CptS 355- Programming Language Design

Functional Programming in Haskell
Part-2

Instructor: Sakire Arslan Ay

Fall 2023

Washington State University
World Class. Face to Face.

Tail Recursion

- So far we haven't talked about the memory efficiency of recursion. For which situations do we need to improve efficiency of recursion?

- Call Stacks:
  - While a program runs, there is a stack of function calls that have started but not yet returned,
    - Calling a function `f` pushes an instance of `f` on the stack
    - When a call to `f` is finished, it is popped from the stack
  - These stack-frames (activation records) store information like the value of a local variables and "what is left to do " in the function.
  - Due to recursion, multiple stack frames may include the calls to the same function.

Program Stack

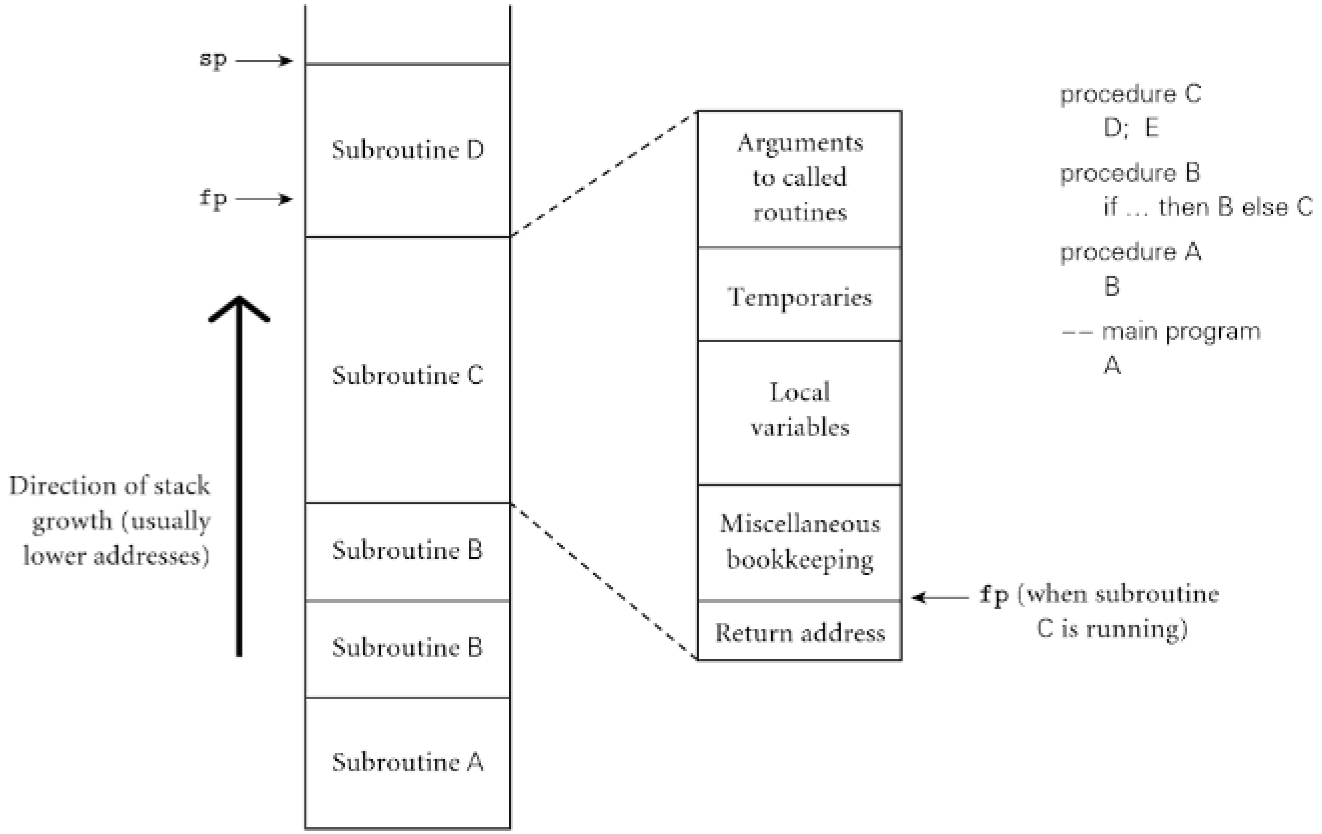

Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (sp) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (fp) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Tail Recursion

- Example: `addup` function

```
addup :: Num p => [p] -> p
addup []     = 0
addup (x:xs) = x + (addup xs)

sum1 = addup [1,2,3]          -- evaluates to 6
```

1	2	3	4
			addup[]
		addup [3]	addup [3]:3+_
	addup [2,3]	addup [2,3]: 2+_	addup [2,3]: 2+_
addup[1,2,3]	addup [1,2,3]: 1+_	addup [1,2,3]: 1+_	addup [1,2,3]: 1+_

5	6	7	8
addup[]:0			
addup [3]:3+_	addup [3]:3+0		
addup [2,3]: 2+_	addup [2,3]: 2+_	addup [2,3]: 2+3	
addup [1,2,3]: 1+_	addup [1,2,3]: 1+_	addup [1,2,3]: 1+_	addup [1,2,3]: 1+5

Tail Recursion

- Here is a second version of `addup`.

```
addup2 :: Num p => p-> [p] -> p
addup2 accum [] = accum
addup2 accum (x:xs) = (addup2 (accum + x) xs)

sum2 = addup2 0 [1,2,3]
```

1	2	3	4
			addup2 (3+3) []
		addup2 (1+2) [3]	addup2 (1+2) [3]:_
	addup2 (0+1) [2,3]	addup2 (0+1) [2,3]:_	addup2 (0+1) [2,3]:_
addup2 0 [1,2,3]	addup2 0 [1,2,3]:_	addup2 0 [1,2,3]:_	addup2 0 [1,2,3]:_

5	6	7	8
addup2 (3+3) []:6			
addup2 (1+2) [3]:_	addup2 (1+2) [3]:6		
addup2 (0+1) [2,3]:_	addup2 (0+1) [2,3]:_	addup2 (0+1) [2,3]:6	
addup2 0 [1,2,3]:_	addup2 0 [1,2,3]:_	addup2 0 [1,2,3]:_	addup2 0 [1,2,3]:6

- It is simply unnecessary to keep around a stack frame just so it can get a call's result and return it without any further evaluation.

Tail Recursion

- Such a situation is called a tail call. Haskell recognizes these tail recursive calls in the compiler and treats them differently.
  - Pop the caller before the call, allowing the callee to reuse the same stack space.
  - (Along with other optimizations) this is as efficient as a loop.

- Tail recursive call:

1	2	3	4
`addup2 0 [1,2,3]`	`addup2 (0+1) [2,3]`	`addup2(2+1) [3]`	`addup2 (3+3) []`

- We reused the stack space for the caller each time and we never used an additional stack space for the recursive calls.
- This is more efficient. Why/when does it matter?

Tail Recursion

- Let’s look at the type of `addup2`:

```
:t addup2
addup2 :: Num p => p -> [p] -> p
```

- The type is different than our original `addup` function. We will treat `addup2` as an auxiliary function and define `addup` as follows:

```
addup :: Num p => [p] -> p
addup list = let
                  addup2 accum [] = accum
                  addup2 accum (x:xs) = (addup2 (accum + x) xs)
               in addup2 0 list
```

Recursive Functions in Haskell

- Reverse:
  - What is the time complexity of `reverse'`?

```
snoc x xs = xs ++ [x]
```

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = x `snoc` (reverse' xs)
```

  - We will give a more efficient definition of `reverse`.

Recursive Functions in Haskell

- Reverse (revisited)
  - First implement reverse-append:
    - We append the first list to the second in reverse order.

```
revAppend :: [a] -> [a] -> [a]
revAppend [] acc = acc
revAppend (x:xs) acc = revAppend xs (x:acc)
```

  - How can we implement reverse using revAppend?

```
fastReverse :: [a] -> [a]
fastReverse xs = revAppend xs []
                 where
                  revAppend :: [a] -> [a] -> [a]
                  revAppend [] acc = acc
                  revAppend (x:xs) acc = revAppend xs (x:acc)
```

Recursive Functions – one more example

Calculate the lengths of the sublists in a list:

```
lengthofSublist :: [[a]] -> [Int]
lengthofSublist [] = []
lengthofSublist (x:xs) = (length x) : (lengthofSublist xs)
```

```
k = lengthofSublist [[1,2,3],[4,5],[6],[]] -- returns [3,2,1,0]
```

Haskell: Higher Order Functions

- A function is higher-order if:
  - it takes another function as an argument, or
  - it returns a function as its result.

- Functional programs make extensive use of higher-order functions to make programs smaller and more elegant.

- We use higher-order functions to encapsulate common patterns of computation.

Higher Order Functions: `map`

- Creating a new list with the same number of elements (by altering a given list) is a very common pattern that we do in programming.

- Examples: `allSquares` and `strToUpper`

```
allSquares :: Num a => [a] -> [a]
allSquares [] = []
allSquares (x : xs) = x * x : allSquares xs
```

```
strToUpper :: String -> String
strToUpper [] = []
strToUpper (chr : xs) = (Data.toUpper chr) : (strToUpper xs)
```

- This type of computation is very common. Haskell has a built-in function `map` which takes a function `op`, and a list as arguments and constructs a new list by applying the function `op` to every element of the input list.

```
map op [e1,e2,e3,e4]
⇓
[(op e1),(op e2),(op e3),(op e4)]
```

Higher Order Functions: map

Map function :

```
map' :: (a -> b) -> [a] -> [b]
map' op [] = []
map' op (x : xs) = (op x) : (map' op xs)
```

- We can redefine `allSquares` and `strToUpper` functions using `map`

```
allSquares' :: Num a => [a] -> [a]
allSquares' xL = map square xL
                 where square x = x * x
```

```
import Data.Char as Data

strToUpper' :: String -> String
strToUpper' xS = map toUpper xS
```

Anonymous Functions in Haskell

- We can also define anonymous functions (i.e., functions without names):
  - Instead of:

```
functionName a1 a2 ⋯ an = body
```

  - We write:

```
\a1 a2 ⋯ an -> body
```

  - Examples:

```
\x -> x * x        -- anonymous function calculating the square.
sq = \x -> x * x   -- can bind the function value to a variable (e.g., sq)
(\x -> x * x) 5    -- can directly call the anonymous function ; this will return 25

-- can pass the anonymous function as argument to a higher order function
sqAll = map (\x -> x * x) [1,2,3,4,5]
```

```
\x y -> (x,y) --anonymous function with two arguments
```

Higher Order Functions: `filter`

- Filter function takes a “predicate” function and a list; and returns a list consisting the elements of the original list for which the predicate function returns true for.
  - predicate function: a function that returns a Bool value

Example:

```
isNeg :: (Ord a, Num a) => a -> Bool
isNeg x = if x<0 then True else False
```

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' op [] = []
filter' op (x : xs) | (op x)     = x : (filter' op xs)
                    | otherwise  = filter' op xs
```

  - Filter examples:

```
negatives :: (Ord a, Num a) => [a] -> [a]
negatives xL = filter isNeg xL
negatives [-3,-2,-1,0,1,2,3] -- returns [-3,-2,-1]

extractDigits' :: String -> String
extractDigits' strings = filter isDigit strings
extractDigits "CptS355" -- returns 355
```

Higher Order Functions: `filter`

- `filterSmaller` – revisited

```
filterSmaller [] v = []
filterSmaller (x:xs) v | (x >= v) = x:(filterSmaller xs v)
                       | otherwise = (filterSmaller xs v)
```

- How can we re-write `filterSmaller` using `filter`?

Higher Order Functions: `foldr`

- Remember the following functions:

```
addup :: Num p => [p] -> p
addup []      = 0
addup (x:xs) = x + (addup xs)
```

```
minList :: [Int] -> Int
minList []      = maxBound
minList (x:xs) = x `min` minList xs
```

```
concatStr :: [String] -> String
concatStr [] = ""
concatStr (x:xs) = x ++ (concatStr xs)
```

- These 3 functions follow the same pattern and they are very similar. There are only small differences, which are:
  - What we did to combine the elements in the list (addition vs comparison vs concatenation)
  - What we used as the base case.

Higher Order Functions: `foldr`

- Now we will look into another higher order function that is an abstraction of this pattern and it is called the "`foldr`" function.

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' op base []     = base
foldr' op base (x:xs) = x `op` (foldr' op base xs)
```

OR

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' op base []     = base
foldr' op base (x:xs) = op x (foldr' op base xs)
```

- `foldr` folds a list together by successively applying the function `f` to the elements of the input list .

```
foldr op base [e1,e2,e3,e4]
        ⇒ op e1 (op e2 (op e3 (op e4 base)))
        OR
        ⇒ e1 `op`(e2 `op`(e3 `op` (e4 `op` base)))
```

Note: Not Haskell syntax

Higher Order Functions

- Examples:

```haskell
minList :: [Int] -> Int
minList xL = foldr min maxBound xL
```

```haskell
addup :: Num a => [a] -> a
addup xL = foldr (+) 0 xL
```

```haskell
concatStr :: [String] -> String
concatStr xL = foldr (++) "" xL
```

```haskell
reverse' :: [a] -> [a]
reverse' iL= foldr (\x xs -> xs ++ [x]) [] iL
```

```haskell
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = x `snoc` (reverse' xs)
            where snoc x xs = xs ++ [x]
```

```haskell
allEven :: [Int] -> Bool
allEven iL = foldr (\x b -> even x && b) True iL
```

```haskell
allEven :: [Int] -> Bool
allEven [] = True
allEven (x:xs) = x `allE` (allEven xs)
          where allE x b = (even x) && b
```

Higher Order Functions: `foldr` - cont.

- How does `foldr` work?
  - It traverses the list from right to left and applies the combining function.

- For example:

```
addup xL = foldr (+) 0 xL
addup [1,2,3]
```

```
1 + (foldr (+) 0 [2,3])
1 + (2 + (foldr (+) 0 [3]))
1 + (2 + (3 + (foldr (+) 0 [])))
1 + (2 + (3 + 0))
1 + (2 + 3)
1 + 5
6
```

- There is a variation of the fold function called “`foldl`” which somewhat traverses the list from left to right. i.e.,

```
((0 + 1) + 2) + 3
```

Tail recursive `foldl`

- "`foldl`" iterates over the elements from left to right.

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' op acc [] = acc
foldl' op acc (x:xs) = foldl' op (acc `op` x) xs
```

Tail-recursive

```
foldl op acc [e1,e2,e3,e4]
      ⇒ (op (op (op (op acc e1) e2) e3) e4)
       OR
      ⇒ (((acc `op` e1) `op` e2) `op` e3) `op`  e4)
```

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' op base []     = base
foldr' op base (x:xs) = x `op` (foldr' op base xs)
```

`foldr`

Examples:

- What will the `mystery` function do?

```
mystery xL = foldr (:) [] xL

mystery [1,2,3,4,5]
```

Tail recursive `foldl`

```
copyList :: [a] -> [a]
copyList xL = foldr (:) [] xL
```

OR

```
copyList :: [a] -> [a]
copyList xL = foldr (\x xs -> x:xs) [] xL
```

- How should we re-write copyList using `foldl` ?

```
copyList2 :: [a] -> [a]
copyList2 xL = reverse (foldl (\xs x -> x:xs) [] xL)
```

Tail recursive map

- `map`

```
map' :: (a -> b) -> [a] -> [b]
map' op [] = []
map' op (x : xs) = (op x) : (map' op xs)
```

- Tail recursive `map: tailmap`

```
tailmap :: (a -> b) -> [a] -> [b]
tailmap op xL = reverse (aux_map op xL [])
                where aux_map f [] acc = acc
                      aux_map f (x:xs) acc = aux_map f xs ((f x) : acc)
```

Tail recursive `filter`

- `filter`

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' op [] = []
filter' op (x : xs) | (op x)      = x : (filter' op xs)
                    | otherwise   = filter' op xs
```

- Tail recursive `filter`: `tailfilter`

```
tailfilter :: (a -> Bool) -> [a] -> [a]
tailfilter op xL = reverse (aux_filter op xL [])
                 where aux_filter f [] acc = acc
                       aux_filter f (x:xs) acc | (f x) = (aux_filter f xs (x : acc))
                                               | otherwise = (aux_filter f xs acc)
```

Examples: `map`, `fold`, `filter`

```haskell
cons0 :: Num a => [a] -> [a]
cons0 xs = 0:xs
```

- How can we use "`map`" and "`cons0`" to add `0` to each sublist in a given list?

  e.g.,

  `[[1,2],[3],[4,5],[]]    => [[0,1,2],[0,3],[0,4,5],[0]]`

```haskell
consX :: a -> [a] -> [a]
consX x xs = x:xs
```

- How can we use "`map`" and "`consX`" to add a value to each sublist in a given list?

  e.g.,

  `[["1"],["2","3"],[]] => [["0","1"],["0","2","3"],["0"]]`

Examples: `map, fold, filter`

```
max' :: Ord a => a -> a -> a
max' x y = if x < y then y else x
```

- How can we use "`foldr`" and "`max`" to find the maximum value in a nested list of integers?

  e.g.,

  `[[6,4,2],[-1,7],[1,3],[]]  => 7`

Combining Multiple Recursive Patterns

- Find the sum of sqrt of elements in a list of numbers?
  e.g., `[-1,4,-4,-3,25,16,-9] => 11.0`

```
sumOfSquareRoots :: (Ord a, Floating a) => [a] -> a
sumOfSquareRoots [] = 0
sumOfSquareRoots (x:xs)
        | x > 0     = sqrt x + sumOfSquareRoots xs
        | otherwise = sumOfSquareRoots xs
```

OR

```
sumOfSquareRoots :: (Ord a, Floating a) => [a] -> a
sumOfSquareRoots xs = sum (allSquareRoots (filterPositives xs))
  where
      allSquareRoots [] = []
      allSquareRoots (x:xs) = (sqrt x) : (allSquareRoots xs)

      filterPositives [] = []
      filterPositives (x:xs)
         | x > 0     = x : filterPositives xs
         | otherwise = filterPositives xs
```

Combining Multiple Recursive Patterns

- How can we use "`map`", and "`filter`" to find the sum of sqrt of elements in a list of integers?

```
sumOfSquareRoots :: (Ord a, Floating a) => [a] -> a

sumOfSquareRoots xs = sum (map sqrt (filter (\x -> x>0 ) xs))
```

- How can we find the sum of sqrt of elements in a nested list of integers?

  e.g. `[[25,16,-9],[0,9,-5],[]] => 12.0`

```
sumOfSqrtNested :: (Ord a, Floating a) => [[a]] -> a

sumOfSqrtNested xs = sum (map sumOfSquareRoots xs)
      where sumOfSquareRoots xL = sum (map sqrt (filter (\x -> x>0 ) xL))
```

- `sumOfSquareRoots [-1,4,-4,-3,25,16,-9]`

- `sumOfSqrtNested [[25,16,-9],[0,9,-5],[]]`

Function application with lower precedence

- Parameterized functions, such as map, filter, and foldr/foldl, are often called combinators.
  - We call the one-line definition of sumOfSquareRoots combinator-based.
  - A combinator-based expression tends to involve many parentheses.
  - To avoid this, Haskell's Prelude provides some more combinators.
  - For example:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

- `$` is right associative and has *precedence level* `0` - which is the weakest level of precedence in Haskell

```
sqrt (average 60 30)
```

```
sqrt $ average 60 30
```

- first evaluate the application of average to `60` and `30`, and then, apply `sqrt` to the result

```
sumOfSquareRoots xs = sum (map sqrt (filter (\x -> x>0 ) xs))
```

```
sumOfSquareRoots xs = sum $ map sqrt $ filter (\x -> x>0) xs
```

Function composition

```
sumOfSquareRoots xs = sum $ map sqrt $ filter (\x -> x>0) xs
```

- We would like to drop the `xs` parameter in `sumOfSquareRoots` and create a partial function.

```
sumOfSquareRoots = sum $ map sqrt $ filter (\x -> x>0)
```

This wont work (will give a compiler error). `filter`, `map`, and `sum` are nested function calls.

- Function composition allows us to apply `filter`, `map`, and `sum` as a pipeline.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

The composition `f . g` of two functions `f` and `g` produces a new function that given an argument `x` first applies `g` to `x`, and then, applies `f` to the result of that first application.

```
sumOfSquareRoots = sum . map sqrt . filter (\x -> x>0)

sumOfSquareRoots [-1,4,-4,-3,25,16,-9]   -- returns 11.0
```

`sumOfSquareRoots` as a partial function.