

Contents

Introduction	1
Background	1
Lazy mark scan	1
Algorithm	1
Avoiding scanning children based on type	1
Quadratic scanning problem	2
Fixing the quadratic scanning problem	4
Implementation	5
Future work	5
Bibliography	5
Why name it FRED?	5

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.

Background

Leo, you can probably skip/skim these first couple sections, they mostly just exist for future me.

Lazy mark scan

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].

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.

This still requires scanning a bunch of objects. There are many things you can do to improve reference counting performance significantly, and I ran into a bunch of such articles. However, I only found a couple that make optimizations based on compile-time information. This is possibly because
TODO

Algorithm

Avoiding scanning children based on type

In a statically typed language, some guarantees can be made about whether or not objects of one type can ever form cycles with objects of another type. At runtime, this lets us reduce the scanning we do.

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 algebraic data types, I think they're called? They're 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).

Quadratic scanning problem

However, if done naively, this can result in garbage not being found in a single sweep of the list of PCRs [3]. 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.

Below, I will give some example code that triggers this problem. 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.

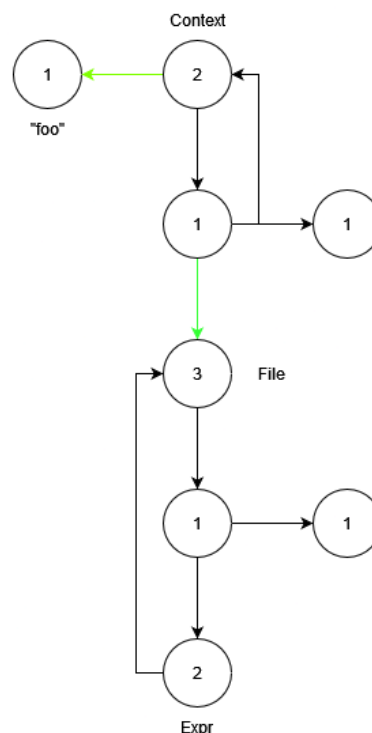
```
data Context = Context {
  name: str,
  mut files: FileList
}
data FileList
  = FileNil {}
  | FileCons {
    ctx: Context,
    head: File,
    tail: FileList,
  }
data File = File {
  mut exprs: ExprList
}
data ExprList
  = ExprNil {}
  | ExprCons {
    head: Expr,
    tail: ExprList
  }
```

```
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. I'm going to call these green edges "innocent", because I don't know what sort of terms are actually used for them by real researchers.

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, not knowing that rather than nirvana, all they will get is an endless cycle of rebirth and deallocation, until your laptop finally stops working and you have to throw it away. Thankfully, planned obsolescence will eventually bring nirvana to these objects in a matter of years.

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

- 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.
- TODO finish this

Now the stuff in the top cycle has been correctly marked as garbage, but not the stuff in the bottom cycle. File lives because it has a reference from the FileCons object, and it keeps the rest of the bottom cycle alive. All the scanning we did on the bottom cycle was in vain, because we'll have to go back and repeat it now.

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

Fixing the quadratic scanning problem

You may notice above that because we don't traverse innocent edges, 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.

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 after finals, 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 (functional as in alcoholic).

Future work

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](http://dx.doi.org/10.1016/0020-0190(92)90088-D)
- [2] A. D. Martínez, R. Wachsenauzer, 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] 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.

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.

I could maybe say that when I was young, I had a cute little hamster called Freddie Krueger, so named because of the 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.