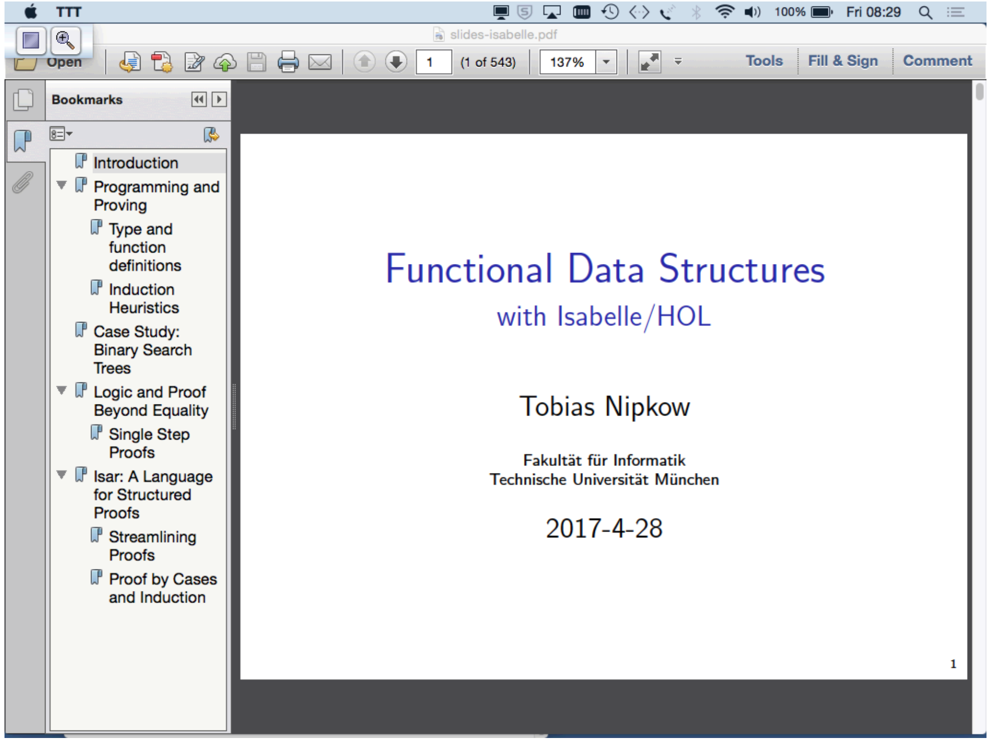
Script **generated by TTT**

Title: FDS (28.04.2017)

Date: Fri Apr 28 08:29:57 CEST 2017

Duration: 92:30 min

Pages: 109

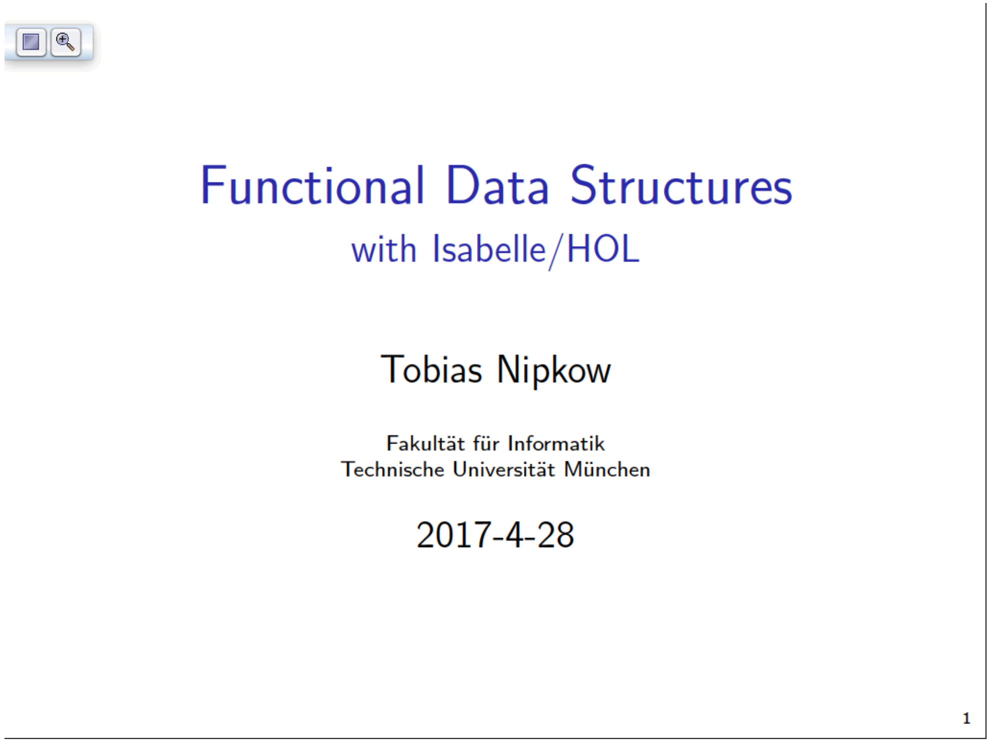


Functional Data Structures
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2017-4-28

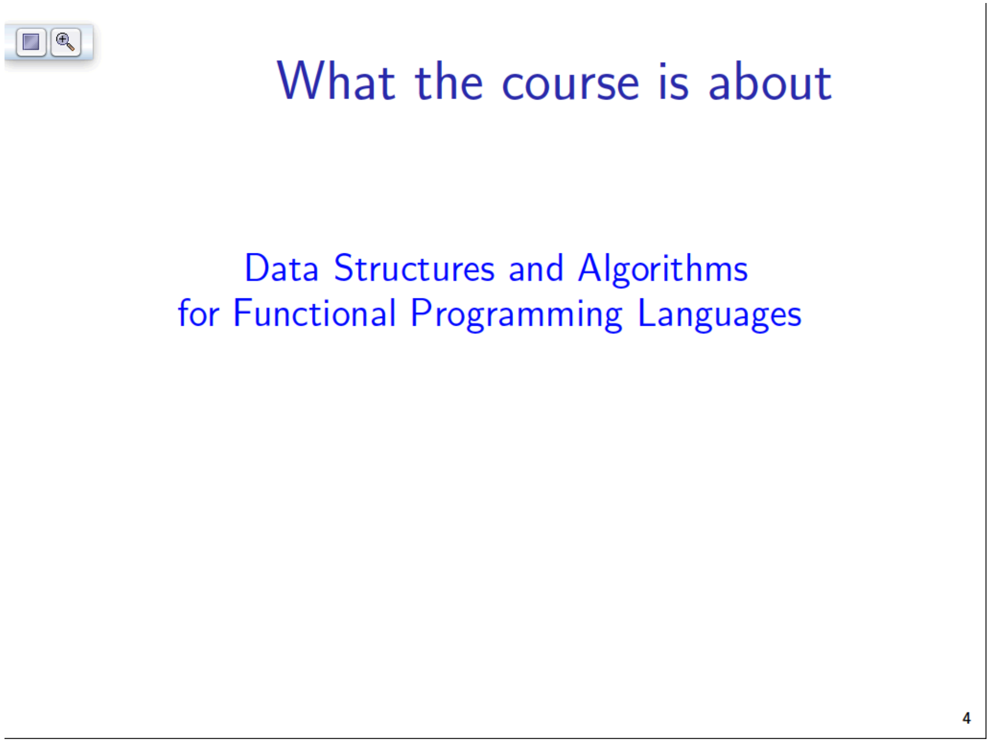


Functional Data Structures
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2017-4-28



What the course is about

Data Structures and Algorithms
for Functional Programming Languages

What the course is about

Data Structures and Algorithms
for Functional Programming Languages

The code is not enough!

Formal Correctness and Complexity Proofs
with the Proof Assistant *Isabelle*

Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step

Terminology

Formal = machine-checked
Verification = formal correctness proof

Two landmark verifications

C compiler

Two landmark verifications

C compiler
Competitive with `gcc -O1`

Xavier Leroy
INRIA Paris
using Coq

Two landmark verifications

C compiler
Competitive with `gcc -O1`

Xavier Leroy
INRIA Paris
using Coq

Operating system
microkernel (L4)

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

Overview of course

- Week 1–5: Introduction to Isabelle
- Rest of semester: Search trees, priority queues, etc and their (amortized) complexity

---

What we expect from you

Functional programming experience with an ML/Haskell-like language

What we expect from you

Functional programming experience with an ML/Haskell-like language

First course in data structures and algorithms

What we expect from you

Functional programming experience with an ML/Haskell-like language

First course in data structures and algorithms

First course in discrete mathematics

What we expect from you

Functional programming experience with an ML/Haskell-like language

First course in data structures and algorithms

First course in discrete mathematics

You will not survive this course without doing the time-consuming homework

---

Part I

Isabelle

Quiz

Which of the following formulas have the same meaning?

1. $A \Longrightarrow (B \Longrightarrow C)$
2. $(A \Longrightarrow B) \Longrightarrow C$
3. $(A \wedge B) \Longrightarrow C$

Notation

Implication associates to the right:

$$A \Longrightarrow B \Longrightarrow C \quad \text{means} \quad A \Longrightarrow (B \Longrightarrow C)$$

Notation

Implication associates to the right:

$$A \Longrightarrow B \Longrightarrow C \quad \text{means} \quad A \Longrightarrow (B \Longrightarrow C)$$

Similarly for other arrows: $\Rightarrow$, $\longrightarrow$

$$\frac{A_1 \quad \dots \quad A_n}{B} \quad \text{means} \quad A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

---

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:

- For the moment: only $term = term$

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:

- For the moment: only $term = term$, e.g. $1 + 2 = 4$
- Later: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\ldots$

---

1. Overview of Isabelle/HOL
   - Types and terms
   - Interface
   - By example: types $bool$, nat and $list$
   - Numeric Types
   - Summary

Types

Basic syntax:

$$\begin{array}{lcll} \tau & ::= & (\tau) & \\ & | & bool \mid nat \mid int \mid \ldots & \text{base types} \end{array}$$

Types

Basic syntax:

$$\begin{array}{lcll} \tau & ::= & (\tau) & \\ & | & bool \mid nat \mid int \mid \ldots & \text{base types} \\ & | & {'a} \mid {'b} \mid \ldots & \text{type variables} \end{array}$$

Types

Basic syntax:

$$\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots & \text{base types} \\
 & | & 'a \mid 'b \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{functions}
\end{array}$$

Types

Basic syntax:

$$\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots & \text{base types} \\
 & | & 'a \mid 'b \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } *\text{)}
\end{array}$$

Types

Basic syntax:

$$\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots & \text{base types} \\
 & | & 'a \mid 'b \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } *\text{)} \\
 & | & \tau\ \mathit{list} & \text{lists}
\end{array}$$

Types

Basic syntax:

$$\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots & \text{base types} \\
 & | & 'a \mid 'b \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } *\text{)} \\
 & | & \tau\ \mathit{list} & \text{lists} \\
 & | & \tau\ \mathit{set} & \text{sets}
\end{array}$$

Types

Basic syntax:

$$\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & \mid & bool \mid nat \mid int \mid \ldots & \text{base types} \\
 & \mid & 'a \mid 'b \mid \ldots & \text{type variables} \\
 & \mid & \tau \Rightarrow \tau & \text{functions} \\
 & \mid & \tau \times \tau & \text{pairs (ascii: } *\text{)} \\
 & \mid & \tau\ list & \text{lists} \\
 & \mid & \tau\ set & \text{sets} \\
 & \mid & \ldots & \text{user-defined types}
\end{array}$$

Terms

Basic syntax:

$$\begin{array}{lcll}
t & ::= & (t) & \\
 & \mid & a & \text{constant or variable (identifier)}
\end{array}$$

Terms

Basic syntax:

$$\begin{array}{lcll}
t & ::= & (t) & \\
 & \mid & a & \text{constant or variable (identifier)} \\
 & \mid & t\ t & \text{function application} \\
 & \mid & \lambda x.\ t & \text{function abstraction} \\
 & \mid & \ldots & \text{lots of syntactic sugar}
\end{array}$$

Terms

Basic syntax:

$$\begin{array}{lcll}
t & ::= & (t) & \\
 & \mid & a & \text{constant or variable (identifier)} \\
 & \mid & t\ t & \text{function application} \\
 & \mid & \lambda x.\ t & \text{function abstraction} \\
 & \mid & \ldots & \text{lots of syntactic sugar}
\end{array}$$

λ-calculus

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:
$t\ ::\ \tau$ means "t is a well-typed term of type τ".

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:
$t\ ::\ \tau$ means "t is a well-typed term of type τ".

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\ u :: \tau_2}$$

Type inference

Isabelle automatically computes the type of each variable in a term.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

User can help with *type annotations* inside the term.
Example: $f\ (x{::}nat)$

Currying

Thou shalt Curry your functions

- Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Predefined syntactic sugar

- *Infix:* $+$, $-$, $*$, $\#$, $@$, ...

Predefined syntactic sugar

- *Infix:* $+$, $-$, $*$, $\#$, $@$, ...
- *Mixfix:* $\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_$, $\mathit{case}\ \_\ \mathit{of}$, ...

Predefined syntactic sugar

- *Infix:* $+, -, *, \#, @, \ldots$
- *Mixfix:* $\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_$, $\mathit{case}\ \_\ \mathit{of}$, $\ldots$

Prefix binds more strongly than infix:

$$! \quad f\,x + y \ \equiv\ (f\,x) + y \ \not\equiv\ f\,(x + y) \quad !$$

Predefined syntactic sugar

- *Infix:* $+, -, *, \#, @, \ldots$
- *Mixfix:* $\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_$, $\mathit{case}\ \_\ \mathit{of}$, $\ldots$

Prefix binds more strongly than infix:

$$! \quad f\,x + y \ \equiv\ (f\,x) + y \ \not\equiv\ f\,(x + y) \quad !$$

Enclose if and case in parentheses:

$$! \quad (\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_) \quad !$$

Predefined syntactic sugar

- *Infix:* $+, -, *, \#, @, \ldots$
- *Mixfix:* $\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_$, $\mathit{case}\ \_\ \mathit{of}$, $\ldots$

Prefix binds more strongly than infix:

$$! \quad f\,x + y \ \equiv\ (f\,x) + y \ \not\equiv\ f\,(x + y) \quad !$$

Enclose if and case in parentheses:

$$! \quad (\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_) \quad !$$

Theory = Isabelle Module

Theory = Isabelle Module

Syntax: `theory` $MyTh$
`imports` $T_1 \ldots T_n$
`begin`
(definitions, theorems, proofs, ...)*
`end`

Theory = Isabelle Module

Syntax: `theory` $MyTh$
`imports` $T_1 \ldots T_n$
`begin`
(definitions, theorems, proofs, ...)*
`end`

$MyTh$: name of theory. Must live in file $MyTh$`.thy`
T_i: names of *imported* theories. Import transitive.

Usually: `imports Main`

Theory = Isabelle Module

Syntax: `theory` $MyTh$
`imports` $T_1 \ldots T_n$
`begin`
(definitions, theorems, proofs, ...)*
`end`

$MyTh$: name of theory. Must live in file $MyTh$`.thy`
T_i: names of *imported* theories. Import transitive.

Usually: `imports Main`

Concrete syntax

In `.thy` files:
Types, terms and formulas need to be inclosed in `"`

Theory = Isabelle Module

Syntax: `theory` $MyTh$
`imports` $T_1 \dots T_n$
`begin`
(definitions, theorems, proofs, ...)*
`end`

$MyTh$: name of theory. Must live in file $MyTh$`.thy`
T_i: names of *imported* theories. Import transitive.

Usually: `imports Main`

Concrete syntax

In `.thy` files:
Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides

---

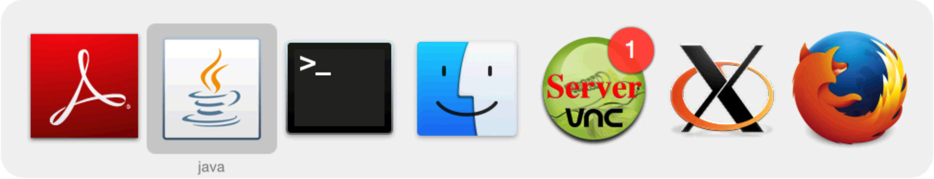
isabelle jedit

- Based on *jEdit* editor
- Processes Isabelle text automatically when editing `.thy` files

---

`Overview_Demo.thy`

`Overview_Demo.thy`



Type *bool*

datatype $bool \; = \; True \; | \; False$

Type *bool*

datatype $bool \; = \; True \; | \; False$

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots :: bool \Rightarrow bool \Rightarrow bool$

Type *bool*

datatype $bool \; = \; True \; | \; False$

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots :: bool \Rightarrow bool \Rightarrow bool$

A *formula* is a term of type *bool*

Type $bool$

datatype $bool = True \mid False$

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots :: bool \Rightarrow bool \Rightarrow bool$

A *formula* is a term of type $bool$

if-and-only-if: $=$

Type nat

datatype $nat = 0 \mid Suc\ nat$

Type nat

datatype $nat = 0 \mid Suc\ nat$

Values of type nat: $0,\ Suc\ 0,\ Suc(Suc\ 0), \ldots$

Type nat

datatype $nat = 0 \mid Suc\ nat$

Values of type nat: $0,\ Suc\ 0,\ Suc(Suc\ 0), \ldots$

Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

Values of type *nat*: $0,\ Suc\ 0,\ Suc(Suc\ 0), \ldots$

Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,\ldots :: 'a, \quad + :: \ 'a \Rightarrow 'a \Rightarrow 'a$

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

Values of type *nat*: $0,\ Suc\ 0,\ Suc(Suc\ 0), \ldots$

Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,\ldots :: 'a, \quad + :: \ 'a \Rightarrow 'a \Rightarrow 'a$

You need type annotations: $1 :: nat,\ x + (y{::}nat)$

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

Values of type *nat*: $0,\ Suc\ 0,\ Suc(Suc\ 0), \ldots$

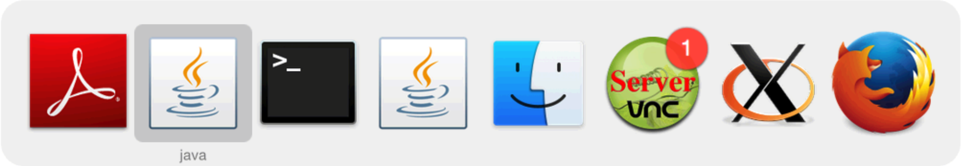
Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,\ldots :: 'a, \quad + :: \ 'a \Rightarrow 'a \Rightarrow 'a$

You need type annotations: $1 :: nat,\ x + (y{::}nat)$
unless the context is unambiguous: $Suc\ z$

`Nat_Demo.thy`

`Nat_Demo.thy`



An informal proof

Lemma $add\ m\ 0 = m$

Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).

An informal proof

Lemma $add\ m\ 0 = m$

Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).
  We need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:
  $$add\ (Suc\ m)\ 0 \;=\; Suc\ (add\ m\ 0) \quad \text{by def. of } add$$

An informal proof

Lemma $add\ m\ 0 = m$

Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).
  We need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:
  $$\begin{aligned} add\ (Suc\ m)\ 0 &= Suc\ (add\ m\ 0) && \text{by def. of } add \\ &= Suc\ m && \text{by IH} \end{aligned}$$

Type $'a\ list$

Lists of elements of type $'a$

Type $'a\ list$

Lists of elements of type $'a$

Type $'a\ list$

Lists of elements of type $'a$

datatype $'a\ list \;=\; Nil \;\mid\; Cons\ 'a\ \ ('a\ list)$

Type $'a\ list$

Lists of elements of type $'a$

datatype $'a\ list \;=\; Nil \;\mid\; Cons\ 'a\ \ ('a\ list)$

Some lists: Nil,

Type $'a\ list$

Lists of elements of type $'a$

datatype $'a\ list\ =\ Nil\ \mid\ Cons\ 'a\ ('a\ list)$

Some lists: Nil, $Cons\ 1\ Nil$, $Cons\ 1\ (Cons\ 2\ Nil)$, $\ldots$

Syntactic sugar:

- $[] = Nil$: empty list
- $x\ \#\ xs = Cons\ x\ xs$:
  list with first element x (*"head"*) and rest xs (*"tail"*)
- $[x_1, \ldots, x_n] = x_1\ \#\ \ldots\ x_n\ \#\ []$

Structural Induction for lists

To prove that $P(xs)$ for all lists xs, prove

- $P([])$ and
- for arbitrary but fixed x and xs,
  $P(xs)$ implies $P(x\#xs)$.

Structural Induction for lists

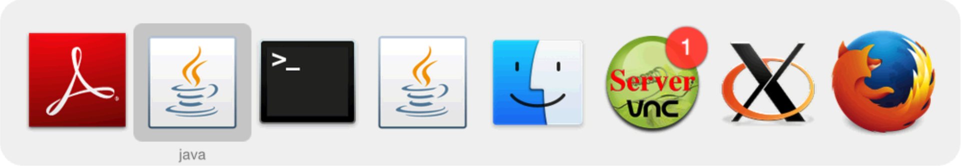
To prove that $P(xs)$ for all lists xs, prove

- $P([])$ and
- for arbitrary but fixed x and xs,
  $P(xs)$ implies $P(x\#xs)$.

$$\frac{P([]) \qquad \bigwedge x\ xs.\ P(xs) \Longrightarrow P(x\#xs)}{P(xs)}$$

`List_Demo.thy`

`List_Demo.thy`



Large library: `HOL/List.thy`

Included in `Main`.

Don't reinvent, reuse!

---

1. Overview of Isabelle/HOL
   - Types and terms
   - Interface
   - By example: types *bool*, *nat* and *list*
   - Numeric Types
   - Summary

Numeric types: *nat*, *int*, *real*

Need conversion functions (inclusions):

$$\begin{aligned} int &:: nat \Rightarrow int \\ real &:: nat \Rightarrow real \\ real\_of\_int &:: int \Rightarrow real \end{aligned}$$

Numeric types: nat, int, $real$

Need conversion functions (inclusions):

$$\begin{aligned} int &:: nat \Rightarrow int \\ real &:: nat \Rightarrow real \\ real\_of\_int &:: int \Rightarrow real \end{aligned}$$

If you need type $real$,
import theory $Complex\_Main$ instead of $Main$

Numeric types: nat, int, $real$

Need conversion functions (inclusions):

$$\begin{aligned} int &:: nat \Rightarrow int \\ real &:: nat \Rightarrow real \\ real\_of\_int &:: int \Rightarrow real \end{aligned}$$

If you need type $real$,
import theory $Complex\_Main$ instead of $Main$

Numeric types: nat, int, $real$

Isabelle inserts conversion functions automatically

Numeric types: nat, int, $real$

Isabelle inserts conversion functions automatically
(with theory $Complex\_Main$)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

Numeric types: nat, int, real

Isabelle inserts conversion functions automatically
(with theory $\mathit{Complex\_Main}$)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

Examples

$(i::\mathit{int}) + (n::\mathit{nat})$

Numeric types: nat, int, real

Isabelle inserts conversion functions automatically
(with theory $\mathit{Complex\_Main}$)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

Examples

$(i::\mathit{int}) + (n::\mathit{nat}) \rightsquigarrow i + \mathit{int}\ n$

Numeric types: nat, int, real

Isabelle inserts conversion functions automatically
(with theory $\mathit{Complex\_Main}$)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

Examples

$(i::\mathit{int}) + (n::\mathit{nat}) \rightsquigarrow i + \mathit{int}\ n$

$((n::\mathit{nat}) + n) :: \mathit{real}$

Numeric types: nat, int, real

Isabelle inserts conversion functions automatically
(with theory $\mathit{Complex\_Main}$)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

Examples

$(i::\mathit{int}) + (n::\mathit{nat}) \rightsquigarrow i + \mathit{int}\ n$

$((n::\mathit{nat}) + n) :: \mathit{real} \rightsquigarrow \mathit{real}(n+n)$, $\mathit{real}\ n + \mathit{real}\ n$

Numeric types: *nat*, *int*, *real*

Coercion in the other direction:

$$nat \quad :: \quad int \Rightarrow nat$$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on *nat* and *int*:
  $div,\ mod :: 'a \Rightarrow 'a \Rightarrow 'a$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on *nat* and *int*:
  $div,\ mod :: 'a \Rightarrow 'a \Rightarrow 'a$
- Division on *real*: $/ :: 'a \Rightarrow 'a \Rightarrow 'a$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on *nat* and *int*:
  $div,\ mod :: 'a \Rightarrow 'a \Rightarrow 'a$
- Division on *real*: $/ :: 'a \Rightarrow 'a \Rightarrow 'a$
- Exponentiation with *nat*: $\hat{}\ :: 'a \Rightarrow nat \Rightarrow 'a$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on nat and int:
  $div, mod :: 'a \Rightarrow 'a \Rightarrow 'a$
- Division on $real$: $/ :: 'a \Rightarrow 'a \Rightarrow 'a$
- Exponentiation with nat: $\hat{} :: 'a \Rightarrow nat \Rightarrow 'a$
- Exponentiation with $real$: $powr :: 'a \Rightarrow 'a \Rightarrow 'a$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on nat and int:
  $div, mod :: 'a \Rightarrow 'a \Rightarrow 'a$
- Division on $real$: $/ :: 'a \Rightarrow 'a \Rightarrow 'a$
- Exponentiation with nat: $\hat{} :: 'a \Rightarrow nat \Rightarrow 'a$
- Exponentiation with $real$: $powr :: 'a \Rightarrow 'a \Rightarrow 'a$

- Absolute value: $abs :: 'a \Rightarrow 'a$

---

1. Overview of Isabelle/HOL
    - Types and terms
    - Interface
    - By example: types $bool$, nat and $list$
    - Numeric Types
    - Summary

- **datatype** defines (possibly) recursive data types.

- **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.

Proof methods

- *induction* performs structural induction on some variable (if the type of the variable is a datatype).

Proof methods

- *induction* performs structural induction on some variable (if the type of the variable is a datatype).
- *auto* solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

Proof methods

- *induction* performs structural induction on some variable (if the type of the variable is a datatype).
- *auto* solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

“=” is used only from left to right!

Proofs

General schema:

lemma *name*: `"..."`
apply (...)
apply (...)
⋮
done

Proofs

General schema:

lemma *name*: `"..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*`[simp]:   "..."`

Top down proofs

Command

sorry

“completes” any proof.

The proof state

$$1.\ \bigwedge x_1 \dots x_p.\ A \Longrightarrow B$$

Multiple assumptions

$$[\![\ A_1; \dots ; A_n\ ]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

Multiple assumptions

$$[\![\, A_1; \ldots ; A_n \,]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$;$ ≈ “and”

---

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification