02157 Functional Programming

Lecture 9: Tail-recursive (iterative) functions (II)

Michael R. Hansen

DTU Compute

Department of Applied Mathematics and Computer Science

Overview

Last week:

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, for example, in order
  - to avoid evaluations with a huge amount of pending operations, e.g.

    $$7+(6+(5\cdots+f\ 2\cdots))$$

  - to avoid inadequate use of `@` in recursive declarations.
- Iterative functions with accumulating parameters correspond to while-loops

Today:

- The concept continuation: Provides a general applicable approach to achieving tail-recursive functions.
  - Can also cope with complicated control structures, such as backtracking and exception handling.
  - Recommended supplementary reading: Chapters 11 and 12: *Programming Language Concepts*, Peter Sestoft, Springer 2012.

Limitation of accumulating parameters

A tail-recursive version of a recursive functions CANNOT be obtained using an accumulating parameter in all cases.

Consider for example:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a>*'a*BinTree<'a>;;

let rec count = function
                | Leaf          -> 0
                | Node(tl,n,tr) -> count tl + count tr + 1;;
```

A counting function:

```
countA: int -> BinTree<'a> -> int
```

using an accumulating parameter will not be tail-recursive due to the expression containing recursive calls on the left and right sub-trees. (Ex. 9.8)

Continuations

Continuation: A representation of the "rest" of the computation.

Example: A summation function that is NOT tail-recursive: WHY?

```
let rec sum xs = match xs with
                 | []      -> 0
                 | x::rst -> x + sum rst;;
```

The continuation-based version of `sum` has

```
k: int -> int
```

as an extra argument. Determines what happens with the result.

```
let rec sumC xs k = match xs with
                    | []      -> k 0
                    | x::rst -> sumC rst (fun v -> k(x+v))
```

This is a tail-recursive function.

- Base case: "feed" the result into the continuation `k`.
- Recursive case: The continuation after `rst` is processed is a function of the result `v` that
  - performs the addition `x+v` and
  - feed that result into the continuation `k`

An evaluation

```
    sumC [1;2;3] id
⇝   sumC [2;3] (fun v -> id(1+v))
⇝   sumC [3] (fun w -> (fun v -> id(1+v))(2+w))
⇝   sumC [] (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u))
⇝   (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u)) 0
⇝   (fun w -> (fun v -> id(1+v))(2+w)) 3
⇝   (fun v -> id(1+v)) 5
⇝   id 6
⇝   6
```

Notice:

- Closures are allocated in the heap.
- Just one stack frame is needed due to tail calls.
- Stack is traded for heap.

A more efficient representation of the continuation

Consider the following version using an accumulating parameter:

```
let rec sumA xs n = match xs with
                    | []     -> n
                    | x::rst -> sumA rst (x+n);;
```

Remember:

```
let rec sumC xs k = match xs with
                    | []     -> k 0
                    | x::rst -> sumC rst (fun v -> k(x+v))
```

What is the relationship between `sumA` and `sumC`?

$$\texttt{sumA}\ xs\ n = \texttt{sumC}\ xs\ (\texttt{fun}\ v \rightarrow n + v)$$

Proof: Structural induction over lists.

Tail recursion using accumulating parameters is not always achievable.

Structural induction over lists

The declaration

```
type 'a list =
   | Nil                      //        Nil is written []
   | Cons of 'a * 'a list     // Cons(x,xs) is written x::xs
```

denotes an inductive definition of lists (of type 'a)

- [] is a list
- if x is an element and xs is a list, then $x :: xs$ is a list
- lists can be generated by above rules only

The following structural induction rule is therefore sound:

$$\begin{array}{lll} 1. & P([]) & \text{base case} \\ 2. & \forall xs.\forall x.(P(xs) \Rightarrow P(x :: xs)) & \text{inductive step} \\ \hline & \forall xs.P(xs) & \end{array}$$

A simple verification

```
let rec sumA xs n = match xs with
                    | []     -> n
                    | x::rst -> sumA rst (x+n)

let rec sumC xs k = match xs with
                    | []     -> k 0
                    | x::rst -> sumC rst (fun v -> k(x+v))
```

Property: $\texttt{sumA}\ xs\ n = \texttt{sumC}\ xs\ (\texttt{fun}\ v \rightarrow n + v)$.

Proof: Structural induction over lists. Base case $xs = []$ is easy.

$\forall n'.\texttt{sumA}\ xs\ n' = \texttt{sumC}\ xs\ (\texttt{fun}\ v \rightarrow n' + v)$ Ind. Hyp.

The inductive step is proved as follows:

```
 sumC (x::xs) (fun v -> n+v)
=sumC xs (fun w -> ((fun v -> n+v)(x+w)))
=sumC xs (fun w -> n+(x+w))                  beta reduction
=sumC xs (fun w -> (n+x)+w)                  arithmetic
=sumA xs (n+x)                               ind. hyp. with n' ↦ n + x
=sumA (x::xs) n
```

Continuations: Another example

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
```

The continuation-based version of `bigList` has a continuation

```
k: int list -> int list
```

as argument:

```
let rec bigListC n k =
   if n=0 then k []
   else bigListC (n-1) (fun res -> k(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

- Base case: “feed” the result into the continuation `k`.
- Recursive case: The continuation after processing `(n-1)` is the function of the result `res` that
  - builds the list `1::res` and
  - feeds that list into the continuation `k`.

Observations

- `bigListC` is a tail-recursive function, and
- the calls of `k` are tail calls in the base case of `bigListC` and in the continuation: `fun res -> k(1::res)`.

The stack will hence neither grow due to the evaluation of recursive calls of `bigListC` nor due to calls of the continuations that have been built in the heap:

```
bigListC 16000000 id;;
Real: 00:00:08.586, CPU: 00:00:08.314,
GC gen0: 80, gen1: 60, gen2: 3
val it : int list = [1; 1; 1; 1; 1;...]
```

- Slower than `bigList`
- Can generate longer lists than `bigList`

Example: Tail-recursive count

```
let rec countC t k =
  match t with
  | Leaf          -> k 0
  | Node(tl,n,tr) ->
     countC tl (fun vl -> countC tr (fun vr -> k(vl+vr+1)))
val countC : BinTree<'a> -> (int -> 'b) -> 'b

countC (Node(Node(Leaf,1,Leaf),2,Node(Leaf,3,Leaf))) id;;
val it : int = 3
```

- Both calls of `countC` are tail calls
- The two calls of `k` are tail calls

Hence, the stack will not grow when evaluating `countC` ***t k***.

- `countC` can handle bigger trees than `count`
- `count` is faster

Summary and recommendations

- Have iterative functions in mind when dealing with efficiency, e.g.
  - to avoid evaluations with a huge amount of pending operations
  - to avoid inadequate use of `@` in recursive declarations.
- Memory management: stack, heap, garbage collection
- Continuations – provide a technique to turn arbitrary recursive functions into tail-recursive ones. trades stack for heap