

How to implement closures without gc?

Asked 16 years, 3 months ago Modified 11 years, 5 months ago

Viewed 4k times



14



I'm designing a language. First, I want to decide what code to generate. The language will have lexical closures and prototype based inheritance similar to javascript. But I'm not a fan of gc and try to avoid as much as possible. So the question: Is there an elegant way to implement closures without resorting to allocate the stack frame on the heap and leave it to garbage collector?

My first thoughts:

1. Use reference counting and garbage collect the cycles (I don't really like this)
2. Use spaghetti stack (looks very inefficient)
3. Limit forming of closures to some contexts such a way that, I can get away with a return address stack and a locals' stack.

I won't use a high level language or follow any call conventions, so I can smash the stack as much as I like.

(Edit: I know reference counting is a form of garbage collection but I am using gc in its more common meaning)

garbage-collection

code-generation

language-design

Share

Improve this question

Follow

edited Dec 7, 2011 at 19:23



Joel Coehoorn

415k ● 114 ● 577 ● 813

asked Sep 18, 2008 at 1:43



artificialidiot

5,369 ● 32 ● 27

3 What does it mean to be "not a fan of GC"? Keep in mind that reference counting is a form of garbage collection. Also, what does "lexical closures" mean in a situation where you "won't... follow any call conventions"? – Allen Sep 18, 2008 at 1:47

1 call conventions like stdcall, fastcall, cdecl, thiscall ...
– artificialidiot Sep 18, 2008 at 1:52

1 @Allen Reference counting is not garbage collection. It's a form of automatic management. Not every kind of automatic memory management is garbage collection. – user529758 Feb 4, 2014 at 17:56

13 Answers

Sorted by:

Highest score (default)





13



This would be a better question if you can explain what you're trying to avoid by not using GC. As I'm sure you're aware, most languages that provide lexical closures allocate them on the heap and allow them to retain references to variable bindings in the activation record that created them.

The only alternative to that approach that I'm aware of is what `gcc` uses for nested functions: create a trampoline for the function and allocate it on the stack. But as the `gcc` manual says:

If you try to call the nested function through its address after the containing function has exited, all hell will break loose. If you try to call it after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

Short version is, you have three main choices:

- allocate closures on the stack, and don't allow their use after their containing function exits.
- allocate closures on the heap, and use garbage collection of some kind.

- do original research, maybe starting from the region stuff that ML, Cyclone, etc. have.

Share Improve this answer

answered Sep 18, 2008 at 2:00

Follow



Allen

5,090 ● 24 ● 30

gcc's implementation of closures is pretty weak that, it only reduce explicit context passing, in my opinion. I want to see how far I can go without gc. – [artificialidiot](#) Sep 18, 2008 at 2:05



[This thread](#) might help, although some of the answers here reflect answers there already.

9

One poster makes a good point:



It seems that you want garbage collection for closures "in the absence of true garbage collection". Note that closures can be used to implement cons cells. So your question seem to be about garbage collection "in the absence of true garbage collection" -- there is rich related literature. Restricting problem to closures does not really change it.

So the answer is: **no**, there is no elegant way to have closures and no real GC. The best you can do is some hacking to restrict your closures to a particular type of closure. All this is needless if you have a proper GC.

So, my question reflects some of the other ones here - why do you not want to implement GC? A simple mark+sweep or stop+copy takes about 2-300 lines of (Scheme) code, and isn't really that bad in terms of programming effort. In terms of making your programs slower:

1. You can implement a more complex GC which has better performance.
2. Just think of all the memory leaks programs in your language won't suffer from.
3. Coding with a GC available is a blessing. (Think C#, Java, Python, Perl, etc... vs. C++ or C).

Share Improve this answer

answered Jan 15, 2009 at 14:43

Follow



Claudiu

229k ● 173 ● 503 ● 697



I understand that I'm very late, but I stumbled upon this question by accident.

9



I believe that full support of closures indeed requires GC, but in some special cases stack allocation is safe.

Determining these special cases requires some escape analysis. I suggest that you take a look at the [BitC language papers](#), such as [Closure Implementation in BitC](#).



(Although I doubt whether the papers reflect the current plans.) The designers of BitC had the same problem you do. They decided to implement a special non-collecting mode for the compiler, which denies all

closures that might escape. If turned on, it will restrict the language significantly. However, the feature is not implemented yet.

I'd advise you to use a collector - it's the most elegant way. You should also consider that a well-built garbage collector allocates memory faster than malloc does. The BitC folks really do value performance and they still think that GC is fine even for the most parts of their operating system, Coyotos. You can mitigate the downsides by simple means:

- create only a minimal amount of garbage
- let the programmer control the collector
- optimize stack/heap use by escape analysis
- use an incremental or concurrent collector
- if somehow possible, divide the heap like Erlang does

Many fear garbage collectors because of their experiences with Java. Java has a fantastic collector, but applications written in Java have performance problems because of the sheer amount of garbage generated. In addition, a bloated runtime and fancy JIT compilation is not really a good idea for desktop applications because of the longer startup and response times.

Share Improve this answer

answered Apr 26, 2009 at 3:14

Follow



ahnurmi

116 ● 1 ● 5

Full closure support *does not* requires GC, at the cost of memory unsafety involved with captures of dangling references. C++'s closure objects work exactly like this. (Note that closure does not means it is free from the so-called funargs problems.) If unsafe captures can be further proved away from the code (e.g. by a type system, or even by disallowing reference captures totally), it will be safe (albeit with limitations on the code). – [FrankHB](#) Jun 22, 2020 at 3:51



4



The C++ 0x spec defines lambdas without garbage collection. In short, the spec allows non-deterministic behavior in cases where the lambda closure contains references which are no longer valid. For example (pseudo-syntax):



```
(int)=>int create_lambda(int a)
{
    return { (int x) => x + a }
}

create_lambda(5)(4)    // undefined result
```

The lambda in this example refers to a variable (**a**) which is allocated on the stack. However, that stack frame has been popped and is not *necessarily* available once the function returns. In this case, it would probably work and return **9** as a result (assuming sane compiler semantics), but there is no way to guarantee it.

If you are avoiding garbage collection, then I'm assuming that you also allow explicit heap vs. stack allocation and (probably) pointers. If that is the case, then you can do

like C++ and just assume that developers using your language will be smart enough to spot the problem cases with lambdas and copy to the heap explicitly (just like you would if you were returning a value synthesized within a function).

Share Improve this answer

answered Sep 18, 2008 at 2:08

Follow



[Daniel Spiewak](#)

55.1k ● 14 ● 111 ● 120

-
- 1 Thanks for the suggestion, but I don't want the programmer to keep track of frames. One use case in my mind is, using a function as an event handler, where the stack frame is certainly unavailable. – [artificialidiot](#) Sep 18, 2008 at 2:18
-

Right, don't have them keep track of frames, but force them to be aware of what's on the stack and what's on the heap. If you're not having garbage collection, then you'll need this anyway just to make functions work. – [Daniel Spiewak](#) Sep 18, 2008 at 2:53



4

Use reference counting and garbage collect the cycles (I don't really like this)



It's possible to design your language so there are no cycles: if you can only make new objects and not mutate old ones, and if making an object can't make a cycle, then cycles never appear. Erlang works essentially this way, though in practice it does use GC.



Share Improve this answer

answered Oct 11, 2008 at 10:06

Follow



Darius Bacon

15.1k ● 6 ● 56 ● 53



3



If you have the machinery for a precise copying GC, you could allocate on the stack initially and copy to the heap and update pointers if you discover at exit that a pointer to this stack frame has escaped. That way you only pay if you actually do capture a closure that includes this stack frame. Whether this helps or hurts depends on how often you use closures and how much they capture.

You might also look into C++0x's approach ([N1968](#)), though as one might expect from C++ it consists of counting on the programmer to specify what gets copied and what gets referenced, and if you get it wrong you just get invalid accesses.

Share Improve this answer

answered Sep 18, 2008 at 2:04

Follow



puetzk

10.8k ● 3 ● 30 ● 33

Oh I have forgotten that one, thanks for reminding! I'm a bit reluctant to move around memory regions though.

– [artificialidiot](#) Sep 18, 2008 at 2:10

"you could allocate on the stack initially and copy to the heap and update pointers if you discover at exit that a pointer to this stack frame has escaped". IIRC, that has been suggested in the literature and examined but it adds significant complexity and does not improve performance.

– [J D](#) Jul 17, 2013 at 19:30



2



Or just don't do GC at all. There can be situations where it's better to just forget the memory leak and let the process clean up after it when it's done.

Depending on your qualms about GC, you might be afraid of the periodic GC sweeps. In this case you could do a selective GC when an item falls out of scope or the pointer changes. I'm not sure how expensive this would be though.

@Allen

What good is a closure if you can't use them when the containing function exits? From what I understand that's the whole point of closures.

Share Improve this answer

answered Sep 18, 2008 at 2:05

Follow



Kyle Cronin

79k ● 45 ● 151 ● 167

You can still pass it to stuff you call. Same value as any other stack-allocated data structure, really. I'd say it's about half the point of closures. – [Allen](#) Sep 18, 2008 at 2:07

@Allen It is more like higher order functions. I think you don't necessarily have to have an implementation of closures for the case you mention. – [artificialidiot](#) Sep 18, 2008 at 2:23

er... sorry, but "forget the memory leak and let the process clean up after it when it's done." sounds like a terrible, terrible way to build a programming language – [Claudiu](#) Jan 15, 2009 at 14:35

@Claudiu: That's just the way all languages without GC worked, like Pascal, C, C++, etc. – [Mike Dunlavey](#) Jun 1, 2010 at 18:37

@MikeDunlavey: I think the suggestion here is to leak all memory and not even provide a `free` function which is certainly not how Pascal, C, C++ etc. work! :-)) – [J D](#) Jul 17, 2013 at 19:39



2



You could work with the assumption that all closures will be called eventually and exactly one time. Now, when the closure is called you can do the cleanup at the closure return.

How do you plan on dealing with returning objects? They have to be cleaned up at some point, which is the exact same problem with closures.

Share Improve this answer

answered Jan 15, 2009 at 14:17


Follow



[user4891](#)

937 ● 2 ● 9 ● 14

@DevinJeanpierre Shall I ask a question about that? That sounds very interesting, especially given that closures in Rust can be involved in asynchronous/iterative processing.
– [bright-star](#) Oct 19, 2016 at 6:18

- 1 @TrevorAlexander Rust has changed a *lot* since I wrote that comment, especially the situation with closures, which have been rewritten twice. I haven't used it in a while. At the time, there existed a closure type that could only be called once, and other closure types with different restrictions. Today, I'd look at doc.rust-lang.org/beta/book/closures.html for more.
- Devin Jeanpierre Nov 2, 2016 at 23:22 
-



2



So the question: Is there an elegant way to implement closures without resorting to allocate the stack frame on the heap and leave it to garbage collector?



GC is the only solution for the general case.



Share Improve this answer

answered Jul 17, 2013 at 19:41

Follow



J D

48.6k ● 14 ● 174 ● 277



1



Better late than never?

You might find this interesting: [Differential Execution](#).

It's a little-known control structure, and its primary use is in programming user interfaces, including ones that can change dynamically while in use. It is a significant alternative to the Model-View-Controller paradigm.



I mention it because one might think that such code would rely heavily on closures and garbage-collection,

but a side effect of the control structure is that it eliminates both of those, at least in the UI code.

Share Improve this answer

Follow

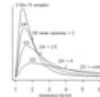
edited May 23, 2017 at 12:00



Community Bot

1 ● 1

answered Jun 1, 2010 at 18:44



Mike Dunlavey

40.6k ● 15 ● 94 ● 138



Create multiple stacks?

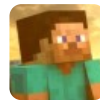
0

Share Improve this answer

Follow



answered Oct 21, 2008 at 13:39



Chris Becke

36k ● 12 ● 81 ● 153



I've read that the last versions of ML use GC only sparingly

0

Share Improve this answer

Follow



answered Jan 15, 2009 at 14:34



niv



If by "ML" you mean the metalanguage family of programming languages (CML, SML, OCaml etc.) then I'm afraid this is not usually true. Those languages (and Scala and Haskell) typically have huge allocation rates and do a lot of unnecessary boxing (e.g. floats, tuples, complex numbers). The only derivative language that has the potential to allocate much less is F# but it still allocates much more heavily than necessary, e.g. complex numbers are unboxed but tuples are still boxed. Consequently, GC performance is absolutely critical in implementations of these languages. – J D Jul 17, 2013 at 19:34

In fact, I built an ML-like language and bespoke VM called HLVM designed specifically to reduce GC stress by avoiding allocations and indirections when possible and some of the results were amazing (beating Java, Haskell, OCaml and MLton SML). flyingfrogblog.blogspot.co.uk/2010/01/... – J D Jul 17, 2013 at 19:37



0



I guess if the process is very short, which means it cannot use much memory, then GC is unnecessary. The situation is analogous to worrying about stack overflow. Don't nest too deeply, and you cannot overflow; don't run too long, and you cannot need the GC. Cleaning up becomes a matter of simply reclaiming the large region that you pre-allocated. Even a longer process can be divided into smaller processes that have their own heaps pre-allocated. This would work well with event handlers, for example. It does not work well, if you are writing compiler; in that case, a GC is surely not much of a handicap.

Follow

A square profile picture with a purple and white geometric pattern of interlocking diamonds.

Jason

1 ● 1

