universität innsbruck

WS 2018/2019

Functional Programming

Lecture 7

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- Mathematical Induction
- Induction over Lists
- Structural Induction
- Formal Verification of Functional Programs

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

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

How is it Applied?

- mathematical induction consists of two steps:
  1. prove base case

     $$P(0)$$

     show property for 0

  2. prove step case

     $$\forall k.\, (P(k) \longrightarrow P(k+1))$$

     assume $P(k)$ (induction hypothesis), show $P(k+1)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n.\, P(n)$)

Example – $P(3)$

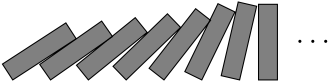
- have $P(0)$
- and $P(0) \longrightarrow P(1)$
- thus $P(1)$
- with $P(1) \longrightarrow P(2)$
- have $P(2)$
- with $P(2) \longrightarrow P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n.\, P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls



What is a “Property”?

- anything that depends on some input and is either true or false
- that is, some function `p :: a -> Bool`

Remark

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

Induction Principle

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

Example – Gauß's Formula

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

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

Algebraic Data Type of Lists

```
data [a] = [] | (:) a [a]
```

Notes

- lists are recursive structures
- non-recursive constructor (base case): `[]`
- recursive constructor (step case): x : xs

Induction Principle for Lists – Informally

- 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 for Lists – Formally

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

Remark

- for lists, P can be seen as function `p :: [a] -> Bool`

Exercise – Nil is right identity of append

- definition of append

```
   []    ++ ys  =  ys
(x:xs)  ++ ys  =  x : (xs ++ ys)
```

- prove that `[]` is right identity of ++, that is,

$$xs \mathbin{++} [] = xs$$

Exercise – Append is associative

- recall

```
  []    ++ ys  =  ys
(x:xs) ++ ys  =  x : (xs ++ ys)
```

- prove that ++ is associative, that is,

$$(xs \mathbin{+\!+} ys) \mathbin{+\!+} zs = xs \mathbin{+\!+} (ys \mathbin{+\!+} zs)$$

Exercise – Length and append

- definition

```
length []      =  0
length (_:xs)  =  1 + length xs
```

- prove that length of combined list is sum of lengths, that is,

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

Example – Terms

```
type Id = String
data Term = Var Id
          | App Term Term
          | Abs Id Term
```

General Structures – Induction Principle

- for every non-recursive constructor, show base case
  - base case: $P(\texttt{Var x})$
- for every recursive constructor, show step case
  - step case 1: $(P(\texttt{s}) \wedge P(\texttt{t})) \longrightarrow P(\texttt{App s t})$
  - step case 2: $P(\texttt{t}) \longrightarrow P(\texttt{Abs x t})$

Example – Binary Trees

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```

Induction Principle for Binary Trees

$$(P(\texttt{Empty}) \land \forall x. \forall l. \forall r. ((P(l) \land P(r)) \longrightarrow P(\texttt{Node}\ x\ l\ r))) \longrightarrow \forall t. P(t)$$

Exercise – Perfect Binary Trees

- a binary tree is perfect if all leaf nodes have same depth

```
perfect Empty        = True
perfect (Node _ l r) =
  height l == height r && perfect l && perfect r

height Empty        = 0
height (Node _ l r) =
  max (height l) (height r) + 1

size Empty        = 0
size (Node _ l r) = size l + size r + 1
```

- lemma: a perfect binary tree t of height n has exactly $2^n - 1$ nodes, that is,

$$P(t) = \left(\texttt{perfect}\ t \longrightarrow \texttt{size}\ t = 2^{\texttt{height}\ t} - 1\right)$$

Isabelle/HOL in a Nutshell

Obvious question:

- What is Isabelle?

Common answer:

- An LCF-style proof assistant.

Typical follow-up questions:

- What is a proof assistant?
- What does LCF-style mean?
- ...

What is a Proof Assistant?

- combination of automated theorem prover (ATP) and proof checker
- some subproofs are found automatically
- others are user-supplied but checked rigorously

Example

- automatic methods: logical reasoning (`blast`), equational reasoning (`simp`), combination of former (`auto`), ...
- manual steps: induction (`induct`), case analysis (`cases`), ...

What does LCF-style mean?

- theorems represented by abstract data type (`thm`)
- set of (basic) logical inferences provided as interface (trusted kernel)
- strong typing guarantees that there is no other way to create theorems (values of type `thm`)

Example

certified term

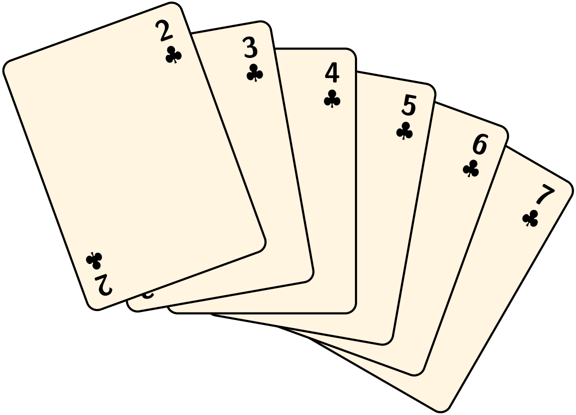
- functions `assume : cterm -> thm` and `implies_elim : thm -> thm -> thm`
- implement inference rules

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash A \Longrightarrow B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

Higher-Order Logic

- HOL = Functional Programming + Logic
- data types (`datatype`)
- recursive functions (`fun`)
- logical operators ($\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$, $\ldots$)

Example – Insertion Sort



A Functional Implementation

- inserting an element into a sorted list

```
insert x [] = [x]
insert x (y:ys) =
  if x <= y then x : y : ys
  else y : insert x ys
```

- sorting by repeatedly inserting elements into the empty list

```
insertionSort = foldr insert []
```

Exercise – Insertion sort is a valid sorting algorithm

- prove that result after applying insertion sort is sorted
- prove that all values occur exactly the same number of times in input and output
- see `Insertion_Sort.thy`

Homework (for December 14th)

1. Read the lecture notes on reasoning about functional programs.
2. Prove $\texttt{map}\ f\ (\texttt{map}\ g\ xs) = \texttt{map}\ (f \circ g)\ xs$ for

```
map f [] = []
map f (x:xs) = f x : map f xs
```

3. Prove $\texttt{filter}\ p\ (\texttt{map}\ f\ xs) = \texttt{map}\ f\ (\texttt{filter}\ (p \circ f)\ xs)$ for

```
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

4. Prove $\texttt{map}\ f\ (xs \mathbin{+\!\!+} ys) = \texttt{map}\ f\ xs \mathbin{+\!\!+} \texttt{map}\ f\ ys$ for

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

5. Prove $\forall xs.\, \texttt{take}\ n\ (\texttt{map}\ f\ xs) = \texttt{map}\ f\ (\texttt{take}\ n\ xs)$ for

```
take n (x:xs) | n > 0 = x : take (n - 1) xs
take _ _ = []
```

Homework (for December 14th, continued)

6. Prove $\forall xs.\, \texttt{take}\ n\ xs +\!\!+ \texttt{drop}\ n\ xs = xs$ for

```
drop n (_:xs) | n > 0 = drop (n - 1) xs
drop _ xs = xs
```

Alternatively

Choose two of the previous exercises and prove them with Isabelle/HOL using the custom type

```
datatype 'a lst = NIL | CONS 'a "'a lst"
```

and your own implementations of the relevant functions among

```
map :: "('a => 'b) => 'a lst => 'b lst"
filter :: "('a => bool) => 'a lst => 'a lst"
app :: "'a lst => 'a lst => 'a lst"
take :: "nat => 'a lst => 'a lst"
drop :: "nat => 'a lst => 'a lst"
```