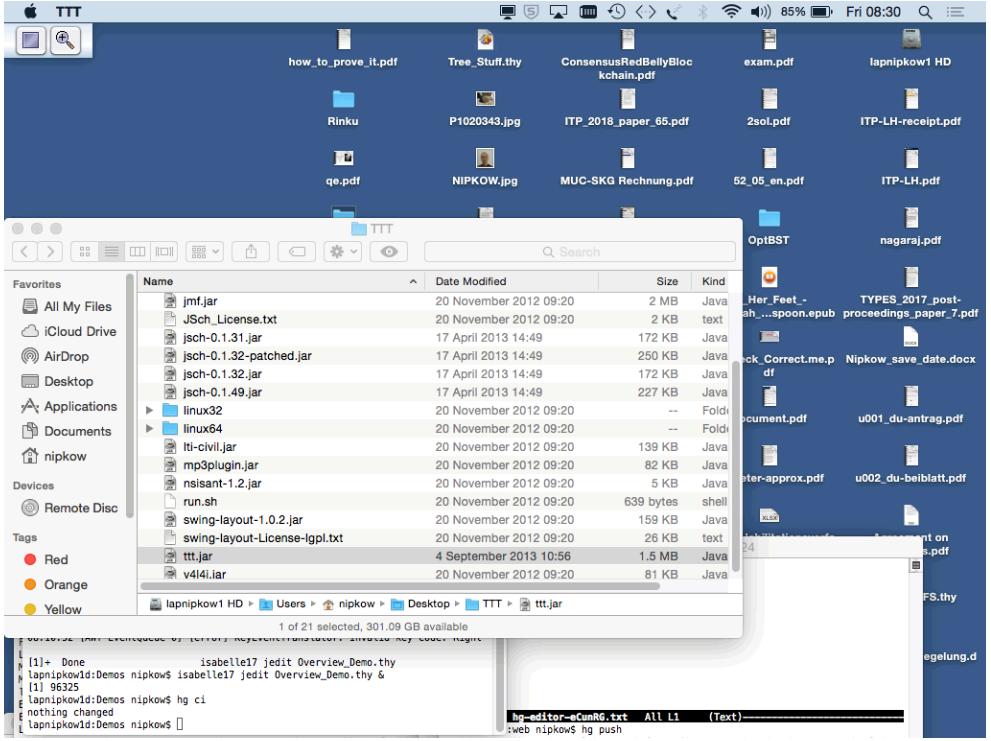
Script **generated by TTT**

Title: FDS (13.04.2018)

Date: Fri Apr 13 08:30:01 CEST 2018

Duration: 91:19 min

Pages: 80



Chapter 1

Introduction

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

Proof Assistants

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

Proof Assistants

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

Government health warnings:

Time consuming

Proof Assistants

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

Government health warnings:

Time consuming
Potentially addictive

Proof Assistants

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

Government health warnings:

Time consuming
Potentially addictive
Undermines your naive trust in informal proofs

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

Operating system
microkernel (L4)

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

Overview of course

- Week 1–5: Introduction to 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

First course in discrete mathematics

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

Part I

Isabelle

Chapter 2

Programming and Proving

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: $\Rightarrow$, $\longrightarrow$

Notation

Implication associates to the right:

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

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

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

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!

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. $1 + 2 = 4$

Types

Basic syntax:

τ	::=	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables

Types

Basic syntax:

τ	::=	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions

Types

Basic syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)

Types

Basic syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	$\tau\ list$	lists

Types

Basic syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	$\tau\ list$	lists
	$\mid$	$\tau\ set$	sets

Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)

Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$t\ t$	function application
	$\mid$	$\lambda x.\ t$	function abstraction

Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$t\ t$	function application
	$\mid$	$\lambda x.\ t$	function abstraction
	$\mid$	$\ldots$	lots of syntactic sugar

Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$t\ t$	function application
	$\mid$	$\lambda x.\ t$	function abstraction
	$\mid$	$\ldots$	lots of syntactic sugar

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:* $+, -, *, \#, @, \ldots$

Predefined syntactic sugar

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

Predefined syntactic sugar

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

Prefix binds more strongly than infix:

! $f\ x + y \ \equiv\ (f\ x) + y \ \not\equiv\ f\ (x + y)$ **!**

Predefined syntactic sugar

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

Prefix binds more strongly than infix:

! $f\ x + y \ \equiv\ (f\ x) + y \ \not\equiv\ f\ (x + y)$ **!**

Enclose *if* and *case* in parentheses:

! $(if\ \_\ then\ \_\ else\ \_)$ **!**

Theory = Isabelle Module

Theory = Isabelle Module

Syntax:

```
theory MyTh
imports T1 ... Tn
begin
(definitions, theorems, proofs, ...)*
end
```


Theory = Isabelle Module

Syntax:

```
theory MyTh
imports T1 ... Tn
begin
(definitions, theorems, proofs, ...)*
end
```


$MyTh$: name of theory. Must live in file $MyTh$`.thy`

T_i: names of *imported* theories. Import transitive.

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 "

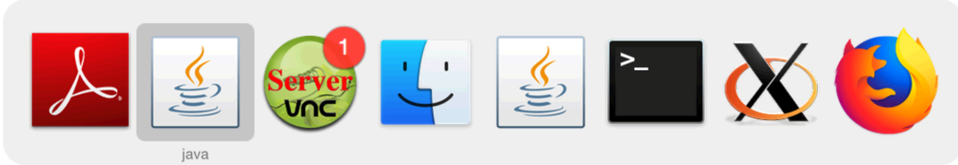
isabelle jedit

isabelle jedit

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

Overview_Demo.thy

Overview_Demo.thy



Type *bool*

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

Type *bool*

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

Predefined functions:

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

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 $bool$

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*: $0, \quad Suc\ 0, \quad Suc(Suc\ 0), \ \ldots$

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*: $0, \quad Suc\ 0, \quad Suc(Suc\ 0), \ \ldots$

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*: $0, \quad Suc\ 0, \quad Suc(Suc\ 0), \ \ldots$

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: $Suc\ z$

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

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

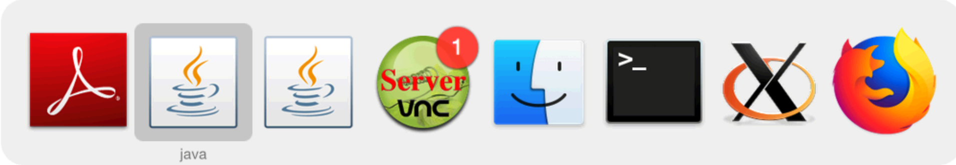
Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

! 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.

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 $add\ (Suc\ m)\ 0 = Suc\ m$.

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:
  $add\ (Suc\ m)\ 0 \ = \ Suc\ (add\ m\ 0)$ 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 $'a$

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

Type $'a$ $list$

Lists of elements of type $'a$

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

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

Type $'a$ $list$

Lists of elements of type $'a$

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

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

Syntactic sugar:

- $[] = Nil$: empty list

Type $'a$ $list$

Lists of elements of type $'a$

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

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"*)

Type $'a$ *list*

Lists of elements of type $'a$

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

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

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