Accumulators, tail recursion, and continuation passing style

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Recursion uses the stack: we want to avoid stack overflow for deeply recursive problems
- Recursion lends itself well to backtracking: we will cache recurring computations
- Tail recursion can be automated with higher-order functions

Introduction

Accumulators and tail recursion

- Recursion often uses the stack to store intermediate values
- Lots of intermediate values are then unwinded while popping the stack
- We can condense these values into an accumulator
- Compiler removes stack usage for such functions
- No stack overflow possible

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n-1)
```

```
let factAcc n =
  let rec aux n acc =
    match n with
    | 0 -> acc
    | _ -> aux (n-1) (n * acc)
  aux n 1
```

```
let sum l =
  let rec aux l acc =
    match l with
    | [] -> acc
    | x :: xs -> aux xs (x + acc)
  aux l 0
```

```
let mul l =
  let rec aux l acc =
    match l with
    | [] -> acc
    | x :: xs -> aux xs (x * acc)
  aux l 1
```

```
let reverse l =
  let rec aux l acc =
    match l with
    | [] -> acc
    | x :: xs -> aux xs (x :: acc)
  aux l []
```

```
let foldAcc z f l =
  let rec aux l acc =
    match l with
    | [] -> acc
    | x :: xs -> aux xs (f x acc)
  aux l z
```

```
let sum = foldAcc 0 (+)
let mul = foldAcc 1 (*)
let reverse = foldAcc [] (fun x xs -> x :: xs)
let rec fibo n =
  match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> fibo (n-1) + fibo (n-2)
```

```
let fiboAcc n =
  let rec aux n curr prev =
    match n with
    | 0 -> 0
    | 1 -> curr
    | _ ->
      aux (n-1) (curr+prev) curr
  aux n 1 0
```

Introduction

Continuation passing style

- We can automate the process of tail recursion
- We define our functions so that they take `k`, an additional parameter
- Instead of returning a result `x`, *we give it to* `k` and return `k x`
- We may create a new `k` as an anonymous function within which recursive calls happen

```
let rec factorialCPS n k =
  if n <= 1 then k n
  else
    factorialCPS (n-1) (fun fac_n_min_1 -> k (n *
        fac_n_min_1))
```

```
let rec fibonacciCPS n k =
  if n <= 1 then k n
  else
    fibonacciCPS (n-1) (fun fib_n_min_1 ->
                            fibonacciCPS (n-2) (fun
                                fib_n_min_2 ->
                                                    k
                                                       (
                                                       fib_

                                                       +

                                                       fib_
                                                       )
                                                       )
                                                       )
```

```
type Continuation<’a,’b> = (’a -> ’b) -> ’b

let (>>=) (k1 : Continuation<’a,’b>) (k2 : ’a ->
    Continuation<’a1,’b>) : Continuation<’a1,’b> =
     fun (a1_to_b) -> k1 (fun a -> k2 a a1_to_b)

let (!) (x:’a) : Continuation<’a,’b> = fun (k:’a -> ’b
    ) -> k x
```

```
let rec factCPSMonadic n =
  if n <= 1 then !n
  else
    factCPSMonadic (n-1) >>= (fun n1 -> !(n * n1))
```

```
let rec fibCPSMonadic n =
  if n <= 1 then !n
  else
    fibCPSMonadic (n-1) >>= (fun n1 ->
    fibCPSMonadic (n-2) >>= (fun n2 ->
    !(n1+n2)))
```

Introduction

Memoization

- Inputs that are recurring have the same output
- Same input yields same output only with referential transparency
- We cache the results to avoid wasteful computation

```
let fiboMem n =
  let rec aux n mem =
    match mem |> Map.tryFind n with
    | Some res -> res,mem
    | None ->
      match n with
      | 0 -> 0,mem
      | 1 -> 1,mem
      | _ ->
        let res1,mem'  = aux (n-1) mem
        let res2,mem'' = aux (n-2) (mem' |> Map.add (n
            -1) res1)
        res1 + res2,(mem'' |> Map.add (n-2) res2)
  aux n Map.empty |> fst
```

```
let store (cache:Ref<Map<’a,’b>>) k input output =
  cache := cache.Value |> Map.add input output
  k output
```

```
let memoize (cache:Ref<Map<'a,'b>>) f input k =
  match cache.Value |> Map.tryFind input with
  | Some output -> k output
  | None ->
    f cache input k
```

```
let rec fibonacciCPSMem (cache:Ref<Map<int,int>>) n k
    =
  if n <= 1 then store cache n n k
  else
    memoize cache fibonacciCPSMem (n-1)
      (fun fib_n_min_1 ->
           memoize cache fibonacciCPSMem (n-2)
             (fun fib_n_min_2 ->
                   store cache n (fib_n_min_1 +
                       fib_n_min_2) k))
```

Introduction

Course conclusions

- In the next course we will continue on building abstractions like `(>>=)` and `(!)` or `store` and `memoize`
- We will also show how to combine such abstractions together
- While building a small compiler for F# itself (a small subset)

Introduction

Conclusions and assignment

- The assignments are on Natschool
- **Restore** the games to a working state
- Hand-in a **printed** report that only contains your **sources** and the associated **documentation**

Introduction

Conclusions and assignment

- Any book on the topic will do
- I did write my own (Friendly F#) that I will be loosely following for the course, **but it is absolutely not mandatory or necessary to pass the course**

Dit is het

The best of luck, and thanks for the attention!