Haskell recursion and memory usage

Asked 12 years ago Modified 6 years, 8 months ago Viewed 8k times

32

I'm getting comfortable with the idea of replacing loops with recursion. I'm fiddling around with a pet project, and I wanted to test some text input functionality so I wrote up a little command line interface that repeatedly asks for input until it receives a specific quit command.

It looks something like this:

```
getCommandsFromUser = do
    putStrLn "Enter command: "
    keyboardInput <- getLine
    let command = map toLower keyboardInput
    if command == "quit"
        then
            putStrLn "Good-bye!"
        else do
            -- stuff
            -- more stuff
            putStrLn ("Sending command: " ++ commandURI)
            simpleHTTP $ getRequest commandURI
            getCommandsFromUser

main = do
    getCommandsFromUser
```

This works exactly as expected, but coming from a C/Java background it still tickles the deep, dark, unconscious parts of my brain and makes me want to break out in hives because I can't shake the thought that every single recursive call of `getCommandsFromUser` being made is creating a new stack frame.

Now, I don't know anything about IO, monads, state, arrows, etc. yet. I'm still working my way through Real World Haskell, I haven't reached that part yet, and some of this code is pattern matched off things that I found on Google.

In addition, I know that the whole point of the GHC is that it's a maddeningly optimizing compiler that is designed to do incredible things such as beautiful unrolling of tail recursive functions and the like.

So can somebody explain whether or not this implementation is "correct", and if so explain to me what is going on behind the scenes that would prevent this program from blowing up if it were put in the hands of an infinite number of monkeys?

I know what tail call optimization is. I'm more concerned about how it works in this case, what with the actions and general functional impurity going on.

This question wasn't so much based on the idea that I was confused about how Haskell used the stack and that I was expecting it to work like an imperative language; it was based in the fact that I had no idea how Haskell handled the stack and wanted to know what it was doing differently from traditional C-like languages.

memory recursion functional-programming haskell

Share
Improve this question
Follow

edited Apr 13, 2018 at 15:33

Aaron Hall ♦
394k ● 91 ● 412 ● 339

asked Dec 8, 2012 at 21:41

Doug Stephen
7,341 ● 1 ● 39 ● 46

1 I suggest you lookup "tail call optimization" either here on SO or on the web in general. It's a general compilers/languages concept and implemented not just by Haskell compilers but also GCC and the like. – Thomas M. DuBuisson Dec 8, 2012 at 22:15

6 @ThomasM.DuBuisson It's actually not obvious from a strict interpretation of TCO that it applies to things on the right side of a `(>>=)`. In fact, for many monads, it doesn't. A lot of care is taken to make sure that it does apply in IO. – Carl Dec 8, 2012 at 22:50

@ThomasM.DuBuisson As indicated in the original question, I'm familiar with tail call optimization and the general idea of how it works. I'm asking about it in the case of Haskell, and specifically, its use in this (monadic, impure) case. – Doug Stephen Dec 8, 2012 at 22:51

8 "tail call optimization" isn't really a phrase that makes sense when you're talking about evaluating non-strict languages like Haskell. Evaluation happens in a completely different way; "TCO" is what you get by default, not really an "optimization" at all. – shachaf Dec 8, 2012 at 22:58

4 Answers

Sorted by: Highest score (default)

44

Don't worry quite so much about the stack. There is nothing fundamental that says function calls have to be implemented using stack frames; that is merely one possible technique for implementing them.

Even when you have "the stack", there's certainly nothing that says the stack has to be limited to a small fraction of available memory. That's essentially a heuristic tuned to imperative programming; where you don't use recursion as a problem solving technique, very deep call stacks tend to result from infinite-recursion bugs, and limiting the stack size to something quite small means such programs die quickly instead of consuming all available memory and swap and then dying.

To a functional programmer, having a program terminate "run out" of memory to make more function calls when the computer still has gigabytes of RAM available is a

ridiculous flaw in the language design. It would be like C limiting loops to some arbitrary number of iterations. So even if a functional language *is* implementing function calls by using a stack, there would be a strong motivation to avoid using the standard tiny stack we know from C if possible.

In fact, Haskell does have *a* stack which can overflow, but it's not the call stack you're familiar with from C. It is quite possible to write non-tail-recursive functions which infinitely recurse and will consume all available memory without hitting a limit on call depth. The stack Haskell does have is used to track the "pending" values that need to be evaluated a bit more in order to make a decision (I'll go into this a bit more later). You can read in more detail about this kind of stack overflow here.

Lets work through an example to see how your code could be evaluated.[1] I'll use an even simpler example than yours though:

```
main = do
    input <- getLine
    if input == "quit"
        then
            putStrLn "Good-bye!"
        else do
            putStrLn $ "You typed: " ++ input
            main
```

Haskell's evaluation is lazy[2]. Simplistically, that means it will only bother to evaluate a term when it needs the value of that term to make a decision. For example, if I compute `1 + 1` and then prepend the result of that to the front of a list, it can be left as a "pending" `1 + 1` in the list[3]. But if I use `if` to test whether the result was equal to 3, *then* Haskell will need to actually do the work of turning `1 + 1` into `2`.

But if that's all there was to it, nothing would ever happen. The entire program would just be left as a "pending" value. But there is an outer driver that needs to know what IO action `main` evaluates to, in order to execute it.

Back to the example. `main` is equal to that `do` block. For `IO`, a `do` block makes an a big `IO` action out of a series of smaller ones, which must be executed in order. So the Haskell runtime sees `main` evaluating to `input <- getLine` followed by some unevaluated stuff which it doesn't need yet. That's enough to know to read from the keyboard and call the resulting `String` `input`. Say I typed "foo". That leaves Haskell with something like the following as its "next" `IO` action:

```
if "foo" == "quit"
    then
        putStrLn "Good-bye!"
    else do
        putStrLn $ "You typed: " ++ "foo"
        main
```

Haskell is only looking at the very outermost thing, so this pretty much looks like "**if** blah blah blah blah ...". `if` isn't something that the IO-executor can do anything with, so it needs to be evaluated to see what it returns. `if` just evaluates to either the `then` or the `else` branch, but to know which decision to make Haskell is required to evaluate the condition. So we get:

```
if False
    then
        putStrLn "Good-bye!"
    else do
        putStrLn $ "You typed: " ++ "foo"
        main
```

Which allows the whole `if` to be reduced to:

```
do
    putStrLn $ "You typed: " ++ "foo"
    main
```

And again, `do` gives us an `IO` action which consists of an ordered sequence of sub-actions. So the next thing the IO-executor has to do is the `putStrLn $ "You typed: " ++ "foo"`. But that's not an `IO` action either (it's an unevaluated computation that should result in one). So we need to evalute it.

The "outermost" part of `putStrLn $ "You typed: " ++ "foo"` is actually `$`. Getting rid of the infix operator syntax so you can see it the same way the Haskell runtiem does, it would look like this:

```
($) putStrLn ((++) "You typed: " "foo")
```

But the `$` operator just defined by `($) f x = f x`, so substituting the right hand side immediately gives us:

```
putStrLn ((++) "You typed: " "foo")`
```

Now ordinarily we'd evaluate this by substituting in the definition of `putStrLn`, but it's a "magic" primitive function that isn't directly expressible in Haskell code. So it doesn't actually get evaluated like this; the outer IO-executor simply knows what to do with it. But it requires that the *argument* of `putStrLn` be fully evaluated, so we can't leave it as `(++) "You typed: " "foo"`.

There's actually a number of steps to fully evaluate that expression, going through the definition of `++` in terms of list operations, but lets just skip over that and say it evaluates to `"You typed: foo"`. So then the IO-executor can execute the `putStrLn`

(writing the text to the console), and move on to the second part of the `do` block, which is just:

```
`main`
```

Which is not something that can be immediately executed as an `IO` action (it's not built in to Haskell like `putStrLn` and `getLine` are), so we evaluate it by using the right hand side of the definition of `main` to get:

```
do
    input <- getLine
    if input == "quit"
        then
            putStrLn "Good-bye!"
        else do
            putStrLn $ "You typed: " ++ input
            main
```

And I'm sure you can see where the rest is going.

Note that I haven't said anything about any kind of stack. All of this is just building up a data structure describing the `IO` action that is `main`, so the outer driver can execute it. It's not even a particularly special data structure; from the point of view of the evaluation system it's just like any other data structure, and so there are no arbitrary limitations on its size.

In this case lazy evaluation means the generation of this data structure is interleaved with its consumption (and the generation of later parts of it can depend on what happened as a result of consuming earlier parts of it!), and so this program can run in a constant amount of space. But as noted by shachaf's comment on the question, this isn't really an optimization for removing unnecessary stack frames; it's just what happens automatically with lazy evaluation.

So I hope that was sufficiently helpful for you to see what's going on. Basically by the time Haskell gets to evaluating the recursive call to `getCommandsFromUser`, it's already done with all of the data generated within the previous iteration, and so it gets garbage collected. So you can keep running this program indefinitely without needing more than a fixed amount of memory. This is just a straightforward consequence of lazy evaluation, and isn't substantially different when `IO` is involved.

---

[1] I'm going to disclaim up front that I do not know much in detail about the actual current implementation of Haskell. I do however know general techniques for implementing lazy pure languages like Haskell. I'm also going to try to avoid diving too much into the details, and just explain how things work in an intuitive way. So this

account may well be incorrect in some of the fine details of what's actually going on inside your computer, but it should show you how these things *can* work.

[2] The language spec technically just says that evaluation should be "non-strict". The evaluation I'm going to describe, which is known as "lazy" informally, is really only one possible "non-strict" evaluation strategy, but it's what you get in practice.

[3] And the new list can in fact be left as a "pending" result of `(1 + 1) : originalList` until someone needs to know whether or not it's empty.

Share
Improve this answer
Follow

edited Dec 9, 2012 at 13:01

Vitus
11.9k ● 7 ● 38 ● 64

answered Dec 9, 2012 at 12:31

Ben
71.3k ● 21 ● 132 ● 183


11

This implementation is correct.

I don't think tail-call optimization is really the bit that makes this work efficiently. Instead, what allows it to work efficiently is, believe it or not, immutability of IO actions. Are you surprised that IO actions are immutable? I was at first! What that means is this: `getCommandsFromUser` is a recipe for "things to do"; and every time you evaluate `getCommandsFromUser`, it evaluates to *the same recipe*. (Though of course not every time you follow the recipe do you get the same result! But that's a different phase of the execution entirely.)

The upshot of this is that all evaluations of `getCommandsFromUser` can be shared -- GHC just keeps one copy of the recipe in memory, and part of that recipe includes a pointer back to the beginning of the recipe.

Share Improve this answer Follow

answered Dec 9, 2012 at 1:41

Daniel Wagner
152k ● 9 ● 227 ● 388

So then what about assignment? Pretend that there's a "let foo = bar" statement inside of the "do stuff" comment blocks (there is in my running code, and it's based on the command line input). This brings me back to my original question about stack frames. I guess I understand what you're saying conceptually, and it's not that I don't understand that it is possible for mechanics to exist to make this work; I just want to know what those mechanics are because I find it so absolutely fascinating. Why don't I overflow the stack after a million recursive calls to getCommandsFromUser? – Doug Stephen Dec 9, 2012 at 3:36

4 @DougStephen Well, part of your confusion comes from not understanding how Haskell uses the stack, too, I guess. Function calls don't add stack frames in Haskell; instead, stack frames come from nesting thunks. It might help you to take a look at this question, which I think illustrates the difference between how an imperative programmer expects the stack to be used and how GHC actually uses the stack quite nicely. – Daniel Wagner Dec 9, 2012 at 12:09

@DanielWagner that's exactly the sort of thing I was looking for. I guess I didn't phrase my question very well; it wasn't so much confusion as it was the fact that I just didn't know and was looking for a good explanation. Stack frames being tied to thunks makes sense intuitively now that you've said it. Thanks so much! – Doug Stephen Dec 9, 2012 at 17:31

I'm pretty sure that the tail call has something to do with it. If `getCommandsFromUser` wasn't the last item in the `do` notation, then the `IO` evaluation would need to keep a stack (for "pointing back") instead of pointing to the beginning only. I don't see what this has to do with immutability. – Bergi Jan 1, 2019 at 14:43

5

As I understand it you should forget about TCO: instead of asking whether a recursive call is in tail position, think in terms of *guarded recursion*. This answer I think has it right. You might also check out the post on Data and Codata from the always interesting and challenging "Neighborhood of Infinity" blog. Finally check out the Space Leak Zoo.

EDIT: I'm sorry the above doesn't address your question about monadic actions directly; I'm interested to see other answers like DanielWagner's that address the IO monad specifically.

Share
Improve this answer
Follow

edited May 23, 2017 at 10:27
Community Bot
1 • 1

answered Dec 9, 2012 at 1:41

jberryman
16.6k • 6 • 46 • 89

1

It doesn't matter that IO is involved. You can read about it in Haskell wiki:

IO inside

Or, for a more in-depth experience with Haskell's IO:

Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell

Share
Improve this answer
Follow

edited Dec 9, 2012 at 11:22

answered Dec 9, 2012 at 10:55

ljedrz
22k • 4 • 75 • 107