Making lazy mark scan a bit faster by avoiding scanning some objects based on their type

Yash Thakur

CMSC499

12/13/2024

Contents

1. Introduction

One major problem with reference counting is the fact that it cannot free objects that are involved in reference cycles. Lazy mark scan is a cyclic reference counting algorithm that aims to fix this issue, but it requires traversing all objects reachable from any potential cyclic roots [1]. This write-up describes a way to reduce the number of objects traversed by applying information about types known at compile-time. My project involved creating a language Fred and implementing a runtime for it that takes advantage of this optimization.

2. Background

Lazy mark scan is a lazy version of an algorithm called local mark scan.

With local mark scan, every time an object's reference count is decremented, if its reference count does not hit 0, all of the objects reachable from that object are scanned recursively, and if it turns out that any of these objects were part of a cycle and are now no longer reachable, they will be freed [2]. This technique is referred to as trial deletion, because you basically pretend those objects were deleted, then see if they still would've had references coming in from outside. Note: I'm not sure whether trial deletion always means that you're doing local mark scan at the lowest level of your algorithm, or if there are other completely different algorithms out there that fall under the term trial deletion.

Since this process is expensive, lazy mark scan merely adds each object to a list of **potential cyclic roots** (**PCRs**). Every once in a while, this list of PCRs is traversed and mark scan is performed on all of these PCRs at once. Note that each PCR is not simply scanned individually, in sequence, because this would essentially be the same as local mark scan. Rather, each phase of the mark scan algorithm is sequentially performed on all PCRs before moving on to the next phase.

3. Prior work

However, lazy mark scan still requires scanning a bunch of objects. In a strongly-typed language, some guarantees can often be made about whether or not objects of one type can ever form cycles with objects of another type, and this can let us do less scanning.

For example, [3] takes advantage of type information available at runtime on the JVM. It doesn't add objects to the list of PCRs if their type makes it impossible for them to ever be part of cycles. [4] does the same, but as a compiler optimization.

[5] tries to take this a step further. During the mark scan process, they considered not scanning objects known to be acyclic based on their type. However, they note that this cannot be done naively, as I will discuss in Section 4.2. Instead, they restrict themselves to not scanning known acyclic objects only if such objects do not have any references to cycles. This project is focused on removing this restriction.

4. Algorithm

4.1. Avoiding scanning children based on type

As mentioned before, we can look at the types of objects to determine whether they will ever form cycles with other objects. Fewer guarantees can be made if the type system in question includes subtyping or something, but this project only looks at a very simple language, with no subtyping, polymorphism, dependent types, closures, or other bells and whistles.

Fred's user-defined types are only tagged unions of product types. And rather than assume every field is mutable, fields need to be marked mutable explicitly. Immutability isn't central to my project, but it does give us some extra knowledge to avoid more scanning.

This makes it easy to represent all the types in a program as a directed graph where the nodes are types. The fields inside every type can be represented as edges going from that type to the type of the field.

Now that we have a graph of types, we can see that two objects `a` and `b` (not necessarily distinct) of types `A` and `B`, respectively, can only form a cycle if:

- `A` and `B` form a cycle,
- and somewhere along the path from `A` to `B` or `B` to `A`, there's a mutable field.

Although I may be lazy, Fred is not, and so there is currently no way to create cycles using only immutable fields. I don't feel like proving this or Googling for existing proofs of it.

Now that we know that certain objects cannot form cycles with certain other objects, we can apply this knowledge at runtime. When recursively scanning the objects reachable from a PCR, every time we come across some object, we can avoid scanning those of its children that can never form a cycle with that object (based on their types). We will also only add an object to the list of PCRs in the first place if it's possible for that object to be part of a cycle (note the two rules above).

4.2. Quadratic scanning problem

However, if done naively, this can result in not all garbage being collected in a single sweep of the list of PCRs [5]. A quick fix for this would be to go over the list of PCRs multiple times until all garbage is gone, but this makes cycle collection quadratic in the number of objects.

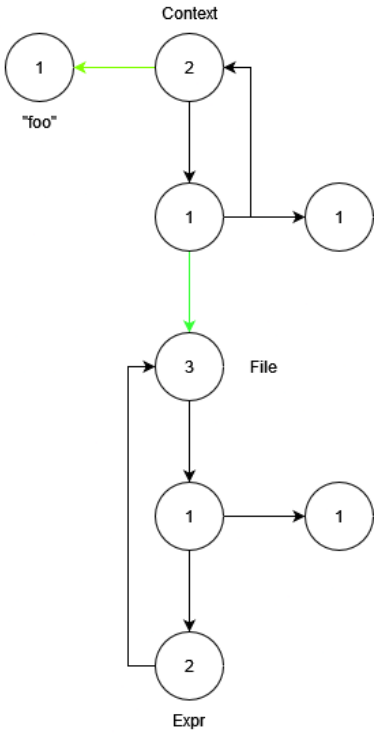
In this subsection, I will demonstrate this problem with an example. Suppose you are creating a compiler and you have the following types. You can have `Context -> FileList -> Context` cycles, as well as `File -> ExprList -> Expr -> File` cycles.

```
data Context = Context {
  name: str,
  mut files: FileList
}
data FileList
  = FileNil {}
  | FileCons {
    ctx: Context,
    head: File,
    tail: FileList,
  }
data File = File {
  mut exprs: ExprList
}
data ExprList
  = ExprNil {}
  | ExprCons {
    head: Expr,
    tail: ExprList
  }
data Expr = Expr {
  file: File,
  // other stuff here
}
```

And for whatever reason, you have the following code that creates a context with a file that contains one expression:

```
let ctx = Context { name: "foo", files: FileNil {} } in
let file = File { exprs: ExprNil {} } in
let expr = Expr { file: file } in
ctx.files = FileCons { ctx: ctx, head: file, tail: ctx.files };
file.exprs = ExprCons { head: expr, tail: file.exprs }
```

After running that code, this is what the graph of objects looks like:



The green edges in the diagram above are references that are known not to introduce any cycles. Therefore, when doing mark-scan, we will not follow them (this is our modification from the previous section, not part of lazy mark scan). There are other references that don't cause cycles in there, but we can't know this at compile-time.

At some point, the variables `ctx`, `file`, and `expr` will go out of scope, so the `Context`, `File`, and `Expr` objects will all have their refcounts decremented before being added to the list of PCRs. All the objects in the diagram above have become garbage and are eagerly waiting to be freed.

Let's trace what our naively modified lazy mark scan algorithm would do here:

- All objects start out colored black.
- First, we go to every object reachable from the `Context`, `File`, and `Expr` objects (without traversing green edges) and mark it gray.
  - The reference count of every object reachable from these gray objects is decremented, as if the gray objects have been deleted.
  - After doing this, the `Context` and `FileList` objects have a refcount of 0. `File` has a refcount of 1, while the `Expr` and `ExprList` objects have a refcount of 0.
- Now we run the Scan operation on those same objects.
  - The `Context` and `FileList` will have refcounts of 0, so they'll be colored white.
  - The `File` has refcount 1, so the ScanBlack operation will be run on it.
    - At the end of this, `File` and everything it points to (the bottom cycle) will be colored black.
    - The `File` object will have refcount 2, while the `Expr` and `ExprList` objects will have refcount 1.
- Now we collect white objects.
  - The `Context` and `FileList` will be collected, since they're white.
  - But the `File`, `Expr`, and `ExprList` objects won't be collected, since they're black!

`File` lives because it has a reference from the `FileCons` object, and it keeps the rest of the bottom cycle alive. Thus, all the scanning we did on the bottom cycle was in vain. We'll have to go back and re-scan the `File` object.

Therefore, it is not enough to simply not traverse known acyclic edges. Fortunately, the solution to this is pretty simple.

4.3. Fixing the quadratic scanning problem

You may notice above that because we don't traverse references known not to cause cycles, processing the `Context` object happens completely separately from processing the `File` and `Expr` objects. We could have processed the `Context` object first, found it to be garbage, decremented the `File` object's reference count, and then processed the `File` and `Expr` objects together. This way, all of the objects would have been marked as garbage without processing any PCR multiple times.

In general, how do we determine which objects should be processed before which objects? We can do this based on which objects can reference which objects (directly or indirectly). A `Context` object can refer to `File` and `Expr` objects, so it must be processed before them. `File` and `Expr` objects can both refer to each other, so they must be processed together.

A simple way to determine this in practice is to take the graph of types and partition it into its **strongly-connected components** (**SCCs**). These SCCs have an ordering. Objects from the same SCC will all be processed together, but objects from earlier SCCs will be processed before objects from later SCCs.

To make this more concrete, let's return to the previous problematic example. There, you have the following SCCs:

1. `[Context, FileList]`
2. `[File, ExprList, Expr]`
3. `[str]`

I've listed them in the order they would be processed, although the second and third SCCs can be swapped, since they are unrelated.

This time, we would process only the `Context` PCR first, and we would find it and the two `FileList` objects to be garbage. This would cause the string `"foo"` and the `File` object to have their refcounts decremented. The string would be freed at this point.

Next, we would process the `File` and `Expr` PCRs together. This time, the refcount of `File` would be only 1, since it is being kept alive only by the cycle it's part of. After processing, the whole cycle would have refcount 0 and be freed.

As a bonus, grouping objects by SCC and processing them separately also lets us make cycle collection more incremental. If necessary, we can process PCRs from the first SCC, then continue on with the rest of the program rather than process all of the remaining PCRs too. I didn't explore this in my project, though.

5. Implementation

Fred was created to try out this algorithm. The implementation can be found at https://github.com/ysthakur/fred. The language uses automatic reference counting and is compiled to C. Partly because it is compiled to C and partly because I made it, it involves copious amounts of jank. When I have time, I will try to get rid of some of this awfulness, as well as document my code better, but in the meantime, Fred is mostly functional (not functional like Haskell, functional like an alcoholic).

PCRs are stored in buckets, which are linked lists. Each bucket stores PCRs from only one SCC. These buckets are themselves stored in a linked list, and they are kept in sorted order according to their SCC. Note: while writing this report, I switched to using an array for performance reasons (discussed in Section 6.2).

There's a `runtime.h` header file that does all the PCR stuff. It contains `addPCR`, `removePCR`, and `processAllPCRs` functions.

When an object's refcount is decremented and it doesn't hit 0, it's added to the list of PCRs using `addPCR` and its `addedPCR` is set to true. The next time the object's refcount is decremented, if `addedPCR` is true, there won't be any need to call `addPCR`.

`removePCR` is called whenever an object's refcount is decremented and hits 0, since it will be freed. I could also have chosen to call `removePCR` on an object every time its refcount was incremented (since that means it's cyclic garbage). However, this could have resulted in `addPCR` and `removePCR` repeatedly being called over and over on the same object as it was passed around to various places.

That said, I saw when writing this report that [3] colors objects black whenever their refcounts are incremented (without removing from the list of PCRs). This seems like a reasonable optimization, and I could implement it.

`processAllPCRs` iterates over all the PCR buckets and runs lazy mark scan on each bucket individually. When it's done, all PCRs and all PCR buckets are freed.

`processAllPCRs` buckets is only called at the end of the `main` function. In real life, you'd want to throw in a check every time you allocate a new object or something that determines whether or not to collect cycles, but calling `processAllPCRs` only at the end of the program was good enough for the short programs I was testing. The user can still insert calls to `processAllPCRs` wherever they want.

One minor advantage of putting everything into a bucket is that `removePCR` is faster. You don't need to search through a single flat list of all PCRs. Once you find the right bucket for that PCR's SCC (which is linear in the number of SCCs), you only need to search within that bucket.

5.1. Disabling the optimization

For benchmarking, I needed to be able to run code both with and without my optimization. So, I added a compiler flag `--lazy-mark-scan-only` to output code that uses base lazy mark scan rather than my algorithm. It doesn't actually use lazy mark scan, but it's close enough. The only change it makes is treat every type as if they're in the same SCC. This means that all objects are put into the same bucket and are scanned together, just as with lazy mark scan.

6. Benchmarks

I would like to preface this section by noting that it is complete bogus and that you can safely skip it. Nevertheless, I have used these benchmarks to convince myself that my algorithm is vastly superior to the base lazy mark scan algorithm. Feel free to do the same.

I have two benchmarks at the moment. They can be found in the `benchmarks` folder.

The benchmarks work by running a piece of code a bunch of times, then looking at how much the processor's timestamp counter increased (using `rdtscp`) as well as the processor time (using `clock()`). Since within each benchmark, the code being timed is run lots of times, I only recorded the times after running each benchmark program once, rather than running each program multiple times and noting the mean and range.

6.1. `game.fred`

Here's the code. This program is supposed to be a game, except it does basically nothing. It demonstrates a case where normal lazy mark scan will unnecessarily scan a bunch of objects, but my algorithm won't.

It has the following types:

```
data Player
  = Player { store: Store }
  // This exists only to make the compiler think that Player can be involved in
cycles
  | PlayerCyclic {
    mut player: Player
  }
data Store = Store { datums: Data }
data Data
  = DataCons {
    value: int,
    // This is mut only so the compiler thinks there can be a cycle at runtime
    mut next: Data
  }
  | DataNil {}
```

`Store` represents some kind of shared state or resources or something that all `Player` objects have a reference to. This sort of thing is probably more common in Java than in a functional language, but whatever.

This is what `game.fred` does:

1. Create a ginormous `Store` object
2. Do the following 50,000 times:
   1. Create a `Player` object
   2. Increment and decrement its refcount so that it's added to the list of PCRs
   3. Invoke `processAllPCRs()`

The `processAllPCRs()` call above will cause the `Player` object to be scanned. When it's scanned, with my algorithm, the `Store` object won't be scanned, because it's in a separate SCC. But with base lazy mark scan, the `Store` object will have to be scanned, so it will be slower.

Here are the results:

Lazy mark scan only?	Timestamp counter	Clock (s)
No	74647376	0.028586
Yes	29478684752	11.289244

I'd go into how my algorithm is orders of magnitude faster than base lazy mark scan, but this benchmark means basically nothing. I can tune the results by changing the size of the `Store` or the number of `Player` objects created. The only thing it really tells you is that there can be cases where my algorithm is faster than lazy mark scan.

6.2. `stupid.fred`

If the previous benchmark wasn't artificial enough for you, this one definitely will be. I wanted to come up with something where my algorithm would perform worse than base lazy mark scan. This can happen if the overhead from inserting PCRs into the right bucket (sorted) is too high. You need to have a bunch of SCCs, and you need to often have objects from higher SCCs being added to the list of PCRs after objects from lower SCCs. This is a situation that probably isn't uncommon in real codebases.

I couldn't actually come up with a decent example, so I wrote a script to do it for me. The script first generates 200 types. Each type T_{i+1} has a field of type T_i. The script then generates an object of type

T_{199}. Then it goes from T_{199} down to T_0, adding objects to the list of PCRs. With base lazy mark scan, adding PCRs is a constant time operation, but with my algorithm, it's linear time, since an object of type T_i here would have to go through $199 - i$ objects first.

All of the stuff described above is then run 50,000 times. Here are the results:

Lazy mark scan only?	Timestamp counter	Clock (s)
No	27741037106	10.623692
Yes	11054113602	4.233204

But I realized when writing this report that this problem only happens with my specific implementation. The PCR buckets can be held in an array rather than a linked list. This is fine because the number of SCCs is known statically and will always be low. When adding a new PCR, we can index into this array using the PCR's SCC. Adding PCRs now becomes a constant-time operation.

After making this modification, I ran the stupid benchmark again. Now, both algorithms have about the same performance!

Lazy mark scan only?	Timestamp counter	Clock (s)
No	8890733476	3.404776
Yes	10184751983	3.900381

Now that this change has been made, I can't think of any cases where my algorithm would perform noticeably worse than base lazy mark scan.

7. Conclusion

I made a language that uses automatic reference counting and in it, I implemented a compiler optimization to make lazy mark scan avoid scanning certain objects based on type information. Whether this optimization would actually be better than base lazy mark scan on real code remains to be seen, but it should never perform significantly worse than base lazy mark scan.

It should limit how much of the heap you scan, because when processing PCRs from one SCC, you only ever scan objects from that same SCC. This lets you avoid entire subgraphs, which could be useful if you have objects close to the GC roots constantly being deleted. I have no idea how often a situation like that would arise in real code, though.

The work I've done here is more applicable to functional languages than languages like Java, where a lot of garbage collection research is focused. Java has subtyping, which makes it harder to tell which types can form cycles with which other types. Mutability is also a lot more common, partly because a lot of Java developers don't use `final` as much as they should, so from the compiler's perspective, it's harder to tell if cycles can happen. It's also just more common to have cycles [citation needed I guess but whatever].

8. Future work

8.1. Formal verification

I worry a little that this algorithm isn't actually sound. It would be nice to prove using Coq or something that, if you group and sort PCRs according to the SCC of their type and process each group separately in order, you'll still collect all cycles.

8.2. Applying this to newer algorithms

The papers I was working off are pretty old. My implementation was built on top of the original lazy mark scan algorithm, even though cycle collection for reference counting has come a long way since. Even back in 2001, [6]/[3] had a complicated concurrent cycle collector, and I have no idea if my stuff applies there. If newer algorithms can't be improved using the stuff I talked about here, then my project won't have been very useful.

In addition to concurrency, collectors can also use tracing rather than trial deletion for finding cycles. The stuff I worked on here isn't immediately applicable outside of trial deletion, but it would be interesting to explore how information about types could make tracing more focused or something.

8.3. Adding this to existing languages

There's nothing special about Fred as a language, and so the compiler optimizations and runtime worked on here can be applied to an existing compiled language. Such a language would have plenty of code available already, and this could be used for creating more meaningful benchmarks than the ones I made above.

8.4. Double reference counts

In [5], Chang et al. propose using two reference counts. Similarly to that, we can use two reference counts: an acyclic refcount for known acyclic references from other SCCs, and a cyclic refcount for possibly cyclic references from the same SCC. As long as an object's acyclic refcount is positive, it must be alive, so it doesn't need to be treated as a PCR.

8.5. Restrictions to help the compiler

Aside from immutable fields, the compiler doesn't have much information to help it determine what can form cycles with what. It would be interesting to explore possible restrictions that programmers could put on function parameters or whatever in order to give the compiler more information and let it make more optimizations.

For example, if you have a `ClosureWrapper` type that holds a closure, you don't want cycles between the closure and `ClosureWrapper`. So you can say that `ClosureWrapper`'s closure-holding field will only accept closures that don't have references to any `ClosureWrapper` objects.

8.6. Big cycles

The compiler optimizations here won't be very useful if nearly all of the objects are of types from the same SCC. If every object is in the same SCC, it's basically just doing base lazy mark scan. In all of the Java benchmarks tried by [5], most of the created objects are cyclic. Although I am mainly trying to improve things for functional languages, not Java, and the cyclic objects didn't necessarily all have types in the same SCC, this is a problem worth finding a solution to. I don't think there's a way to solve this problem entirely without the programmer helping the compiler out a bit.

9. Why name it Fred?

I was going to name it Foo, but there's already an esolang by that name that's fairly well-known (by esolang standards). So I went to the Wikipedia page on metasyntactic variables and picked "fred." I figured that if I needed to, I could pretend that it was something meaningful, like maybe an acronym or the name of a beloved childhood pet.

For example, I could say that when I was young, I had a cute little hamster called Freddie Krueger, so named because of the hamster-sized striped red sweater my grandmother had knitted for him, as

well as his proclivity for murdering small children. In his spare time, Fred would exercise on his hamster wheel, or as he liked to call it, his Hamster Cycle.

But one day, I came home to find Fred lying on the hamster cycle, unresponsive. The vet said that he'd done too much running and had had a heart attack. I was devastated. It was then that I decided that, to exact my revenge on the cycle that killed Fred, I would kill all cycles.

Bibliography

[1] R. D. Lins, "Cyclic Reference Counting With Lazy Mark-Scan," *Information Processing Letters*, vol. 44, no. 4, pp. 215–220, Dec. 1992, Accessed: Sep. 18, 2024. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(92)90088-D

[2] A. D. Martínez, R. Wachenchauzer, and R. D. Lins, "Cyclic reference counting with local mark-scan," *Information Processing Letters*, vol. 34, no. 1, pp. 31–35, Feb. 1990, doi: 10.1016/0020-0190(90)90226-N.

[3] D. F. Bacon and V. T. Rajan, "Concurrent Cycle Collection in Reference Counted Systems," in *ECOOP 2001 — Object-Oriented Programming*, J. L. Knudsen, Ed., Berlin, Heidelberg: Springer, 2001, pp. 207–235. doi: 10.1007/3-540-45337-7_12.

[4] P. G. Joisha, "Compiler optimizations for nondeferred reference: counting garbage collection," in *Proceedings of the 5th international symposium on Memory management*, in ISMM '06. New York, NY, USA: Association for Computing Machinery, Jun. 2006, pp. 150–161. doi: 10.1145/1133956.1133976.

[5] J. Morris Chang, W.-M. Chen, P. A. Griffin, and H.-Y. Cheng, "Cyclic reference counting by typed reference fields," *Computer Languages, Systems & Structures*, vol. 38, no. 1, pp. 98–107, Apr. 2012, doi: 10.1016/j.cl.2011.09.001.

[6] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith, "Java without the coffee breaks: a nonintrusive multiprocessor garbage collector," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, in PLDI '01. New York, NY, USA: Association for Computing Machinery, May 2001, pp. 92–103. doi: 10.1145/378795.378819.