COMP105 Lecture 27

Tail Recursion

Strict evaluation of factorial

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

```
fact 4
```

$\rightarrow$ `4 * fact 3`

$\rightarrow$ `4 * (3 * fact 2)`

$\rightarrow$ `4 * (3 * (2 * fact 1))`

$\rightarrow$ `4 * (3 * (2 * 1))`

$\rightarrow$ 24

Recursion and memory

Recursion seems to use a lot of **memory**

```
4 * (3 * (2 * fact 1))
```

This is the call stack

- Each recursive call adds a new value to the stack
- We can only evaluate the stack when we hit a base case

We will **run out of memory** if the call stack is too big

Tail recursion

A function is tail recursive if there is **nothing left to do** after the recursive call

```
is_even 0 = True
is_even 1 = False
is_even n = is_even (n-2)
```

```
is_even 4
→ is_even 2
→ is_even 0
→ True
```

Tail recursion

With tail recursion

- There is nothing left to do after the recursive call
- No call stack is built up
- **Less memory is used**

This is an important concept in **all** languages

- Even for imperative languages
- eg. most C compilers will optimize tail recursive calls

Writing tail recursive functions

We can make functions tail recursive

- By adding an **accumulator** as an argument
- It is initialized to some value
- Each recursive call modifies it
- Just like for `folds` and `scans`

```
fact_tail 1 acc = acc
fact_tail n acc = fact_tail (n-1) (acc * n)

factorial n = fact_tail n 1
```

`fact_tail` in strict evaluation

```
factorial 4
```

$\rightarrow$ `fact_tail 4 1`

$\rightarrow$ `fact_tail 3 4`

$\rightarrow$ `fact_tail 2 12`

$\rightarrow$ `fact_tail 1 24`

$\rightarrow$ `24`

No call stack is built up, so **no** memory problems

`fact_tail` in lazy evaluation

$\rightarrow$ `fact_tail 4 1`

$\rightarrow$ `fact_tail 3 (4*1)`

$\rightarrow$ `fact_tail 2 (3*4*1)`

$\rightarrow$ `fact_tail 1 (2*3*4*1)`

$\rightarrow$ `24`

Memory problem is just **transferred** to the accumulator

Fixing fact_tail

We can fix this with the $! (strict evaluation) operator

```
fact_tail 1 acc = acc
fact_tail n acc = fact_tail (n-1) $! (acc * n)

factorial n = fact_tail n 1
```

Now there are **no** memory problems

Left folds via tail recursion

`foldl` is naturally implemented via tail recursion

- In strict languages, `foldl` should be preferred over `foldr`

```
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

```
ghci> foldl (+) 0 [1,2,3,4,5]
15
```

Strict left folds

`foldl'` is the strict version of `foldl`

- It is in the `Data.List` package

```
foldl' f acc []     = acc
foldl' f acc (x:xs) = (foldl' f $! (f acc x)) xs
```

This will usually use **less memory** than `foldl`

Left folds and laziness

A left fold can **destroy laziness**

- We must reach the end of the list to get the accumulator

```
ghci> let f = foldr (\ x acc -> x : acc) [] [1..]
ghci> take 4 f
[1,2,3,4]
```

```
ghci> let g = foldl (\ acc x -> acc ++ [x]) [] [1..]
ghci> take 4 g
<never terminates>
```

Folds summary

`foldr`

- Should be used by default in Haskell
- Because it **preserves laziness**

`foldl`

- There is no real reason to use this in Haskell

`foldl'`

- Should be used when we know that **laziness won't help**
- eg. If we know that we will fold the entire list
- Can have performance benefits

Exercises

1. Write an efficient tail recursive implementation of sum
2. Write an efficient tail recursive implementation of length
3. Write an efficient implementation of length that uses a fold