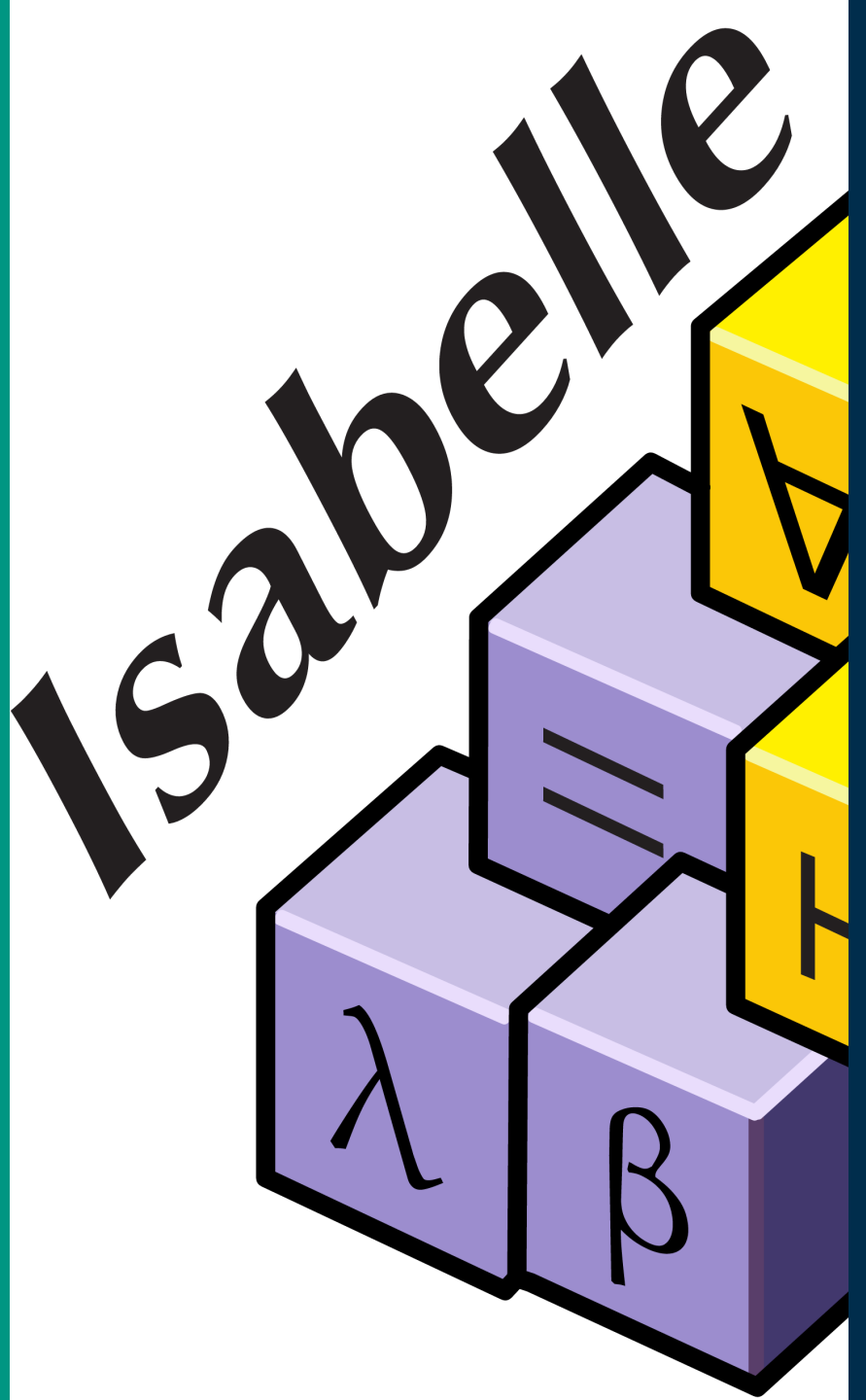
Theorem Prover Lab



Episode 1: Introduction

Henriette Färber | 2025-10-29

Previously

Nothing.

Previously

Nothing. But ideally, you are familiar with

- Basics of functional programming
- Logic, e.g., via the courses Formal Systems, Constructive Logic etc.

Today

Course Organisation

and Homework, yay!

Lectures

- 18 time slots
  - 8 lectures
  - 9 slots for project work
  - 1 slot for presentations
- Wednesday, 14:00 - 15:30 at InformatiKOM SR2
- After lectures, before winter break: project topics

Link to Website

Course Organisation

and Homework, yay!

Lectures

- 18 time slots
  - 8 lectures
  - 9 slots for project work
  - 1 slot for presentations
- Wednesday, 14:00 - 15:30 at InformatiKOM SR2
- After lectures, before winter break: project topics

Homework

- Only during lectures
- One exercise sheet per week
- Optional, no bonus points
- Exercises and submission via ILIAS (This is the only thing that we will use ILIAS for!)

Link to Website

Link to ILIAS Course

Course Organisation

and Homework, yay!

Lectures

- 18 time slots
  - 8 lectures
  - 9 slots for project work
  - 1 slot for presentations
- Wednesday, 14:00 - 15:30 at InformatiKOM SR2
- After lectures, before winter break: project topics

Homework

- Only during lectures
- One exercise sheet per week
- Optional, no bonus points
- Exercises and submission via ILIAS (This is the only thing that we will use ILIAS for!)

Link to Website

Link to ILIAS Course

Questions, discourse and announcements over on the Matrix Channel :)

Recommended References

Books and Official Documentation

- https://isabelle.in.tum.de/
- Concrete Semantics book
- Isabelle/HOL: A Proof Assistant for Higher-Order Logic

Additional Resources

- A Gentle Introduction to Isabelle
- Crash Course in Formale Systeme II - Anwendung
- Functional Data Structures and Algorithms
  Book is evolving over time!
- Archive of Formal Proofs (collection of proof libraries and examples)

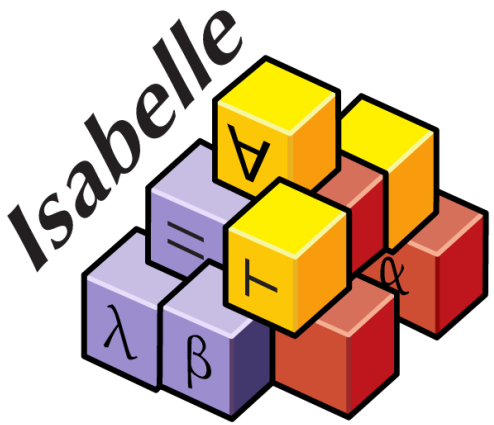
For your final project, we expect you to be able to use what we covered in our lectures.
These references go beyond that, but may deepen your understanding of the material.

What is Isabelle?

The official website states:

Isabelle is a *generic proof assistant*. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.

What does that mean?



What is Isabelle?

The official website states:

> **Isabelle is a *generic proof assistant*. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.**

- Generic infrastructure for building deductive systems
- Support for formal mathematical proofs
- **Interactive** work with formulas and proofs
- **Automatic** correctness checking

Why is this useful?

What is Isabelle?

The official website states:

> **Isabelle is a *generic proof assistant*. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.**

- Generic infrastructure for building deductive systems
- Support for formal mathematical proofs
- **Interactive** work with formulas and proofs
- **Automatic** correctness checking

- Human reasoning is error-prone; machine-checked reasoning provides a “safety net”
- Discovery of subtle mistakes (famous example: Four Color Theorem, formalized in Coq)
- Automation tools can help find proofs / proof steps!
- Export to nicely typeset text, code generation

What is Isabelle?

Do people actually use this in practice?

What is Isabelle?

Do people actually use this in practice? **Yes, they do!**

C compiler (CompCert)
Competitive with `gcc -O1`,
Won *2021 ACM Software System Award*

Xavier Leroy (& Co)
INRIA Paris
using Rocq

Operating system microkernel (seL4),
Used in safety-critical applications
Won *2022 ACM Software System Award*

Gerwin Klein (& Co)
University of New South Wales
using Isabelle

SAT solver (IsaSAT)
Won *EDA Fixed CNF Encoding Race* in 2021

Mathias Fleury (& Co)
University of Freiburg
using Isabelle

What is Isabelle/HOL?

Isabelle:

- **Generic** infrastructure for building deductive systems
- Small fixed kernel, the “**meta-logic**”, containing fundamental inference rules
- Meta-logic is extended by so called object logics

Isabelle/HOL:

- HOL: most versatile object logic for Isabelle
- Isabelle/HOL: most widespread instance of Isabelle
- What exactly is HOL? Wait for lecture 4! ;)

For now, think:

HOL = Functional Programming + Logic

Babies First Proof

```
chapter ‹ Commutativity \label{ch:com} ›

lemma or_comm: "A ∨ B ⟶ B ∨ A"
proof
  assume "A ∨ B"
  then show "B ∨ A"
  proof
    assume A
    then show "B ∨ A" by simp
  next
    assume B
    then show "B ∨ A" by simp
  qed
qed
```

This is a simple proof written in the **Isar** proof language

- Structured proofs, not linear
- Readable, (hopefully) intuitive
- Need to state what is to proven at any given point

Babies First Proof

```
chapter ‹ Commutativity \label{ch:com} ›

lemma or_comm: "A ∨ B⟶ B ∨ A"
proof
  assume"A ∨ B"
  then show "B ∨ A"
  proof
    assume  A
    then show "B ∨ A" by simp
  next
    assume B
    then show "B ∨ A" by simp
  qed
qed
```

Three different languages for writing theory content:

1. **Inner syntax**: mathematics

Babies First Proof

```
chapter ‹ Commutativity \label{ch:com} ›

lemma or_comm: "A ∨ B ⟶ B ∨ A"
proof
  assume "A ∨ B"
  then show "B ∨ A"
  proof
    assume A
    then show "B ∨ A" by simp
  next
    assume B
    then show "B ∨ A" by simp
  qed
qed
```

Three different languages for writing theory content:

1. **Inner syntax**: mathematics
2. **Textural content**, extended from LaTeX source code

Babies First Proof

```
chapter ‹ Commutativity \label{ch:com} ›

lemma or_comm: "A ∨ B ⟶ B ∨ A"
proof
  assume "A ∨ B"
  then show "B ∨ A"
  proof
    assume A
    then show "B ∨ A" by simp
  next
    assume B
    then show "B ∨ A" by simp
  qed
qed
```

Three different languages for writing theory content:

1. **Inner syntax**: mathematics
2. **Textural content**, extended from LaTeX source code
3. **Outer syntax**: organization of fragments in inner syntax and textual content

Babies First Proof

```
chapter ‹ Commutativity \label{ch:com} ›

lemma or_comm: "A ∨ B ⟶ B ∨ A"
proof
  assume "A ∨ B"
  then show "B ∨ A"
  proof
    assume A
    then show "B ∨ A" by simp
  next
    assume B
    then show "B ∨ A" by simp
  qed
qed
```

Three different languages for writing theory content:

1. **Inner syntax**: mathematics
2. **Textural content**, extended from LaTeX source code
3. **Outer syntax**: organization of fragments in inner syntax and textual content

Content in different syntax must be clearly separated:

- Inner syntax within double quotes “. . . ”
- Textual content within cartouche delimiters ‹. . . ›

Quotation marks around a single identifier can be dropped!

Babies First Proof

```
chapter ‹ Commutativity \label{ch:com} ›

lemma or_comm: "A ∨ B ⟶ B ∨ A"
proof
  assume "A ∨ B"
  then show "B ∨ A"
  proof
    assume A
    then show "B ∨ A" by simp
  next
    assume B
    then show "B ∨ A" by simp
  qed
qed
```

What does `simp` mean?

Babies First Proof

```
1 (* Generally: *)
2
3 proof
4 assume form_0
5 have form_1 by method_1
6 ...
7 have form_n by method_n
8 show thesis by ...
9 qed
```

What does `simp` mean? It's a **proof method**.

All proofs, whether generated interactively or automatically, are ultimately reduced to a sequence of steps that the kernel can check for validity.

A proof method an automated procedure that attempts to prove a goal using proven lemmas / theorems / rules.

Babies First Proof

```
1 (* Generally: *)
2
3 proof
4 assume form_0
5 have form_1 by method_1
6 ...
7 have form_n by method_n
8 show thesis by ...
9 qed
```

What does `simp` mean? It's a **proof method**.

All proofs, whether generated interactively or automatically, are ultimately reduced to a sequence of steps that the kernel can check for validity.

A proof method an automated procedure that attempts to prove a goal using proven lemmas / theorems / rules. Prominent examples:

- `simp` simplifies assumptions and conclusion using all available simplification rules
- `auto` solves as many subgoals as it can, mainly by simplification
  '=' is used only from left to right!
- `-` is the empty proof method, which does nothing

Types

Isabelle is strongly typed: every term must have a type

- **Base types**: bool, nat ($\mathbb{N}$), int ($\mathbb{Z}$)
- **Type variables** denoted by preceding prime (e.g. *'a*)
- Types (usually) specified via **`datatype`** command
- **Type constructors**: list, set, $\times$, ...

```
1  value "True" (* :: "bool" *)
2
3  value "3::nat" (* :: "nat" *)
4
5  term "3" (* :: "'a" *)
6
7  datatype color = Red | Green | Blue | Yellow
8
9  datatype 'a list = Nil | Cons 'a "'a list"
```

Types

Isabelle is strongly typed: every term must have a type

- **Base types**: bool, nat ($\mathbb{N}$), int ($\mathbb{Z}$)
- **Type variables** denoted by preceding prime (e.g. *'a*)
- Types (usually) specified via **`datatype`** command
- **Type constructors**: list, set, $\times$, ...
- **Function types** denoted by $\Rightarrow$

Type constructors...

- Written as postfix
- Take precedence over $\Rightarrow$

```
1  (* datatype bool = True | False *)
2
3  datatype 'a list = Nil | Cons 'a "'a list"
4
5  fun conj :: "bool⇒ bool⇒ bool" where
6  "conj True True = True" |
7  "conj _ _ = False"
8
9  fun flip :: "bool list⇒ bool list" where
10 "flip Nil = Nil" |
11 "flip (Cons True x) = Cons False x" |
12 "flip (Cons False x) = Cons True x"
```

Terms and Formulas

A term is either

- A constant
- Obtained via **function application**
- Obtained via **function abstraction**

However, there is also lot of syntactic sugar that we will see later on. There are also some infix symbols like $\wedge$, + and $\leq$.

```
1  term "True" (* :: "bool" *)
2
3  term "conj True False" (* :: "bool" *)
4
5  λx. x (* :: "'a⇒'a" *)
```

Terms and Formulas

A term is either

- A constant
- Obtained via **function application**
- Obtained via **function abstraction**

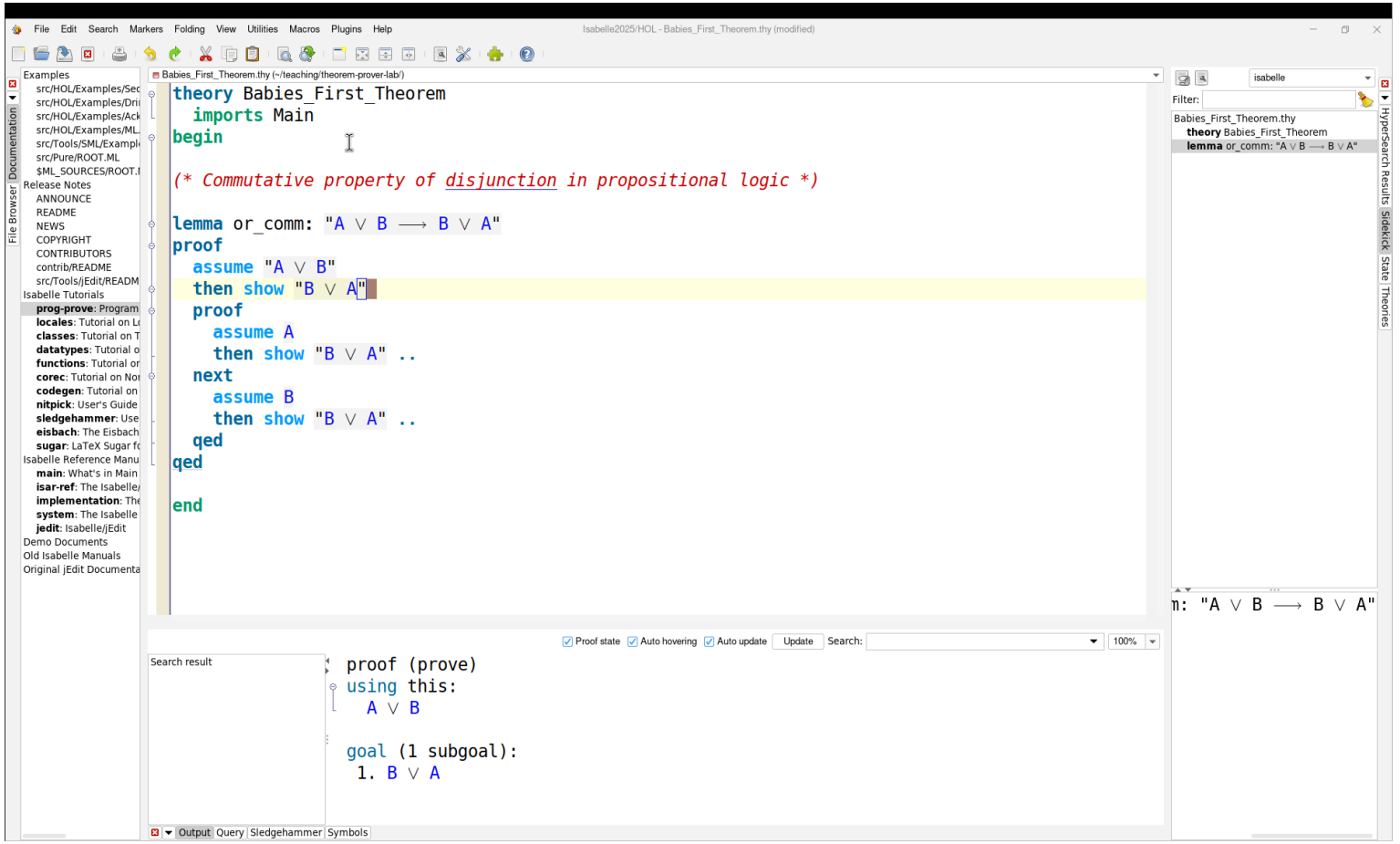
However, there is also lot of syntactic sugar that we will see later on. There are also some infix symbols like $\wedge$, + and $\leq$.

Formulas are **terms of type `bool`**:

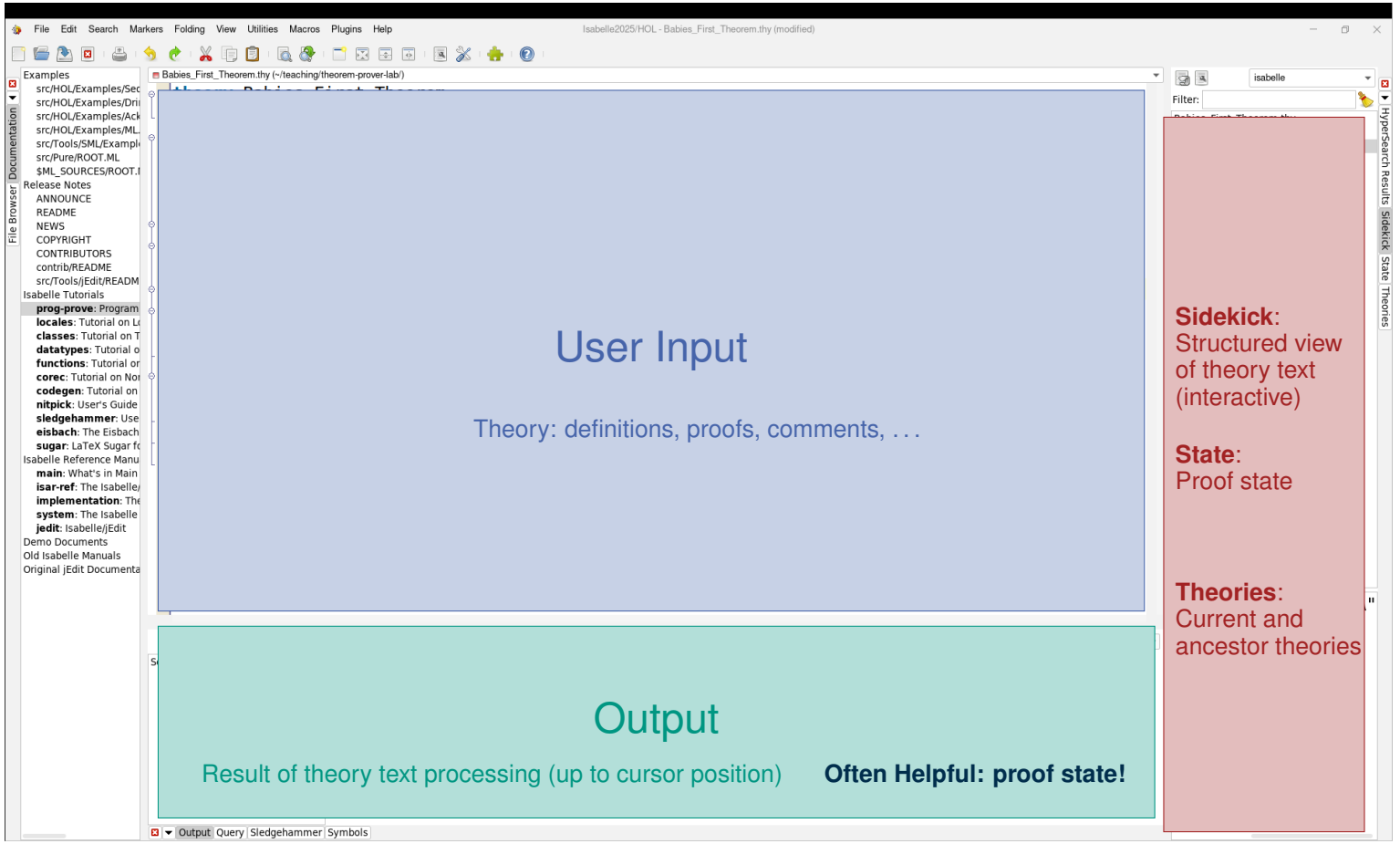
- Constants `True, False`
- Logical connectives $\neg, \wedge, \vee, \longrightarrow$
- Equality available via function $=$ (Also works on formulas!)

```
1  term "True" (* :: "bool" *)
2
3  term "conj True False" (* :: "bool" *)
4
5  λx. x (* :: "'a⇒'a" *)
6
7  term "1 = 1" (* :: "bool" *)
8
9  term "(1 = (2::nat)) = (1 = (3::nat))" (* :: "bool" *)
10
11 term "(=)" (* :: "'a⇒ 'a⇒ bool" *)
```

The Anatomy of the Editor



The Anatomy of the Editor



Break Time!

10 min

If you haven’t done so already, use the time to install Isabelle from https://isabelle.in.tum.de/installation.html

Basic Theory Structure

Isabelle/HOL takes input in form of **theory files**:

- Naming convention: `MyThy.thy`
- Theory = content of a theory file
- Theory `MyThy` must live in `MyThy.thy`!
- Each theory must import at least one theory file!

```
1 theory T
2 imports T1 ... Tn (* parent theories *)
3 begin
4
5 (* defintions, theories, proofs, ... *)
6
7 end
```

Basic Theory Structure

Isabelle/HOL takes input in form of **theory files**:

- Naming convention: `MyThy.thy`
- Theory = content of a theory file
- Theory `MyThy` must live in `MyThy.thy`!
- Each theory must import at least one theory file!

```
1 theory T
2 imports T1 ... Tn (* parent theories *)
3 begin
4
5 (* defintions, theories, proofs, ... *)
6
7 end
```

The Theory *Main*

- Union of all basic predefined theories (arithmetic, lists, sets, ...)
- Generally: always include directly or indirectly!

Peano Numerals

```
1  theory Peano
2  imports Main (* Remember! *)
3  begin
4
5  datatype nat = Zero | Suc nat
6
7  fun leq :: "nat⇒ nat⇒ bool" where
8  (* Your turn! *)
9
10 end
```

Let's build our own (Peano) numbers!

1. How can we define the $\leq$ operator?
2. How can we show that `Zero` is less or equal to all `nat` numbers?

Peano Numerals

```
1  theory Peano
2  imports Main
3  begin
4
5  datatype nat = Zero | Suc nat
6
7  fun leq :: "nat⇒ nat⇒ bool" where
8  "leq Zero _   = True" |
9  "leq (Suc m) Zero  = False" |
10 "leq (Suc m) (Suc n) = leq m n"
11
12 end
```

Let's build our own (Peano) numbers!

1. How can we define the $\leq$ operator?
2. How can we show that `Zero` is less or equal to all `nat` numbers?

Peano Numerals

```
1  theory Peano
2  imports Main
3  begin
4  (* ... *)
5
6  lemma zero_leq_all: "leq Zero n"
7  proof (cases n)
8    case Zero
9    then show ?thesis by simp
10 next
11   case (Suc m)
12   then show ?thesis by simp
13 qed
14
15 end
```

Let's build our own (Peano) numbers!

1. How can we define the $\leq$ operator?
2. How can we show that `Zero` is less or equal to all `nat` numbers?

Peano Numerals

```
1  theory Peano
2  imports Main
3  begin
4
5  datatype nat = Zero | Suc nat
6
7  fun leq :: "nat⇒ nat⇒ bool" where
8  "leq Zero _   = True" |
9  "leq (Suc m) Zero  = False" |
10 "leq (Suc m) (Suc n) = leq m n"
11
12 (* Easier: *)
13 lemma zero_leq_all: "leq Zero n" by simp
14
15 end
```

Let's build our own (Peano) numbers!

1. How can we define the $\leq$ operator?
2. How can we show that `Zero` is less or equal to all `nat` numbers?
3. How do we define addition for our `nats`?

Peano Numerals

```
1  theory Peano
2  imports Main
3  begin
4
5  datatype nat = Zero | Suc nat
6
7  fun add :: "nat⇒ nat⇒ nat" where
8  "add Zero n = n" |
9  "add (Suc m) n = Suc(add m n)"
10
11 end
```

Let's build our own (Peano) numbers!

1. How can we define the $\leq$ operator?
2. How can we show that `Zero` is less or equal to all `nat` numbers?
3. How do we define addition for our `nats`?

Simple Induction

```
1  datatype nat = Zero | Suc nat
2
3  fun add :: "nat⇒ nat⇒ nat" where
4  "add Zero n = n" |
5  "add (Suc m) n = Suc(add m n)"
6
7  lemma add_zero: "add m Zero = m"
8  proof
9  (* ... *)
10 qed
```

How would you prove this on paper?

Simple Induction

```
1  datatype nat = Zero | Suc nat
2
3  fun add :: "nat⇒ nat⇒ nat" where
4  "add Zero n = n" |
5  "add (Suc m) n = Suc(add m n)"
6
7  lemma add_zero: "add m Zero = m"
8  proof
9  (* ... *)
10 qed
```

How would you prove this on paper? **Induction!**

Intuitively:

IB: `add Zero Zero = Zero` by definition of `add`

IH: For some arbitrary but fixed `n`, assume
`add n Zero = n`

IS: Show `add (Suc n)Zero = Suc n`
By definition of `add`:
`add (Suc n)Zero = Suc(add n Zero)`
Together with the IH, we thus show
`add (Suc n)Zero = Suc n`.

Simple Induction

```
1  datatype nat = Zero | Suc nat
2
3  fun add :: "nat⇒ nat⇒ nat" where
4  "add Zero n = n" |
5  "add (Suc m) n = Suc(add m n)"
6
7  lemma add_zero: "add m Zero = m"
8  proof (induction m)
9    case Zero
10   then show ?case by simp
11 next
12   case (Suc m)
13   then show ?case by simp
14 qed
```

How would you prove this on paper? **Induction!**

Intuitively:

IB: `add Zero Zero = Zero` by definition of `add`

IH: For some arbitrary but fixed `n`, assume
`add n Zero = n`

IS: Show `add (Suc n)Zero = Suc n`
By definition of `add`:
`add (Suc n)Zero = Suc(add n Zero)`
Together with the IH, we thus show
`add (Suc n)Zero = Suc n`.

The proof method `induction` performs structural induction on some variable (if the type of the variable is a datatype).

Conclusion

What you should be able to answer now:

- What Isabelle/HOL is and why you might want to use it
- How to define types in Isabelle/HOL
- How to structure simple proofs with Isar
- How to do simple induction in Isabelle/HOL

Conclusion

What you should be able to answer now:

- What Isabelle/HOL is and why you might want to use it
- How to define types in Isabelle/HOL
- How to structure simple proofs with Isar
- How to do simple induction in Isabelle/HOL

Until next week:

- Join the Matrix Channel
- Find a partner for your project
  (or decide that you want to work alone)
- Download and work on the first exercise sheet
- Submit your solution to ILIAS

See you next week! :)

Further Examples

The following are larger examples of how to structure Isar proofs. Notice that you can

- Label cases and assumptions for easier reference
- Use previously proved lemmas via the keyword `using`
- Reference an induction hypothesis via `casename.IH`

```
lemma id_leq: "leq n n"
  by (induction n) simp_all

lemma add_suc_2 : "Suc(add n m) = add n (Suc m)"
by (induction n) simp_all
```

```
lemma leq_suc: "leq n m⟶ leq n (Suc m)"
proof (induction n arbitrary: m)
  case Zero
  then show ?case by simp
next
  case step: (Suc k)
  then show ?case
  proof (cases m)
    case Zero
    then show ?thesis by simp
  next
    case (Suc l)
    then show ?thesis using step.IH by simp
  qed
qed
```

Further Examples

```
lemma leq_sum: "leq m (add m n)"
proof(induction n)
  case Zero
  then show ?case using add_zero id_leq by auto
next
  case step: (Suc k)
  then show ?case
  proof (cases m)
    case Zero
    then show ?thesis by simp
  next
    case m_is_suc: (Suc l)
    then have "leq m (Suc (add l k))" using step by auto
    then have "leq m (add l (Suc k))" using add_suc by simp
    then have "leq m (Suc (add l (Suc k)))" using leq_suc by auto
    then have "leq m (add m (Suc k))" using m_is_suc by auto
    then show ?thesis by simp
  qed
qed
```