CSc 372

Comparative Programming Languages

21 : Haskell — Accumulative Recursion

Department of Computer Science
University of Arizona

collberg@gmail.com

Stack Recursion

- The `dots n` function returns a string consisting of `n` dots.
- The dots are "stacked" until we reach the terminating arm of the recursion. $\mathcal{O}(n)$ items are stored on the stack.

```
dots 0 = ""
dots n = "." ++ dots (n-1)
```

```
dots 3 ⇒ "." ++ dots 2 ⇒
         "." ++ ("." ++ dots 1) ⇒
         "." ++ ("." ++ ("." ++ dots 0)) ⇒
         "." ++ ("." ++ ("." ++ "")) ⇒
         "." ++ ("." ++ ".") ⇒
         "." ++ ".." ⇒ "..."
```

Accumulative Recursion

- We can sometimes get a more efficient solution by giving the function one extra argument, the accumulator, which is used to gather the final result.
- We will need to use an extra function.
- In the case of the `dots` function, the stack recursive definition is actually more efficient.

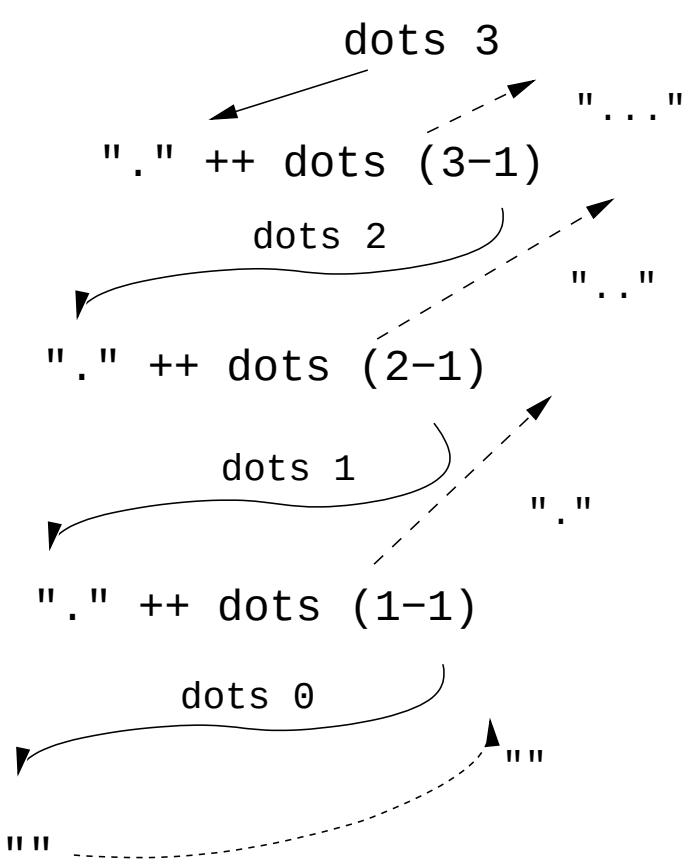
```
dots n = dots' n ""

dots' 0 acc = acc
dots' n acc = dots' (n-1) (acc ++ ".")
```

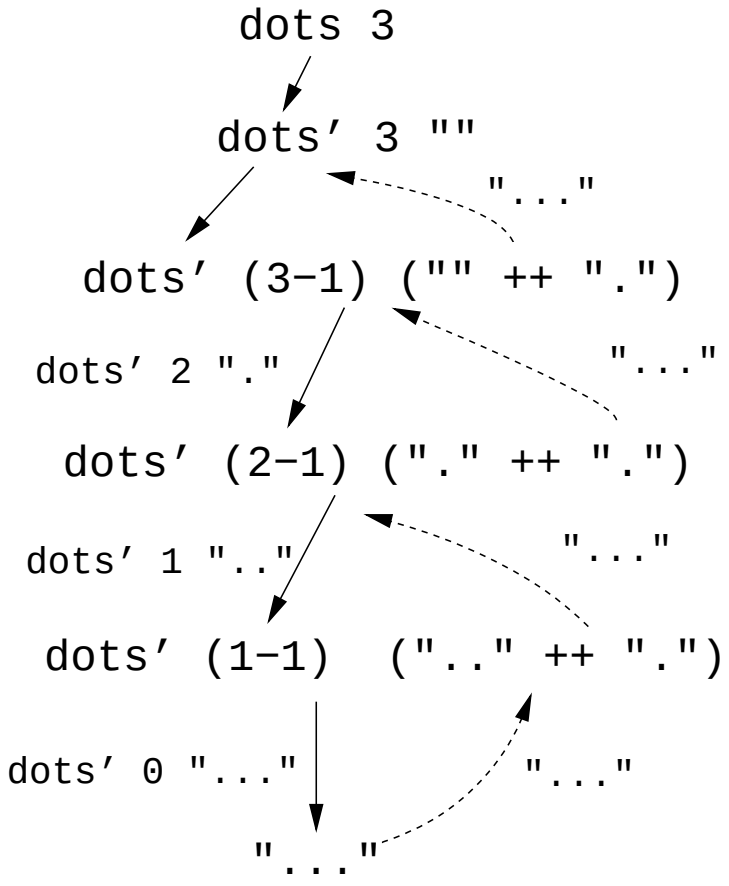
Accumulative Recursion...

```
dots n = dots' n ""
dots' 0 acc = acc
dots' n acc = dots' (n-1) (acc ++ ".")
```

```
dots 3 ⇒
   dots' 3 "" ⇒
   dots' 2 ("" ++ ".") ⇒ dots' 2 (".") ⇒
   dots' 1 ("." ++ ".") ⇒ dots' 1 ("..") ⇒
   dots' 0 (".." ++ ".") ⇒ dots' 0 ("...") ⇒
   "..."
```

Stack vs. Accumulative Recursion





Stack vs. Accumulative Recursion...

- Notice how with stack recursion we're building the result on the way back up through the layers of recursion.
- This means that for each recursive call many arguments have to be "stacked", until they can be used on the way back up.
- With accumulative recursion we're instead building the result on the way down.
- Once we're at the bottom of the recursion (when the base case has been reached) the result is ready and only needs to be passed up through the layers of recursion.

Stack Recursion Over Lists

- Stack recursive functions all look very much alike.
- All we need to do is to fill in the template below with the appropriate values and functions.
- `do` is the operation we want to apply to every element of the list.
- `combine` is the operation we want to use to combine the value computed from the head of the list, with the value produced from the tail.

Template:

```
f [ ]     = final_val
f (x:xs) = combine (do x) (f xs)
```

Stack Recursion Over Lists...

```
f [ ]     = final_val
f (x:xs) = combine (do x) (f xs)

sumlist ::  [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs

final_val=0; do x = x; combine="+"

double ::  [Int] -> [Int]
double [] = []
double (x:xs) = 2*x :  double xs

final_val=[]; do x = 2*x; combine=":"
```

Accumulative Recursion Over Lists

- `main` calls `aux`, the function that does the actual work. `main` passes along `init_val`, the value used to initiate the accumulator.
- `do` is the operation we want to apply to every element of the list.
- `combine` is the operation we want to use to combine the value computed from the head of the list with the accumulator. Template:

```
main xs = aux xs init_val

aux [] acc = acc
aux (x:xs) acc = aux xs (combine do x acc)
```

Accumulative Recursion Over Lists...

```
main xs = aux xs init_val

aux [] acc = acc
aux (x:xs) acc = aux xs (combine do x acc)
```

Example `sumlist`:

```
sumlist xs = sumlist' xs 0

sumlist' [] acc = acc
sumlist' (x:xs) acc = sumlist' xs (x + acc)

init_val=0; do x = x;
combine="+"
```

Accumulative Recursion Over Lists...

```
main xs = aux xs init_val

aux [] acc = acc
aux (x:xs) acc = aux xs (combine do x acc)
```

Example `maxlist`:

```
maxlist [] = error("...")
maxlist (x:xs) = maxlist' xs x

maxlist' [] acc = acc
maxlist' (x:xs) acc = maxlist' xs (max x a)

init_val=head xs; do x = x;
combine="max"
```

The `reverse` Function

"The reverse of an empty list is the empty list. The reverse of a list `(x:xs)` is the reverse of `xs` followed by `x`."

Examples:

```
reverse [1,2] ⇒
   reverse [2] ++ [1] ⇒
      (reverse [] ++ [2]) ++ [1] ⇒
      ([] ++ [2]) ++ [1] ⇒
   [2] ++ [1] ⇒ [2,1]
```

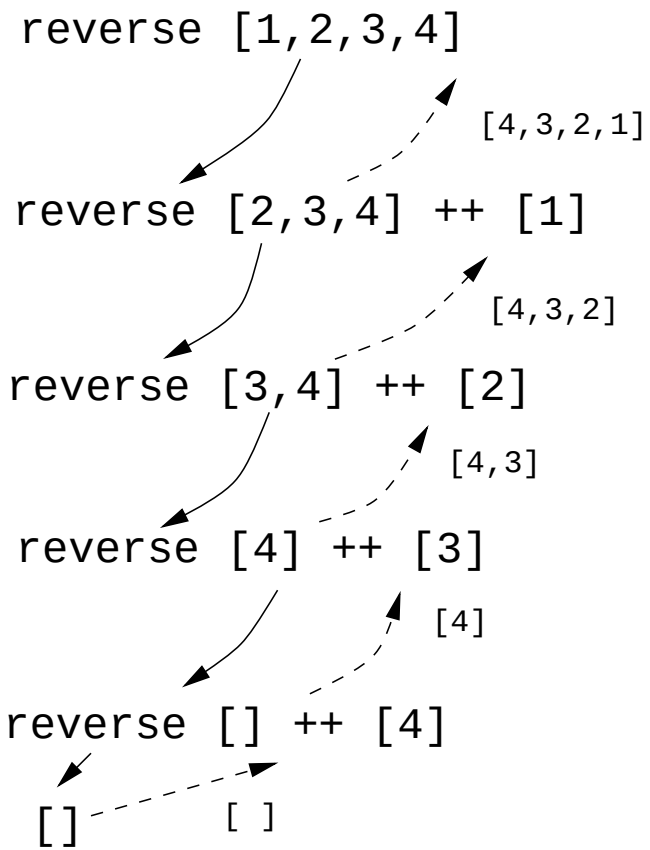
In Haskell:

```
reverse ::  [Int] -> [Int]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

The reverse Function...

```
reverse [1,2,3,4] ⇒
  reverse [2,3,4] ++ [1] ⇒
    (reverse [3,4] ++ [2]) ++ [1] ⇒
      ((reverse [4] ++ [3]) ++ [2]) ++ [1] ⇒
        (((reverse [] ++ [4]) ++ [3]) ++ [2]) ++ [1] ⇒
        ((([] ++ [4]) ++ [3]) ++ [2]) ++ [1] ⇒
      (([4] ++ [3]) ++ [2]) ++ [1] ⇒
    ([4,3] ++ [2]) ++ [1] ⇒
  [4,3,2] ++ [1] ⇒
[4,3,2,1]
```

The `reverse` Function...



- Each list append `A ++ B` takes $\mathcal{O}(\texttt{length}\ A)$ time.
- There are $\mathcal{O}(n)$ applications of `reverse`, where n is the length of the list. Each application invokes append on a list of length $\mathcal{O}(n)$. Total time $= \mathcal{O}(n^2)$.

The `reverse` Function...

- We can devise a more efficient solution by using accumulative recursion.
- At each step we tack the first element of the remaining list on to the beginning of the accumulator.

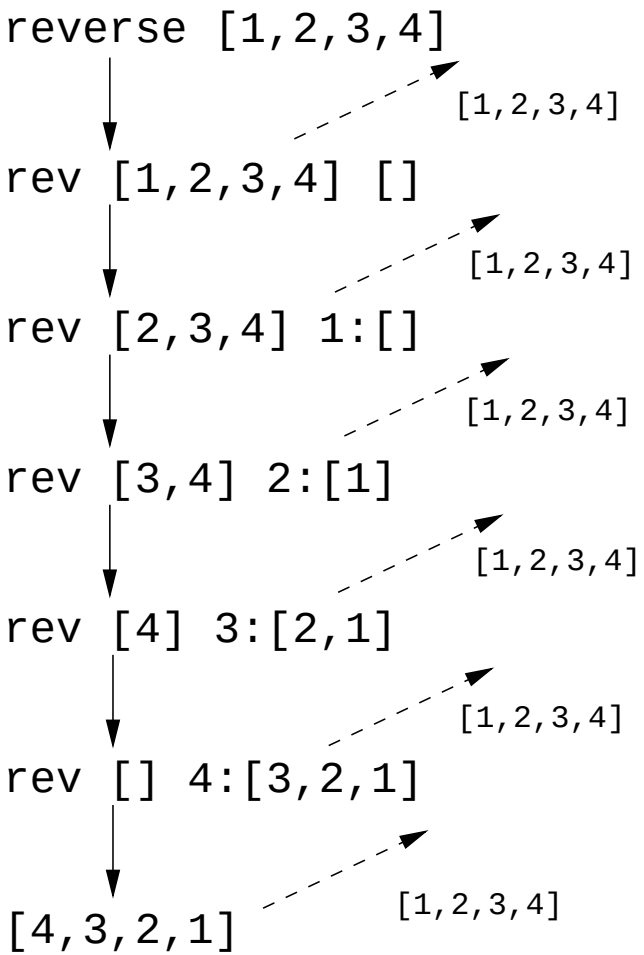
Examples:

```
reverse [1,2] ⇒
   reverse' [1,2] [] ⇒
      reverse' [2] (1:[]) ⇒
         reverse' [] (2:[1]) ⇒ [2,1]
```

In Haskell:

```
reverse xs = rev xs []
rev [] acc = acc
rev (x:xs) acc = rev xs (x:acc)
```

The `reverse` Function...



- There are $\mathcal{O}(n)$ applications of `reverse`. Each application of `rev` invokes `:` which is an $\mathcal{O}(1)$ operation. Total time $= \mathcal{O}(n)$.

Summary

- Accumulative recursion uses an extra parameter in which we collect new information as we go deeper into the recursion. The computed value is returned unchanged back up through the layers of recursion.
- Stack recursion performs much of the work on the way back up through the layers of recursion.
- Accumulative recursion is often more efficient than stack recursion.

Exercise

- `occurs x xs` returns the number of times the item `x` occurs in the list `xs`.

1. Write a stack recursive definition of `occurs`.
2. Write an accumulative recursive definition of `occurs`.
3. Try the two definitions with a large list as input. How many cells/reductions do they use?

Template:

```
occurs ::  Int -> [Int] -> Int
```

Examples:

```
? occurs 1 [3,1,4,5,1,1,2,1]
  4
```