation would do

Instead, we'll make sure the list is sorted according to the strongly-connected component (SCC) that each object appeared in. At compile-time, the compiler would take the graph of types and make a tree of SCCs from it. Then, each SCC can be assigned a number based on its level in the tree. Here, `Context` and `FileList` would be at level 0, while `File`, `Expr`, and `ExprList` would be at level 1.

If we sort the list of possible cyclic roots based on the SCC that each object was in, it would look like `[<Context object>, <File object>, <Expr object>]` (the last two could be

swapped, since they both appear in the same SCC). This time, the `Context` object will be checked first and found to be garbage, after which the `File` and `Expr` objects will also be found to be garbage.

Problems

- I'm not doing generics here, but in really large programs, the number of types would be enormous. I assume compilation time would suffer?
    - However, a couple of other papers ("Cyclic reference counting by typed reference fields" and "Compiler Optimizations for Nondeferred Reference-Counting Garbage Collection") were fine with finding acyclic types at runtime, so it's probably not that bad
- Most of the papers I'm working off of are really old (2012 and before). It's possible people have already tried this idea in the meantime and rejected it or found improvements to it.
    - The latest paper I could find that mentioned using types for collecting cycles was from 2012 (Cyclic reference counting by typed reference fields). It claimed that at the time, there wasn't much research that took advantage of type information when doing cyclic reference counting
    - It's possible that this is because most research into collecting cycles is focused on object-oriented languages, where, thanks to subtyping, it's hard to have a ton of guarantees at compile time. My idea, on the other hand, is more for FP languages, which might only be getting attention from academics rather than researchers in the software industry?

Ways to speed up sorting

- Instead of having all PCRs in a single linked list sorted by SCC, create a list of lists of PCRs, where the inner lists of PCRs only contain PCRs in the same SCC. That way, when adding a new PCR, you don't have to go through every single PCR that came before to get to the right spot.

Possible plan

- 1 month for implementation

- 2 weeks for testing performance
    - Make a program that creates a bunch of objects
        - Compile it once normally and profile that
        - Compile it without double reference counts (assume that references to acyclic types are always acyclic but assume that any other references can be cyclic) and profile that
        - See if improvement is worth caring about
- 2 weeks of buffer
- 2 weeks to finish off paper

Possible additions

- Initially, I'd thought about having two references counts (one counting definitely acyclic references, one counting possibly cycle-causing references). I removed that idea from this document because I wasn't sure if the benefits outweighed the added memory usage, but I can try implementing that if I have time.
- In Cyclic reference counting (2008), Lins suggests checking for a cycle every time an assignment is made in order to reduce the number of times you need to scan for possible garbage. Since we have information about types and mutability of each field, we could reduce the amount of unnecessary scanning done.
    - Summary of that paper: it defines a "critical link" as the object that closed the cycle. Every cycle has one critical link marked (shouldn't hurt to mark more than one, though). When this critical link has its refcount decremented, you should scan for cycles, possibly find a new critical link, but otherwise, you don't need to scan for cycles
    - However, checking for a cycle on every assignment that possibly introduces a cycle is not great. It might be possible to color all possible critical links yellow no matter what, then add them to a list. Cycle checking could be deferred, maybe done concurrently.

    - I couldn't find anything in the paper accounting for small cycles being broken out of larger cycles. You'd have to mark a new critical link inside the small cycle every time that happens

Future stuff

- Look into that reference lifetime/subsumption thing. That could avoid a lot of unnecessary updates to the reference count. With type information, it might be possible to avoid even more updates in some specific cases
- Closures
    - Could probably add something like Rust's `dyn` and then have the ability to have a bound saying "this object can be of any type implementing this trait *if* it doesn't have any direct or indirect references to me"
- Coalescing
    - Implementing this at runtime seems hard to do
    - Maybe I could have the compiler generate instructions to decrement and increment each reference only at the start and end of each function/code block instead?
        - You'd also have to increment before function calls