Functional Programming

WS 2012/13

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

Computational Logic
Institute of Computer Science
University of Innsbruck

week 7

Overview

- Week 7 - Induction
  - Summary of Week 6
  - Mathematical Induction
  - Induction Over Lists
  - Structural Induction

Overview

- Week 7 - Induction
  - Summary of Week 6
  - Mathematical Induction
  - Induction Over Lists
  - Structural Induction

Rewrite Strategies

Outermost

- choose the (leftmost) outermost redex
- redex is outermost if not subterm of different redex

Rewrite Strategies

Outermost

- choose the (leftmost) outermost redex
- redex is outermost if not subterm of different redex

Rewrite Strategies

Outermost

- choose the (leftmost) outermost redex
- redex is outermost if not subterm of different redex

Innermost

- choose the (leftmost) innermost redex
- redex is innermost if no proper subterm is redex

Rewrite Strategies

Outermost

- choose the (leftmost) outermost redex
- redex is outermost if not subterm of different redex

Innermost

- choose the (leftmost) innermost redex
- redex is innermost if no proper subterm is redex

Reduction Strategies

Call-by-name

- use outermost strategy
- stop as soon as WHNF is reached

Reduction Strategies

Call-by-name

- use outermost strategy
- stop as soon as WHNF is reached

Call-by-value

- use innermost strategy
- stop as soon as WHNF is reached

Reduction Strategies

Call-by-name

- use outermost strategy
- stop as soon as WHNF is reached

Call-by-value

- use innermost strategy
- stop as soon as WHNF is reached

WHNF (Intuition)

Thou shalt not reduce below lambda.

Evaluation Strategies

Lazy

- call-by-name + sharing
- only evaluate if necessary
- e.g. Haskell

Evaluation Strategies

Lazy

- call-by-name + sharing
- only evaluate if necessary
- e.g. Haskell

Strict/Eager

- call-by-value
- evaluate arguments before calling a function
- e.g. OCaml (also support for lazyness)

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

When?

Goal

"*prove that some property P holds for all natural numbers*"

Formally

$$\forall n.P(n) \qquad \text{(where } n \in \mathbb{N}\text{)}$$

How?

2 goals to show

1. $P(0)$
2. $\forall k.(P(k) \rightarrow P(k+1))$

How?

2 goals to show

1. $P(0)$
2. $\forall k.(P(k) \rightarrow P(k+1))$

Gives

$$(P(0) \wedge \forall k.(P(k) \rightarrow P(k+1))) \rightarrow \forall n.P(n)$$

Why Does This Work?

We have

- $P(0)$
- $\forall k.(P(k) \rightarrow P(k+1))$

Why Does This Work?

We have

- $P(0)$ “property P holds for 0”
- $\forall k.(P(k) \rightarrow P(k+1))$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \rightarrow P(1)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \to P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \to P(1)$
- have $P(1)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \rightarrow P(1)$
- have $P(1)$
- have $P(1) \rightarrow P(2)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \rightarrow P(1)$
- have $P(1)$
- have $P(1) \rightarrow P(2)$
- ...

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \rightarrow P(1)$
- have $P(1)$
- have $P(1) \rightarrow P(2)$
- ...
- have $P(n-1)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \to P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \to P(1)$
- have $P(1)$
- have $P(1) \to P(2)$
- ...
- have $P(n-1)$
- have $P(n-1) \to P(n)$

Why Does This Work?

We have

- $P(0)$ "property P holds for 0"
- $\forall k.(P(k) \rightarrow P(k+1))$ "if property P holds for arbitrary k then it also holds for $k+1$"

We want

$\forall n.P(n)$ "P holds for every n"

We get

- for the moment fix n
- have $P(0)$
- have $P(0) \rightarrow P(1)$
- have $P(1)$
- have $P(1) \rightarrow P(2)$
- ...
- have $P(n-1)$
- have $P(n-1) \rightarrow P(n)$
- hence $P(n)$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0)$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 \qquad\qquad)$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 \qquad\qquad)$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \rightarrow P(k+1)$

What is Meant by ‘Property’?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \to P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2})$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \to P(k + 1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2})$
  show: $P(k + 1)$

What is Meant by ‘Property’?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \to P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2})$
  show: $P(k+1)$

$$1 + 2 + \cdots + (k+1)$$

What is Meant by ‘Property’?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \to P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2})$
  show: $P(k+1)$

$$1 + 2 + \cdots + (k+1) = (1 + 2 + \cdots + k) + (k+1)$$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \to P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2})$
  show: $P(k+1)$

$$\begin{aligned} 1 + 2 + \cdots + (k+1) &= (1 + 2 + \cdots + k) + (k+1) \\ &\overset{\text{IH}}{=} \frac{k \cdot (k+1)}{2} + (k+1) \end{aligned}$$

What is Meant by 'Property'?

anything that depends on some variable and is either true or false can be seen as function `p : 'a -> bool`

Example

- $P(x) = (1 + 2 + \cdots + x = \frac{x \cdot (x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0 \cdot (0+1)}{2})$
- step case: $P(k) \rightarrow P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2})$
  show: $P(k+1)$

$$\begin{aligned} 1 + 2 + \cdots + (k+1) &= (1 + 2 + \cdots + k) + (k+1) \\ &\overset{\text{IH}}{=} \frac{k \cdot (k+1)}{2} + (k+1) \\ &= \frac{(k+1) \cdot (k+2)}{2} \end{aligned}$$

Remark

- of course the base case can be changed
- e.g., if base case $P(1)$, property holds for all $n \geq 1$

Overview

- Week 7 - Induction
  - Summary of Week 6
  - Mathematical Induction
  - Induction Over Lists
  - Structural Induction

Recall

Type

```
type 'a list = [] | (::) of 'a * 'a list
```

Recall

Type

```
type 'a list = [] | (::) of 'a * 'a list
```

Note

- lists are recursive structures
- base case: `[]`
- step case: *x* `::` *xs*

Induction Principle on Lists

Intuition

- to show $P(xs)$ for all lists xs
- show base case: $P(\texttt{[]})$
- show step case: $P(xs) \longrightarrow P(x :: xs)$ for arbitrary x and xs

Induction Principle on Lists

Intuition

- to show $P(xs)$ for all lists xs
- show base case: $P(\,[]\,)$
- show step case: $P(xs) \rightarrow P(x :: xs)$ for arbitrary x and xs

Formally

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

Induction Principle on Lists

Intuition

- to show $P(xs)$ for all lists xs
- show base case: $P(\texttt{[]})$
- show step case: $P(xs) \to P(x :: xs)$ for arbitrary x and xs

Formally

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

Induction Principle on Lists

Intuition

- to show $P(xs)$ for all lists xs
- show base case: $P(\texttt{[]})$
- show step case: $P(xs) \rightarrow P(x :: xs)$ for arbitrary x and xs

Formally

$$(P(\texttt{[]}) \land \forall x : \alpha.\forall xs : \alpha\ \texttt{list}.(\overbrace{P(xs)}^{\text{IH}} \rightarrow P(x :: xs))) \\ \rightarrow \forall ls : \alpha\ \texttt{list}.P(ls)$$

Induction Principle on Lists

Intuition

- to show $P(xs)$ for all lists xs
- show base case: $P(\texttt{[]})$
- show step case: $P(xs) \to P(x :: xs)$ for arbitrary x and xs

Formally

$$(P(\texttt{[]}) \land \forall x : \alpha.\forall xs : \alpha\ \texttt{list}.(\overbrace{P(xs)}^{\text{IH}} \to P(x :: xs))) \to \forall ls : \alpha\ \texttt{list}.P(ls)$$

Remarks

- $y : \beta$ reads '*y is of type* β'
- for lists, P can be seen as function `p : 'a list -> bool`

Example - Lst.append

Recall

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

Example - `Lst.append`

Recall

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

Lemma

`[]` *is right identity of* `@`, *i.e.,*

$$xs \texttt{ @ } \texttt{[]} = xs$$

Example - Lst.append

Recall

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

Lemma

[] *is* *right identity* *of* @, *i.e.*,

$$xs \; @ \; [] = xs$$

Proof.

Blackboard □

Example - Lst.length

Recall

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

Example - Lst.length

Recall

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

Lemma

sum of lengths equals length of combined list, i.e.,

$$\texttt{length}\ xs + \texttt{length}\ ys = \texttt{length}(xs\ \texttt{@}\ ys)$$

Example - Lst.length

Recall

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

Lemma

sum of lengths equals length of combined list, i.e.,

$$\texttt{length}\ xs + \texttt{length}\ ys = \texttt{length}(xs\ \texttt{@}\ ys)$$

Proof.

Blackboard □

Overview

- Week 7 - Induction
  - Summary of Week 6
  - Mathematical Induction
  - Induction Over Lists
  - Structural Induction

General Structures

Type

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

Induction Principle

General Structures

Type

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

Induction Principle

- for every non-recursive constructor there is a base case

General Structures

Type

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

Induction Principle

- for every non-recursive constructor there is a base case
  - base case: `Var x`

General Structures

Type

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

Induction Principle

- for every non-recursive constructor there is a base case
  - base case: `Var x`
- for every recursive constructor there is a step case

General Structures

Type

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

Induction Principle

- for every non-recursive constructor there is a base case
  - base case: `Var x`
- for every recursive constructor there is a step case
  - step case: `Abs`(x, t)

General Structures

Type

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

Induction Principle

- for every non-recursive constructor there is a base case
  - base case: `Var x`
- for every recursive constructor there is a step case
  - step case: `Abs`(x, t)
  - step case: `App`(s, t)

Induction Principle on General Structures

Intuition

- to show $P(s)$ for all structures s
- show base cases
- show step cases

Recall

Type

```
type 'a btree = Empty | Node of ('a btree * 'a * 'a btree)
```

Recall

Type

```
type 'a btree = Empty | Node of ('a btree * 'a * 'a btree)
```

Induction Principle

$$
\begin{gathered}
(P(\texttt{Empty}) \land \\
\forall v \quad .\forall l \qquad .\forall r \qquad . \\
((P(l) \land P(r)) \longrightarrow P(\texttt{Node}(l, v, r)))) \\
\longrightarrow \\
\forall t \qquad .P(t)
\end{gathered}
$$

Recall

Type

```
type 'a btree = Empty | Node of ('a btree * 'a btree * 'a btree)
```

Induction Principle

$$(P(\texttt{Empty}) \wedge$$
$$\forall v : \alpha . \forall l : \alpha\ \texttt{btree} . \forall r : \alpha\ \texttt{btree} .$$
$$((P(l) \wedge P(r)) \rightarrow P(\texttt{Node}(l, v, r))))$$
$$\longrightarrow$$
$$\forall t : \alpha\ \texttt{btree} . P(t)$$

Example - Trees

Definition (Perfect Binary Trees)

binary tree is perfect if all leaf nodes have same depth

```
let rec perfect = function
  | Empty        -> true
  | Node(l,_,r) -> height l = height r && perfect l
                                         && perfect r
```

Example - Trees

Definition (Perfect Binary Trees)

binary tree is perfect if all leaf nodes have same depth

```
let rec perfect = function
  | Empty        -> true
  | Node(l,_,r) -> height l = height r && perfect l
                                        && perfect r
```

Example - Trees (cont'd)

Recall

```
let rec height = function
 | Empty        -> 0
 | Node(l,_,r) -> max (height l) (height r) + 1

let rec size = function Empty       -> 0
                      | Node(l,_,r) -> size l + size r + 1
```

Example - Trees (cont'd)

Recall

```
let rec height = function
  | Empty        -> 0
  | Node(l,_,r) -> max (height l) (height r) + 1

let rec size = function Empty        -> 0
                      | Node(l,_,r) -> size l + size r + 1
```

Lemma

perfect binary tree t of height n has exactly $2^n - 1$ *nodes*

Example - Trees (cont'd)

Recall

```
let rec height = function
  | Empty         -> 0
  | Node(l,_,r) -> max (height l) (height r) + 1

let rec size = function Empty         -> 0
                      | Node(l,_,r) -> size l + size r + 1
```

Lemma

perfect binary tree t of height n has exactly $2^n - 1$ nodes

Proof.

To show: $P(t) = \big(\texttt{perfect}\ t \rightarrow (\texttt{size}\ t = 2^{(\texttt{height}\ t)} - 1)\big)$
Blackboard □