Functional Programming

WS 2012/13

Harald Zankl (VO)
Cezary Kaliszyk (PS) Thomas Sternagel (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 8

Overview

- Week 8 - Efficiency
  - Summary of Week 7
  - Fibonacci Numbers
  - Tupling
  - Tail Recursion

Overview

Mathematical Induction

Induction Principle

$$(P(m) \wedge \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Mathematical Induction

Induction Principle

$$(\underbrace{P(m)}_{\text{base case}} \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Mathematical Induction

Induction Principle

$$(P(m) \land \underbrace{\forall k \geq m.(P(k) \rightarrow P(k+1))}_{\text{step case}}) \rightarrow \forall n \geq m.P(n)$$

Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls

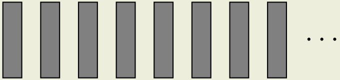
Mathematical Induction

Induction Principle

$$(P(m) \wedge \forall k \geq m.(P(k) \to P(k+1))) \to \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



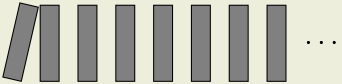
Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



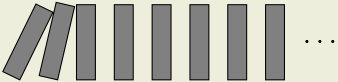
Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \to P(k+1))) \to \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



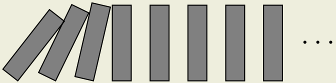
Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls

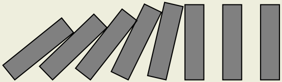
Mathematical Induction

Induction Principle

$$(P(m) \wedge \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



$\ldots$

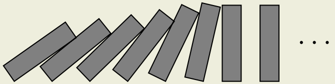
Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



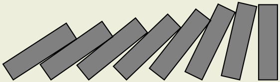
Mathematical Induction

Induction Principle

$$(P(m) \land \forall k \geq m.(P(k) \rightarrow P(k+1))) \rightarrow \forall n \geq m.P(n)$$

Example

- first domino will fall
- if a domino falls also its right neighbor falls



...

Induction on Lists

Induction Principle (without Types)

$$(P([\,]) \wedge \forall x.\forall xs.(P(xs) \to P(x :: xs))) \to \forall ls.P(ls)$$

Induction on Lists

Induction Principle (without Types)

$$(\underbrace{P(\texttt{[]})}_{\text{base case}} \wedge \forall x.\forall xs.(P(xs) \rightarrow P(x :: xs))) \rightarrow \forall ls.P(ls)$$

Induction on Lists

Induction Principle (without Types)

$$(P(\,[\,]\,) \wedge \underbrace{\forall x.\forall xs.(P(xs) \rightarrow P(x :: xs))}_{\text{step case}}) \rightarrow \forall ls.P(ls)$$

Induction on Lists

Induction Principle (without Types)

$$(P([\,]) \land \forall x.\forall xs.(P(xs) \rightarrow P(x :: xs))) \rightarrow \forall ls.P(ls)$$

Lemma (append is associative)

$$xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$$

where

```
let rec (@) xs ys = match xs with
  | []     -> ys
  | x::xs -> x :: (xs @ ys)
```

Induction on Lists

Induction Principle (without Types)

$$(P(\texttt{[]}) \wedge \forall x.\forall xs.(P(xs) \rightarrow P(x :: xs))) \rightarrow \forall ls.P(ls)$$

Lemma (append is associative)

$$xs \mathbin{\texttt{@}} (ys \mathbin{\texttt{@}} zs) = (xs \mathbin{\texttt{@}} ys) \mathbin{\texttt{@}} zs$$

where

```
let rec (@) xs ys = match xs with
  | []    -> ys
  | x::xs -> x :: (xs @ ys)
```

Proof.

Blackboard □

Structural Induction

Usage

- can be used on every variant type
- base cases correspond to non-recursive constructors
- step cases correspond to recursive constructors

Structural Induction

Usage

- can be used on every variant type
- base cases correspond to non-recursive constructors
- step cases correspond to recursive constructors

Example

Structural Induction

Usage

- can be used on every variant type
- base cases correspond to non-recursive constructors
- step cases correspond to recursive constructors

Example

- lists

Structural Induction

Usage

- can be used on every variant type
- base cases correspond to non-recursive constructors
- step cases correspond to recursive constructors

Example

- lists
- trees

Structural Induction

Usage

- can be used on every variant type
- base cases correspond to non-recursive constructors
- step cases correspond to recursive constructors

Example

- lists
- trees
- λ-terms

Structural Induction

Usage

- can be used on every variant type
- base cases correspond to non-recursive constructors
- step cases correspond to recursive constructors

Example

- lists
- trees
- λ-terms
- ...

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction,
reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, ...

Overview

- Week 8 - Efficiency
  - Summary of Week 7
  - Fibonacci Numbers
  - Tupling
  - Tail Recursion

Mathematical

Definition (n-th Fibonacci number)

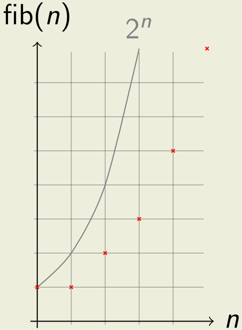
$$\mathsf{fib}(n) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } n \leq 1 \\ \mathsf{fib}(n-1) + \mathsf{fib}(n-2) & \text{otherwise} \end{cases}$$

Mathematical

Definition (n-th Fibonacci number)

$$\mathrm{fib}(n) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } n \leq 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise} \end{cases}$$

Graph



Mathematical (cont'd)

Example

1

Mathematical (cont'd)

Example

1, 1

Mathematical (cont'd)

Example

1, 1, 2

Mathematical (cont'd)

Example

1, 1, 2, 3

Mathematical (cont'd)

Example

1, 1, 2, 3, 5

Mathematical (cont'd)

Example

1, 1, 2, 3, 5, 8

Mathematical (cont'd)

Example

1, 1, 2, 3, 5, 8, 13

Mathematical (cont'd)

Example

1, 1, 2, 3, 5, 8, 13, 21

Mathematical (cont'd)

Example

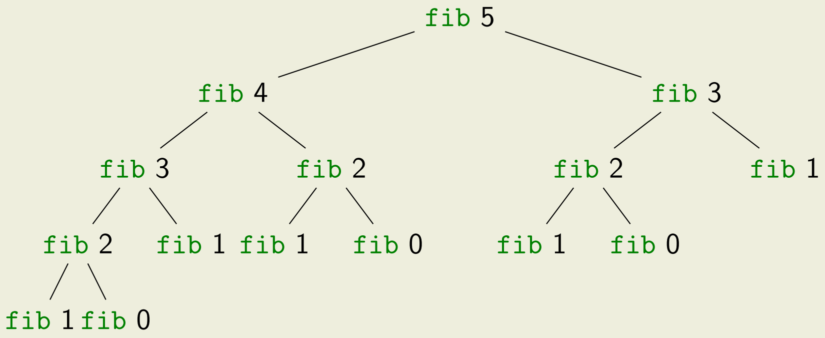
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181 ,6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073, . . .

OCaml

Definition

```
let rec fib n = if n < 2 then 1 else fib(n-1) + fib(n-2)
```

Example



Overview

Tupling

Idea

- use tuples to return more than one result
- make results available as return values instead of recomputing them

Fibonacci Numbers

Example

```
let rec fibpair n = if n < 1 then (0,1) else (
 if n = 1 then (1,1)
         else let (f1,f2) = fibpair (n-1) in (f2,f1+f2)
)
```

Fibonacci Numbers

Example

```
let rec fibpair n = if n < 1 then (0,1) else (
 if n = 1 then (1,1)
          else let (f1,f2) = fibpair (n-1) in (f2,f1+f2)
)
```

- this function is linear

Fibonacci Numbers

Example

```
let rec fibpair n = if n < 1 then (0,1) else (
 if n = 1 then (1,1)
         else let (f1,f2) = fibpair (n-1) in (f2,f1+f2)
)
```

- this function is linear

Lemma

$$\texttt{fibpair}(n+1) = (\texttt{fib}\ n, \texttt{fib}(n+1))$$

Fibonacci Numbers

Example

```
let rec fibpair n = if n < 1 then (0,1) else (
 if n = 1 then (1,1)
          else let (f1,f2) = fibpair (n-1) in (f2,f1+f2)
)
```

- this function is linear

Lemma

$$\texttt{fibpair}(n+1) = (\texttt{fib}\ n, \texttt{fib}(n+1))$$

Proof.

Blackboard □

A Second Example

Goal

compute average value of an integer list

Naive Approach

- `let average xs = IntLst.sum xs / Lst.length xs`

A Second Example

Goal

compute average value of an integer list

Naive Approach

- `let average xs = IntLst.sum xs / Lst.length xs`
- 2 traversals of `xs` are done

A Second Example

Goal

compute average value of an integer list

Naive Approach

- `let average xs = IntLst.sum xs / Lst.length xs`
- 2 traversals of `xs` are done

Combined Function

- ```
  let rec sumlen = function
    | []    -> (0,0)
    | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- `let average1 xs = let (sum,len) = sumlen xs in sum/len`
  ```

A Second Example

Goal

compute average value of an integer list

Naive Approach

- `let average xs = IntLst.sum xs / Lst.length xs`
- 2 traversals of `xs` are done

Combined Function

- ```
  let rec sumlen = function
    | []     -> (0,0)
    | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- ```
  let average1 xs = let (sum,len) = sumlen xs in sum/len
  ```
- one traversal of `xs` suffices

Overview

- Week 8 - Efficiency
  - Summary of Week 7
  - Fibonacci Numbers
  - Tupling
  - Tail Recursion

Recursion vs. Tail Recursion

Idea

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- special kind of recursion is tail recursion

Recursion vs. Tail Recursion

Idea

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- special kind of recursion is tail recursion

Recursion vs. Tail Recursion

Idea

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- special kind of recursion is tail recursion

Recursion vs. Tail Recursion

Idea

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- special kind of recursion is tail recursion

Recursion vs. Tail Recursion

Idea

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- special kind of recursion is tail recursion

Definition (Tail recursion)

a function is called tail recursive if the last action in the function body is the recursive call

Examples

Length

- ```
  let rec length = function []    -> 0
                          | _::xs -> 1 + length xs
  ```
  ```

Examples

Length

- ```
  let rec length = function []    -> 0
                          | _::xs -> 1 + length xs
  ```
- not tail recursive
  ```

Examples

Length

- ```
  let rec length = function []     -> 0
                          | _::xs -> 1 + length xs
  ```
- not tail recursive

Even/Odd

- ```
  let rec is_even = function 0 -> true
                           | 1 -> false
                           | n -> is_odd(n-1)
  and is_odd      = function 0 -> false
                           | 1 -> true
                           | n -> is_even(n-1)
  ```

Examples

Length

- ```
  let rec length = function []     -> 0
                          | _::xs -> 1 + length xs
  ```
- not tail recursive

Even/Odd

- ```
  let rec is_even = function 0 -> true
                           | 1 -> false
                           | n -> is_odd(n-1)
  and is_odd      = function 0 -> false
                           | 1 -> true
                           | n -> is_even(n-1)
  ```
- mutually recursive (btw: also tail recursive)

Parameter Accumulation

Idea

- make function tail recursive
- provide data as input instead of computing it before recursive call
- Why? (tail recursive functions can automatically be transformed into space-efficient loops)

Example (Sumlen)

- ```
  let rec sumlen = function
    | []    -> (0,0)
    | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
  ```

Example (Sumlen)

- ```
  let rec sumlen = function
    | []    -> (0,0)
    | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- not tail recursive
  ```

Example (Sumlen)

- ```
  let rec sumlen = function
   | []    -> (0,0)
   | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- not tail recursive
- ```
  let sumlen xs =
   let rec sumlen sum len = function
    | []    -> (sum,len)
    | x::xs -> sumlen (sum+x) (len+1) xs
   in
   sumlen 0 0 xs
  ```

Example (Sumlen)

- ```
  let rec sumlen = function
   | []    -> (0,0)
   | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- not tail recursive
- ```
  let sumlen xs =
   let rec sumlen sum len = function
    | []    -> (sum,len)
    | x::xs -> sumlen (sum+x) (len+1) xs
   in
   sumlen 0 0 xs
  ```
- tail recursive

Example (Sumlen)

- ```
  let rec sumlen = function
   | []    -> (0,0)
   | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- not tail recursive
- ```
  let sumlen xs =
   let rec sumlen sum len = function
    | []    -> (sum,len)
    | x::xs -> sumlen (sum+x) (len+1) xs
   in
   sumlen 0 0 xs
  ```
- tail recursive
- ```
  let sumlen xs =
   Lst.foldl (fun (sum,len) x -> (sum+x,len+1)) (0,0) xs
  ```
  ```

Example (Sumlen)

- ```
  let rec sumlen = function
   | []    -> (0,0)
   | x::xs -> let (sum,len) = sumlen xs in (sum+x,len+1)
  ```
- not tail recursive
- ```
  let sumlen xs =
   let rec sumlen sum len = function
    | []    -> (sum,len)
    | x::xs -> sumlen (sum+x) (len+1) xs
   in
   sumlen 0 0 xs
  ```
- tail recursive
- ```
  let sumlen xs =
   Lst.foldl (fun (sum,len) x -> (sum+x,len+1)) (0,0) xs
  ```
- tail recursive
  ```

Example (Range)

- ```
  let rec range m n = if m >= n then []
                                  else m::range (m+1) n
  ```
  ```

Example (Range)

- ```
  let rec range m n = if m >= n then []
                                 else m::range (m+1) n
  ```
- not tail recursive
  ```

Example (Range)

- ```
  let rec range m n = if m >= n then []
                                 else m::range (m+1) n
  ```
- not tail recursive
- ```
  let range_tl m n =
   let rec range acc m n =
    if m >= n then acc else range ((n-1)::acc) m (n-1)
   in
   range [] m n
  ```

Example (Range)

- ```
  let rec range m n = if m >= n then []
                                else m::range (m+1) n
  ```
- not tail recursive
- ```
  let range_tl m n =
   let rec range acc m n =
    if m >= n then acc else range ((n-1)::acc) m (n-1)
   in
   range [] m n
  ```
- tail recursive

Examples (Reverse)

- ```
  let rec reverse = function [] -> []
                           | x::xs -> (reverse xs) @ [x]
  ```
- not tail recursive
  ```

Examples (Reverse)

- ```
  let rec reverse = function [] -> []
                           | x::xs -> (reverse xs) @ [x]
  ```
- not tail recursive
- ```
  let rev xs =
   let rec rev acc = function []     -> acc
                            | x::xs -> rev (x::acc) xs
   in
   rev [] xs
  ```

Examples (Reverse)

- ```
  let rec reverse = function [] -> []
                      | x::xs -> (reverse xs) @ [x]
  ```
- not tail recursive
- ```
  let rev xs =
   let rec rev acc = function []     -> acc
                            | x::xs -> rev (x::acc) xs
   in
   rev [] xs
  ```
- tail recursive

Examples (Reverse)

- ```
  let rec reverse = function [] -> []
                    | x::xs -> (reverse xs) @ [x]
  ```
- not tail recursive
- ```
  let rev xs =
   let rec rev acc = function []     -> acc
                     | x::xs -> rev (x::acc) xs
   in
   rev [] xs
  ```
- tail recursive
- ```
  let rev xs = Lst.foldl (fun acc x -> x::acc) [] xs
  ```
  ```

Examples (Reverse)

- ```
  let rec reverse = function [] -> []
                    | x::xs -> (reverse xs) @ [x]
  ```
- not tail recursive
- ```
  let rev xs =
   let rec rev acc = function []     -> acc
                    | x::xs -> rev (x::acc) xs
   in
   rev [] xs
  ```
- tail recursive
- ```
  let rev xs = Lst.foldl (fun acc x -> x::acc) [] xs
  ```
- tail recursive
  ```