Tail Recursion and Streams

Abhinav Ashar
CS 61A: Structure and Interpretation of Computer Programs

January 18, 2019

1 Tail Recursion

Tail recursion is a powerful concept in computer science that helps with the space efficiency in programs. A tail recursive function is where all the recursive calls of a function are in tail contexts. As mentioned in the CSM worksheet: An ordinary recursive function is like building up a long chain of domino pieces, then knocking down the last one. A tail recursive function is like putting a domino piece up, knocking it down, putting a domino piece up again, knocking it down again, and so on. This metaphor helps explain why tail calls can be done in constant space, whereas ordinary recursive calls need space linear to the number of frames (in the metaphor, domino pieces are equivalent to frames).

a) Often, to convert recursion to tail recursion, you make sure all of your values are part of the function arguments (see tail recursion example below) rather than being applied to the recursive call of a smaller recursion function (see ordinary recursion example below). As a result, the lines that contain the recursive call should **normally** not be doing anything outside of the recursive call like +,* etc. (operations inside are okay)

b) If the recursive call is not being "returned", then there should not be any code after the recursive call because that requires keeping the frame open. Note that this does **not** mean that the recursive call can only be the last line. It can also be in the middle of the function, but after the value is returned from that recursive call, the value should immediately be returned to the previous frame rather and the program should not proceed to the following line the same frame.
TLDR: Tail recursive calls should be on lines that "return"

Not Tail Recursive:
recursiveCall(x+1, y+x)
return x + y ← extra code after recursive call = not tail recursive

Tail Recursive:
a = x + y
return recursiveCall(x+1, y+x)

c) **Why is it useful?** Tail recursion improves space efficiency. This is because the function passes in the relevant number into the argument of the recursive call rather than keeping the frame open and waiting for the value to be returned from that smaller recursive call.

d) **Ordinary Recursion Example:**
foo(5)
5 + foo(4)
5 + (4 + foo(3))
5 + (4 + (3 + foo(2)))
5 + (4 + (3 + (2 + foo(1))))
5 + (4 + (3 + (2 + 1))) → 15

Tail Recursion Example:
foo(5,0)
foo(4,5)
foo(3,9)
foo(2,12)
foo(1,14)
foo(0,15)

2 Streams

Streams are tool used for lazy evaluation in computer science. Unlike regular expressions in Scheme, streams in Scheme allows us to compute specific values up to a certain point and create a "promise" to compute future values. This allows you to create a list that can change the way it computes number halfway through a Scheme list.

a) **cons-stream** creates a lazy Scheme list where the first element is an explicit value and the second element is an expression to be computed (a promise)

b) **car** (no such thing as car-stream) evaluates the first value of the stream like normal Scheme

c) **cdr-stream** computes and returns the rest of the stream in the form of a promise. First time you call cdr-stream on a particular stream, it computes and returns the value. However, if you call cdr-stream on the same stream again, it won't recompute the value again; rather, it will return the value you previously computed (caches it)

d) **cdr** returns the second element of the stream, which is a promise (a promise that the expression will be evaluated at some point of time in the future)

e) **Forced Promise** - had forced the computation of cdr in the past using cdr stream

f) **Unforced Promise** - has not called cdr-stream yet on the stream's cdr

g) **Delay** - creates a promise and doesn't evaluate it

h) **Force** - forcibly computes a promise

i) cons-stream uses delay under the hood and cdr-stream uses force under the hood