FPV Tutorübung

Woche 8

OCaml: Tail Recursion, Lazy Lists, Partial Application

Manuel Lerchner

14.06.2023

T01: Tail Recursion 1

a)

```
let rec f a = match a with
  | [] -> a
  | x::xs -> (x + 1)::f xs
```

b)

```
let rec g a b =
  if a = b then 0
  else if a < b then g (a + 1) b
  else g (a - 1) b
```

c)

```
let rec h a b c =
  if b then h a (not b) (c * 2)
  else if c > 1000 then a
  else h (a + 2) (not b) c * 2
```

d)

```
let rec i a = function
  | [] -> a
  | x::xs -> i (i (x,x) [x]) xs
```

1. Decide which of the following functions are implemented tail recursively:

T01: Tail Recursion 2

2. Write tail recursive versions of the following functions (without changing their types). In addition to the definition from the lecture, all functions must use constant stack space ($\mathcal{O}(1)$ in the size of its input). In particular, all the helper functions used need to be tail-recursive and use constant stack space too! If you use a library function, check that the documentation (e.g. for List) marks it as tail-recursive, or when in doubt implement a tail-recursive version yourself!

- Tipp: Use accumulator variables and helper functions

a)

```
let rec fac n =
  if n = 0 then 1
  else n * fac (n - 1)
```

b)

```
let rec remove a = function
  | [] -> []
  | x::xs -> if x = a then remove a xs else x::remove a xs
```

c)

```
let rec partition p l = match l with
  | [] -> [],[]
  | x::xs ->
    let a,b = partition p xs in
    if p x then x::a,b else a,x::b
```

T02: Lazy List Idea

```python
# Classical: Calculate all values at once and return them as a list
def powers_of_two(n):
    return [2 ** i for i in range(n)]


my_powers = powers_of_two(10)

print(my_powers)
# [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

```python
# Generator: Calculate the values on demand, one by one
def powers_of_two_generator(i):
    while True:
        yield 2 ** i
        i += 1


generator = powers_of_two_generator(0)


for i in range(10):
    print(next(generator))

# 1
# 2
# 4
# 8
# 16
# 32
# 64
# 128
# 256
# 512
# ... and so on
```

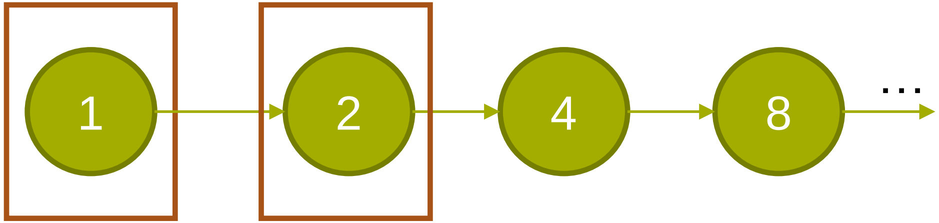
T02: Lazy List

Infinite data structures (e.g. lists) can be realized using the concept of **lazy evaluation**. Instead of constructing the entire data structure immediately, we only construct a small part and keep us a means to construct more on demand.

```
type 'a llist = Cons of 'a * (unit -> 'a
llist)
```

```
int -> int llist
let rec powers_of_2 i =
  Cons (pow 2 i, fun () → powers_of_2 (i + 1))
```

1, fun ()->**n** **n**



T02: Lazy List

1. **lnat** 0 von 1 Tests bestanden
   Implement the function `lnat : int -> int llist` that constructs the list of all natural numbers starting at the given argument.
2. **lfib** 0 von 1 Tests bestanden
   Implement the function `lfib : unit -> int llist` that constructs a list containing the Fibonacci sequence.
3. **ltake** 0 von 1 Tests bestanden
   Implement the function `ltake : int -> 'a llist -> 'a list` that returns the first n elements of the list.
4. **lfilter** 0 von 1 Tests bestanden
   Implement the function `lfilter : ('a -> bool) -> 'a llist -> 'a llist` to filter those elements from the list that do not satisfy the given predicate.

```
type 'a llist = Cons of 'a * (unit -> 'a llist)
```

```
int -> int llist
let rec powers_of_2 i =
  Cons (pow 2 i, fun () → powers_of_2 (i + 1))
```

T02: Lazy List

```typescript
type llist<T> = [T, () => llist<T>];

function fibonaci_generator(): llist<number> {
  function fib_step(a: number, b: number): llist<number> {
    return [a, () => fib_step(b, a + b)];
  }

  return fib_step(0, 1);
}

let fibonacci_numbers = fibonaci_generator();

for (let i = 0; i < 10; i++) {
  let [value, next_generator] = fibonacci_numbers;

  console.log(value);

  fibonacci_numbers = next_generator();
}

// [0 1]
//  0 [1 1]
//  0  1 [1 2]
//  0  1  1 [2 3]
//  0  1  1  2 [...]
```

```ocaml
type 'a llist = Cons of 'a * (unit -> 'a llist)
```

T03: Partial Application

Types of (apparently) n-ary functions are denoted as `arg_1 -> ... -> arg_n -> ret` in OCaml.

1. Discuss, why this notation is indeed meaningful.

2. Give the types of these expressions and discuss to what they evaluate:

```
let a (* : todo *) = (fun a b -> (+) b)

let b (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))

let c (* : todo *) = (fun a b c -> c (a + b)) 3

let d (* : todo *) = (fun a b c -> b (c a) :: [a]) "x"

let e (* : todo *) = (let x = List.map in x (<))
```

T03: Partial Application

Types of (apparently) n-ary functions are denoted as `arg_1 -> ... -> arg_n -> ret` in OCaml.

1. Discuss, why this notation is indeed meaningful.
2. Give the types of these expressions and discuss to what they evaluate:

```
let a (* : todo *) = (fun a b -> (+) b)

let b (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))

let c (* : todo *) = (fun a b c -> c (a + b)) 3

let d (* : todo *) = (fun a b c -> b (c a) :: [a]) "x"

let e (* : todo *) = (let x = List.map in x (<))
```

T03: Partial Application

```
let a (* : todo *) = (fun a b -> (+) b)
```

T03: Partial Application

```
let b (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))
```

T03: Partial Application

```
let c (* : todo *) = (fun a b c -> c (a + b)) 3
```