CS 350 2024-25 Sem I Lecture 6

Satyadev Nandakumar

August 20, 2024

Outline

Outine

Laziness: introduction

Haskell uses *lazy evaluation*.
Values are not produced unless they are required by the calling function (consumer-driven).
So we can directly work with *infinite data structures* without the program hanging.
Example: `take 10 [1..]` will produce `[1,2,3,4,5,6,7,8,9,10]`

Using laziness to deal with infinite data structures

We will see some basic examples. The general style is called *stream-based programming*.

Infinite stream of ones

```
ones = 1:ones
```

Infinite stream of integers

```
adds xs ys = (head xs)+(head ys) : (adds (tail xs) (ta
ints       = 1 ++ (adds ones ints)
```

Recurrence relation for stream of integers

```
ints !! i = (ones !! (i-1)) + (ints !! (i-1))
```

Stream of factorials of integers

Recurrence relation for stream of factorials

```
ints !! i = i+1
factorials !! 0 = 1
factorials !! i = (ints !! (i-1)) * (factorials !! (i-
```

infinite stream of factorials

```
products xs ys = (head xs)*(head ys):
                 (products (tail xs) (tail ys))
factorials = [1] ++ products (tail ints) factorials
```

Outine

Tail recursion

If the last operation in a recursive function is a recursive call, then it is referred to as *tail recursion*.

Tail recursion

If the last operation in a recursive function is a recursive call, then it is referred to as *tail recursion*.

Why is it important? Recursive calls involve deep stacks. Tail recursion helps reduce the depth of these call stacks.

Example of non-tail recursive code

```
fact 1 = 1
fact n = (*) n (fact (n-1))
```

Tail recursion

If the last operation in a recursive function is a recursive call, then it is referred to as *tail recursion*.

Why is it important? Recursive calls involve deep stacks. Tail recursion helps reduce the depth of these call stacks.

Example of non-tail recursive code

```
fact 1 = 1
fact n = (*) n (fact (n-1))
```

The last operation in the inductive case is (∗), so the function is not tail-recursive

Calling frames

illustration of calling frames

```
+-----------------------------------------------------------+
|fact 4 = 4 * +-------------------------------------------+ |
|             | fact 3 = 3 * +---------------------------+| |
|             |              | fact 2 = 2 * +----------+||| |
|             |              |              | fact 1=1 |||| |
|             |              |              +----------+||| |
|             |              +--------------------------+|| |
|             +-------------------------------------------+ |
+-----------------------------------------------------------+
```

Converting to tail-recursive version

To convert to tail-recursive version, we consider the *iterative* factorial.

evaluation

```
fact 4 = 4 * 3 * 2 * 1
```

iterative factorial: pseudocode

```
factorial(n){
  product = 1
  i = n
  while  i = 1
    product = product * i
    i-1
  return product
}
```

Loop variables

We used two variables, `product` and `i`, to keep track of the computation.
`product` : partial product `n*(n-1)*...*i`
Now we write a recursive factorial where computation is updated via *extra arguments* which imitate loop variables.

iterative factorial

```
factorial n = fact_iter 1 n n
  where
    fact_iter product i n =
      if    i>1
      then  fact_iter (product*i) (i-1) n
      else  product
```

How it's done

1. `fact_iter` is the iterative version. It has extra variables, `i` and `product` which are precisely the loop variables.
2. updating variable is done while doing recursive call
3. The last operation in each inductive case is the recursive call (or returning a variable).
4. `fact_iter` needs correct initialization of `i` and `product`. Hence control access using a global function which initializes correctly, and do not give access to the user. (see `where`)

Main insight in tail recursion

If the last operation in the calling function is a recursive call, then, after returning, there is nothing more to do in the calling function.
Hence, we can remove the calling function frame immediately on recursion.
This reduces the depth of the stack!!

Outine

1. Programming technique - Laziness
2. Programming technique - tail recursion
3. Programming technique - Iteration
4. Omitted: Programming technique - continuation-passing style
5. Programming technique - memoization

Introduction to iterative style

Iterative style can be done in Haskell via:

1. "Iterative Style functional programming"
2. List comprehension.

Example problem

Find the maximum of a list (iterative style)

Pseudocode

```
max = $-\infty$
i=0
while i<length(list)
   if list[i] > max
      max = list[i]
   i=i+1
```

Main idea (iterative style)

Loop variables are those variables which are updated in the body of the loop.

Main idea (iterative style)

Loop variables are those variables which are updated in the body of the loop.

1. Write a recursive version with the "loop variables" as extra arguments to the recursive call.
2. Instead of updating variables in the loop, recurse with the updated value of the loop variables.

Example problem (continued)

- loop variables: i, max

Iterative code

```
max_iter i max_curr xs =
     if       i==(length(xs)-1)
     then     max_curr
     else
        if      list !! i > max_curr
        then    max_iter (i+1)              -- updated
                         (list !! i)        -- updated
                         xs
        else    max_iter (i+1)              -- updated
                         max_curr           -- unchanged
                         xs

maximum xs = max_iter 0 (-1) xs
```

Outine

Can we generalize tail recursion to functions with more than one recursive calls to itself? e.g. quick sort, summing elements in a tree etc.?
The 3 envelopes joke. How were the envelopes prepared?
Continuation-passing style generalizes the insight in tail recursion to functions with multiple arguments.
It also can implement generalized control-structures. (e.g. exit from the third level to the first level in a 3-level nested loop, implementing exceptions etc.)

Beyond the scope of this course.

Outine

Avoiding multiple recursive calls for the same value

The standard definion of the nth Fibonacci number:

nth Fibonacci number

```
fib 0 = 1
fib 1 = 1
fib n = (fib (n-1)) + (fib (n-2))
```

For example, `fib 4 = (fib 3) + (fib 2) = (fib 2)+(fib 1) + (fib 1) + (fib 0)`. Here, `fib(1)` is called multiple times.

Memoization: store precomputed values in a table.

Example: memoized fibonacci

We can utilize laziness of Haskell to implement a memoized version of Fibonacci.
The "lookup table" is a list of integers, where the nth element is `fib n`.

memoizing `fib` using lists

```
fib_memo   = (map fib_aux [0..] !!)
              where
                fib_aux 0 = 1
                fib_aux 1 = 1
                fib_aux n = fib_memo (n-2) + fib_memo (
```

Question:

What happens if we change the first line to `fib_memo n = (map fib_aux [0..]) !! n`? Why is the changed version slower?