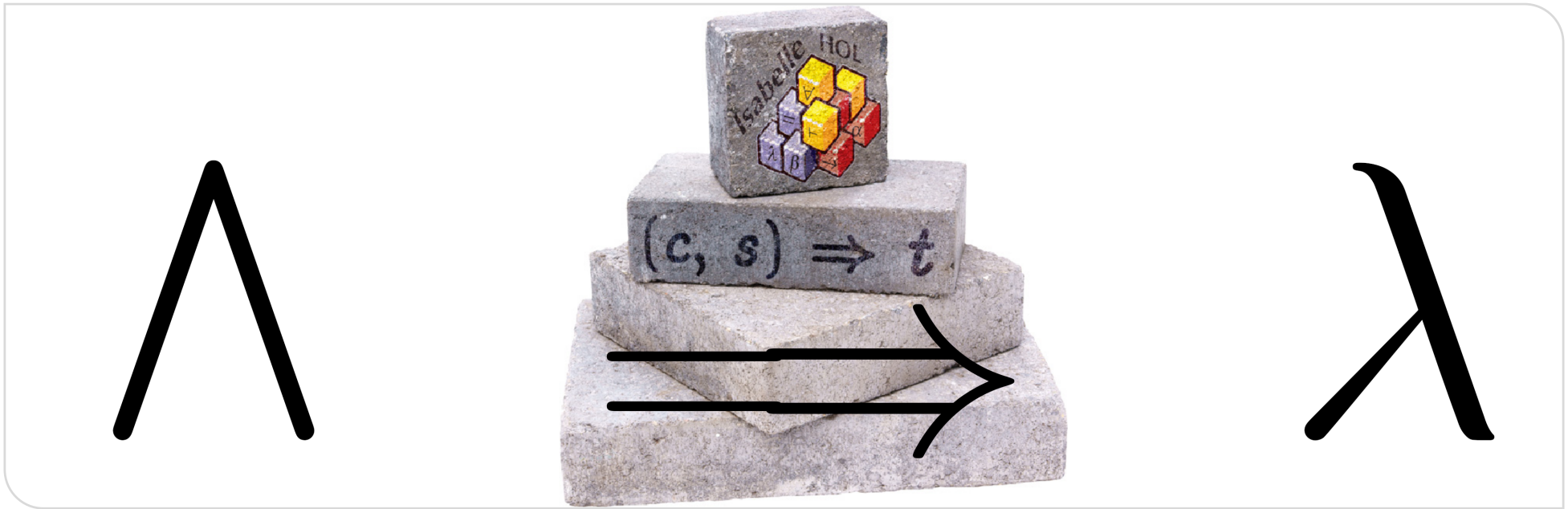
Karlsruhe Institute of Technology

KASTEL

Formal Systems II: Application

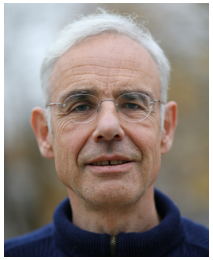
Higher-Order Logics and Interactive Theorem Proving with Isabelle

Bernhard Beckert · Mattias Ulbrich · Michael Kirsten · Alexander Weigl | SS 2023



Credits

Most material (originally) shamelessly stolen from



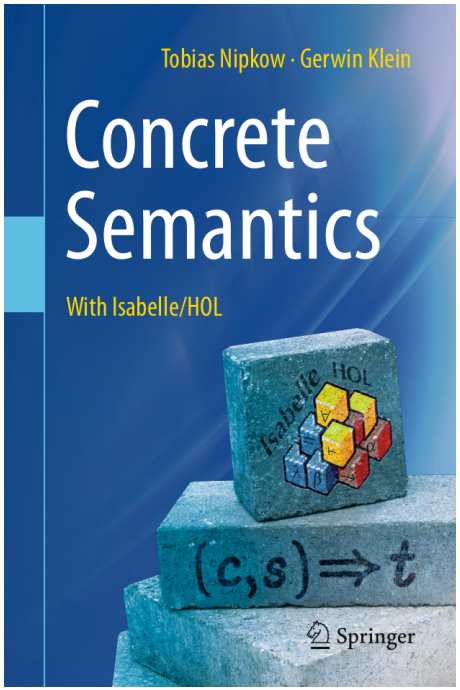
Tobias Nipkow, Lawrence Paulson, Makarius Wenzel



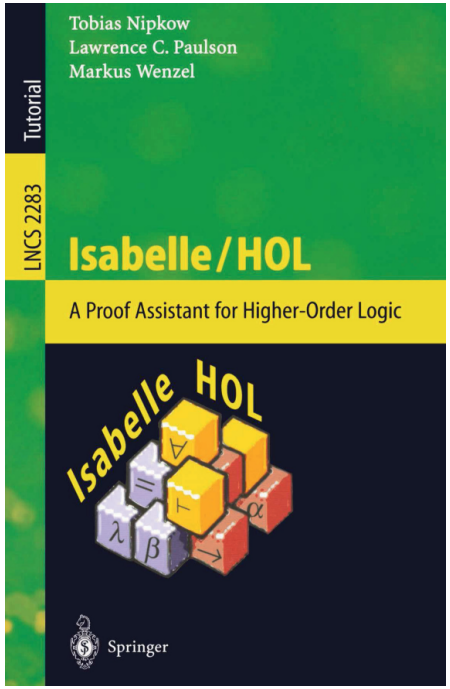
Gerwin Klein, John Harrison, Christian Sternagel

Don’t blame them, errors are mine!

Literature

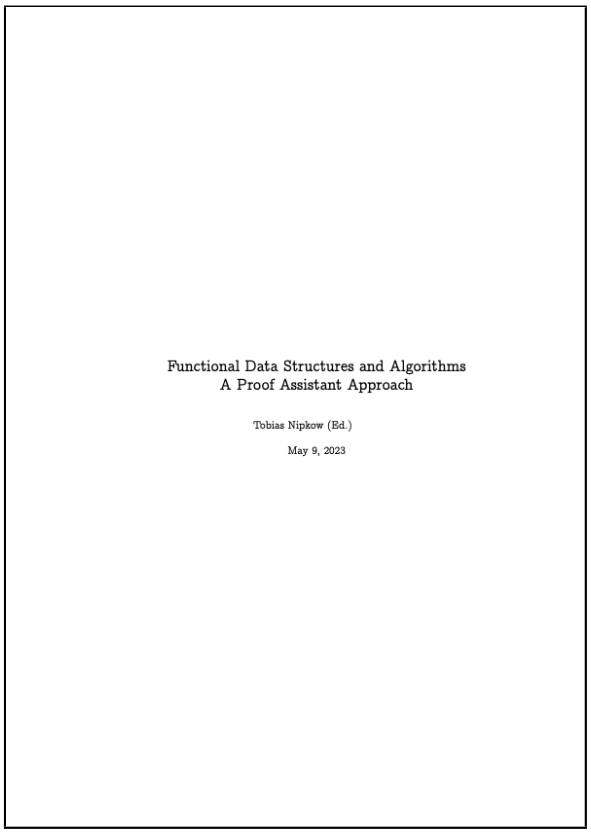


Tobias Nipkow and Gerwin Klein:
Concrete Semantics:
With Isabelle/HOL
Springer International Publishing, 2014
`http://concrete-semantics.org/`



Tobias Nipkow and Markus Wenzel:
Isabelle/HOL:
A Proof Assistant for
Higher-Order Logic
Lecture Notes in Computer Science
Springer-Verlag, 2002

Literature (cont'd)


Functional Data Structures and Algorithms
A Proof Assistant Approach

Tobias Nipkow (Ed.)

May 9, 2023

Tobias Nipkow (Ed.):
Functional Data Structures and Algorithms: A Proof Assistant Approach
Under Development, 2023
`https://functional-algorithms-verified.org`

This book is meant to evolve over time. If you would like to contribute, get in touch!

What is a Theorem Prover?

Implementation of a formal logic on a computer.

What is a Theorem Prover?

Implementation of a formal logic on a computer.

- Fully automated (propositional logic)
- Automated, but not necessarily terminating (first order logic)
- With automation, but mainly interactive (higher order logic)

What is a Theorem Prover?

Implementation of a formal logic on a computer.

- Fully automated (propositional logic)
- Automated, but not necessarily terminating (first order logic)
- With automation, but mainly interactive (higher order logic)

- Based on rules and axioms
- Can deliver proofs
- Examples: CL2, Agda, Coq, HOL (HOL Light, HOL4, ProofPower, HOL Zero), IMPS, Isabelle, Metamath, Mizar, NuprI, PVS, Lean, ...

What is a Theorem Prover?

Implementation of a formal logic on a computer.

- Fully automated (propositional logic)
- Automated, but not necessarily terminating (first order logic)
- With automation, but mainly interactive (higher order logic)

- Based on rules and axioms
- Can deliver proofs
- Examples: CL2, Agda, Coq, HOL (HOL Light, HOL4, ProofPower, HOL Zero), IMPS, Isabelle, Metamath, Mizar, NuprI, PVS, Lean, ...

There are other (algorithmic) verification tools:

- Model checking, static analysis, ...
- Usually do not deliver proofs
- See further topics of this course

Some Impressive Verifications

C compiler (CompCert)

Some Impressive Verifications

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

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

Operating system microkernel (seL4),

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

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

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

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

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

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

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

SAT solver (IsaSAT)

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

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

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

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

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Coq

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

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

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

Mathias Fleury (& Co)
JHK University Linz
using Isabelle

Choice of Foundations

Usually balancing simplicity against flexibility/expressiveness

Choice of Foundations

Usually balancing simplicity against flexibility/expressiveness

Set theory (“traditional” or “standard” foundation for mathematics)

- Standard ZF/ZFC: Metamath and Isabelle/ZF
- Tarski-Grothendieck set theory: Mizar

Choice of Foundations

Usually balancing simplicity against flexibility/expressiveness

Set theory (“traditional” or “standard” foundation for mathematics)

- Standard ZF/ZFC: Metamath and Isabelle/ZF
- Tarski-Grothendieck set theory: Mizar

Type theory (more computer science interconnections)

- Simple type theory: HOL family and Isabelle/HOL
- Martin-Löf type theory: Agda, Nuprl
- Calculus of inductive constructions: Coq
- Other typed formalisms: IMPS, PVS

Choice of Foundations

Usually balancing simplicity against flexibility/expressiveness

Set theory (“traditional” or “standard” foundation for mathematics**)**

- Standard ZF/ZFC: Metamath and Isabelle/ZF
- Tarski-Grothendieck set theory: Mizar

Type theory (more computer science interconnections)

- Simple type theory: HOL family and Isabelle/HOL
- Martin-Löf type theory: Agda, Nuprl
- Calculus of inductive constructions: Coq
- Other typed formalisms: IMPS, PVS

Others

- Primitive recursive arithmetic (no explic. quantifiers): ACL2, NQTHM
- Homotopy type theory with experimental implementations

Software Architecture

Karlsruhe Institute of Technology

"The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code." (John Harrison, Intel Corporation)

Software Architecture

"The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code." (John Harrison, Intel Corporation)

- **de Bruijn approach:**
  Generate proofs that can be certified by a simple, separate checker

Software Architecture

"The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code." (John Harrison, Intel Corporation)

- **de Bruijn approach:**
  Generate proofs that can be certified by a simple, separate checker
- **LCF approach (orig. *Logic of Computable Functions*):**
  Reduce all rules to sequences of primitive inferences implemented by a small logical *kernel*
  - Specification methods and automatic proof procedures expand to full proofs
  - Unsoundness less likely
  - Implementation more complicated, performance can suffer
  - Examples: Isabelle, HOL, Coq

Higher-Order Logic (HOL)

- FOL extended with functions and sets, i.e.,
  functions are values, too!
- Polymorphic types, incl. truth value type
- No distinction between terms and formulas
- ML-style functional programming

Higher-Order Logic (HOL)

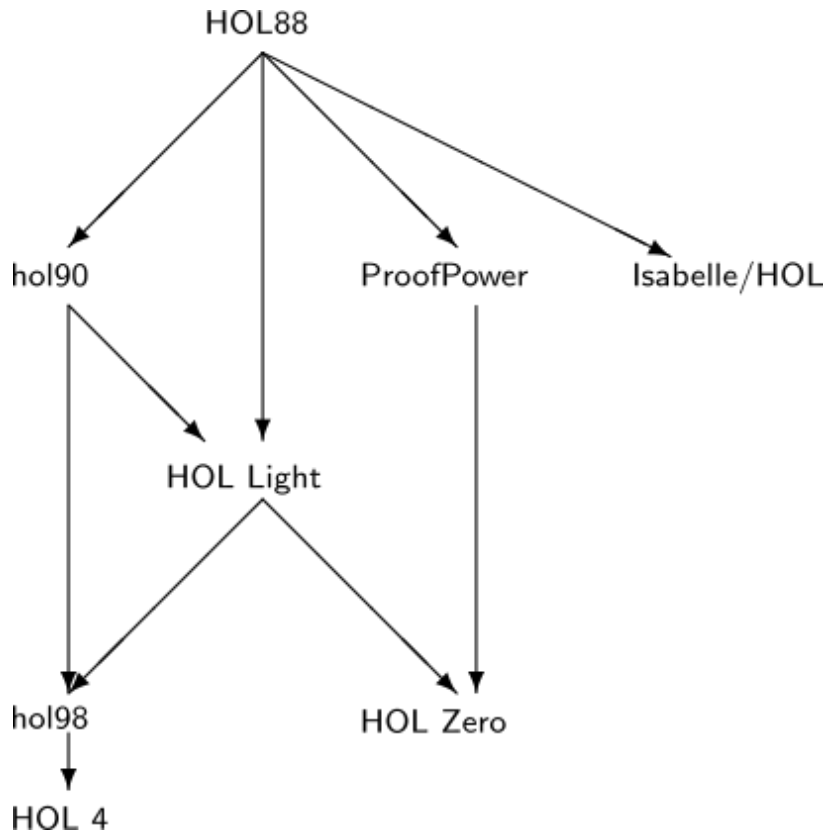
- FOL extended with functions and sets, i.e.,
  functions are values, too!
- Polymorphic types, incl. truth value type
- No distinction between terms and formulas
- ML-style functional programming

"HOL = functional programming + logic"

Higher-Order Logic (HOL)

- FOL extended with functions and sets, i.e., functions are values, too!
- Polymorphic types, incl. truth value type
- No distinction between terms and formulas
- ML-style functional programming

"HOL = functional programming + logic"



What is Isabelle?

A generic interactive proof assistant

What is Isabelle?

A generic interactive proof assistant

Generic:

Not specialized to one particular logic
(two large developments: HOL and ZF, will mainly use HOL)

What is Isabelle?

A generic interactive proof assistant

Generic:

Not specialized to one particular logic
(two large developments: HOL and ZF, will mainly use HOL)

Interactive:

More than just yes/no, you can interactively guide the system

What is Isabelle?

A generic interactive proof assistant

Generic:
Not specialized to one particular logic
(two large developments: HOL and ZF, will mainly use HOL)

Interactive:
More than just yes/no, you can interactively guide the system

Proof Assistant:
Given proof structure, helps to explore, find, and maintain proofs by checking correctness of each step

System Architecture

Isabelle – generic, interactive theorem prover

System Architecture

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

HOL, ZF – object logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

Prover IDE (jEdit) – user interface

HOL, ZF – object logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

Prover IDE (jEdit) – user interface

Scala – connects ML to JVM

HOL, ZF – object logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

System Architecture

Prover IDE (jEdit) – user interface

Scala – connects ML to JVM

HOL, ZF – object logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

User can access all layers!

System Architecture

Prover IDE (jEdit) – user interface	Isabelle/PIDE, Isabelle/jEdit
Scala – connects ML to JVM	Isabelle/Scala
HOL, ZF – object logics	Isabelle/HOL, Isabelle/ZF
Isabelle – generic, interactive theorem prover	Isabelle/Pure
Standard ML – logic implemented as ADT	Isabelle/ML

User can access all layers!

System Architecture

Prover IDE (jEdit) – user interface	Isabelle/PIDE, Isabelle/jEdit
Scala – connects ML to JVM	Isabelle/Scala
HOL, ZF – object logics	Isabelle/HOL, Isabelle/ZF
Isabelle – generic, interactive theorem prover	Isabelle/Pure
Standard ML – logic implemented as ADT	Isabelle/ML

User can access all layers!

System Requirements

- **Linux**, **Windows**, or **MacOS X** (10.8 +)
- **Standard ML** (PolyML implementation)
- **Java** (for jEdit)
- Download from `https://isabelle.in.tum.de` with lots of documentation

Let us start with the basics . . .

Which of the following three formulas have the same meaning?

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

Let us start with the basics ...

Which of the following three formulas have the same meaning?

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

Notation:

- $A \Longrightarrow (B \Longrightarrow C)$ means $A \Longrightarrow B \Longrightarrow C$ means $[\![A; B]\!] \Longrightarrow C$
- $\dfrac{A_1 \qquad \cdots \qquad A_n}{C}$ means $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow C$

Basic Syntax

Types:

$$\tau ::= (\tau)$$

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

Basic Syntax

Types:

$$\tau ::= (\tau)$$

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

Terms:

$$t ::= (t)$$

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

Basic Syntax

Types:

$$\tau ::= (\tau)$$

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

Terms:

$$t ::= (t)$$

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

$\ldots$ must be well-typed!

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$
- Automatic type inference, except for, e.g., *overloaded* constants and functions
- User-provided type annotations $((t :: \tau))$ sometimes required

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$
- Automatic type inference, except for, e.g., *overloaded* constants and functions
- User-provided type annotations ($(t :: \tau)$) sometimes required
- Currying of functions to allow for partial application

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$
- Automatic type inference, except for, e.g., *overloaded* constants and functions
- User-provided type annotations ($(t :: \tau)$) sometimes required
- Currying of functions to allow for partial application
- A formula is a term of type $bool$ (**datatype** $bool = True \mid False$)
- Inclosing of types, terms and formulas (except single identifiers) in "-signs
- Basic formula syntax: (roughly in order of precedence)
  (A), $t = u$, $\neg A$, $A \wedge B$, $A \vee B$, $A \longrightarrow B$, $A \longleftrightarrow B$, $\forall x.\ A$, $\exists x.\ A$

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$
- Automatic type inference, except for, e.g., *overloaded* constants and functions
- User-provided type annotations $((t :: \tau))$ sometimes required
- Currying of functions to allow for partial application
- A formula is a term of type $bool$ (**datatype** $bool = True \mid False$)
- Inclosing of types, terms and formulas (except single identifiers) in "-signs
- Basic formula syntax: (roughly in order of precedence)
  (A), $t = u$, $\neg A$, $A \wedge B$, $A \vee B$, $A \longrightarrow B$, $A \longleftrightarrow B$, $\forall x.\ A$, $\exists x.\ A$

Predefined syntactic sugar:

- *Infix:* $+$, $-$, *, #, @, ...
- *Mixfix:* if __ then __ else __, let __ in __, case __ of __ , ...

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$
- Automatic type inference, except for, e.g., *overloaded* constants and functions
- User-provided type annotations $((t :: \tau))$ sometimes required
- Currying of functions to allow for partial application
- A formula is a term of type $bool$ (**datatype** $bool = True \mid False$)
- Inclosing of types, terms and formulas (except single identifiers) in "-signs
- Basic formula syntax: (roughly in order of precedence)
  (A), $t = u$, $\neg A$, $A \wedge B$, $A \vee B$, $A \longrightarrow B$, $A \longleftrightarrow B$, $\forall x.\ A$, $\exists x.\ A$

Predefined syntactic sugar:

- *Infix:* $+$, $-$, *, #, @, ... (Prefix binds more strongly!)
- *Mixfix:* if __ then __ else __, let __ in __, case __ of __ , ...

Some More Basics

- Automatic β-reduction, e.g., $(\lambda x.\ x + 5)\ 3 = 3 + 5$
- Automatic type inference, except for, e.g., *overloaded* constants and functions
- User-provided type annotations $((t :: \tau))$ sometimes required
- Currying of functions to allow for partial application
- A formula is a term of type $bool$ (**datatype** $bool = True \mid False$)
- Inclosing of types, terms and formulas (except single identifiers) in "-signs
- Basic formula syntax: (roughly in order of precedence)
  (A), $t = u$, $\neg A$, $A \wedge B$, $A \vee B$, $A \longrightarrow B$, $A \longleftrightarrow B$, $\forall x.\ A$, $\exists x.\ A$

Predefined syntactic sugar:

- *Infix:* $+$, $-$, *, #, @, ... (Prefix binds more strongly!)
- *Mixfix:* if __ then __ else __, let __ in __, case __ of __ , ...
  (Enclose in parentheses!)

Theory = Isabelle Module

Syntax:

```
theory MyTh
imports T1 ... Tn
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`

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 (directly) *imported* theories (imports are 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`

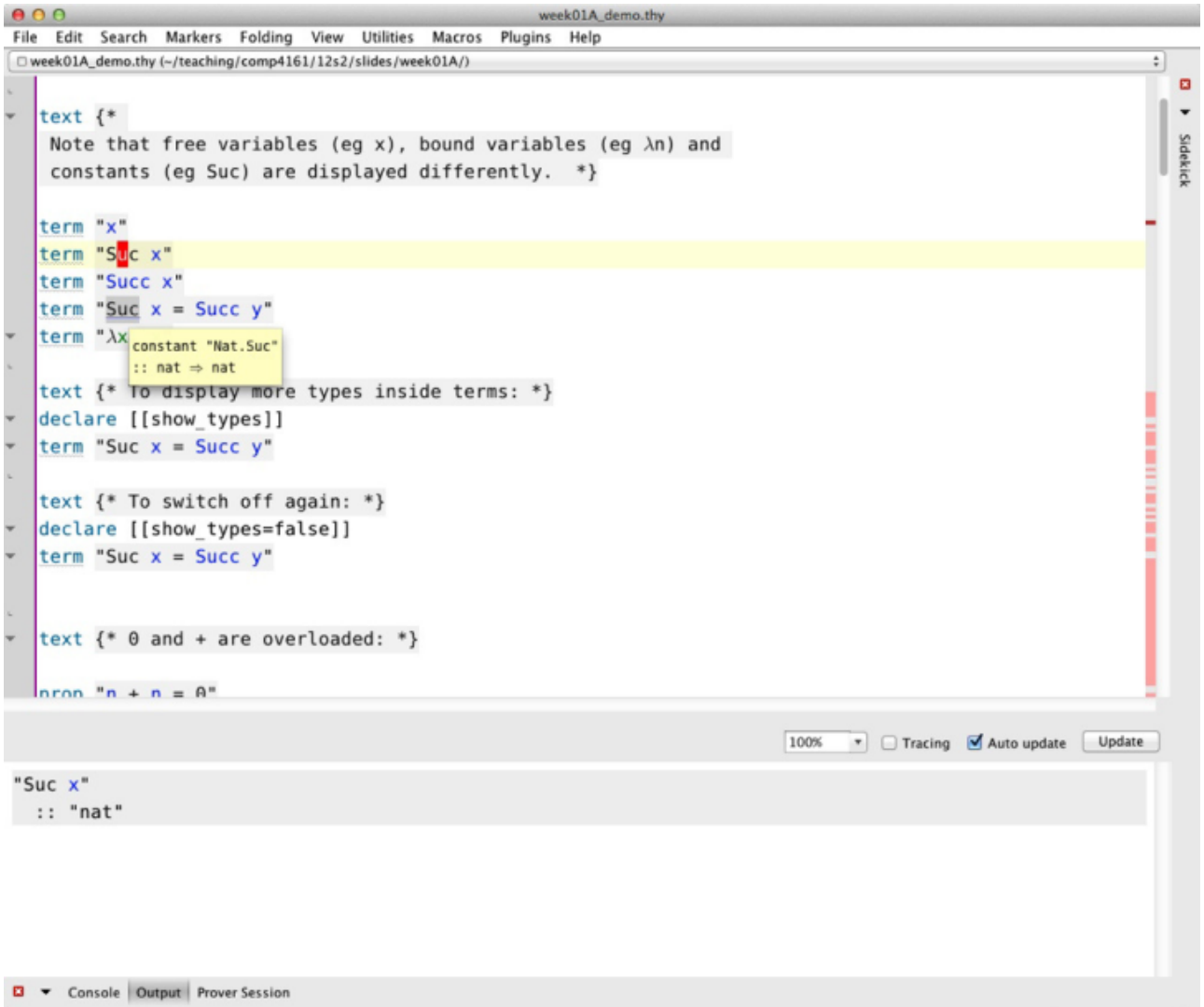
T_i**:** Names of (directly) *imported* theories (imports are transitive)

Usually: `imports Main`

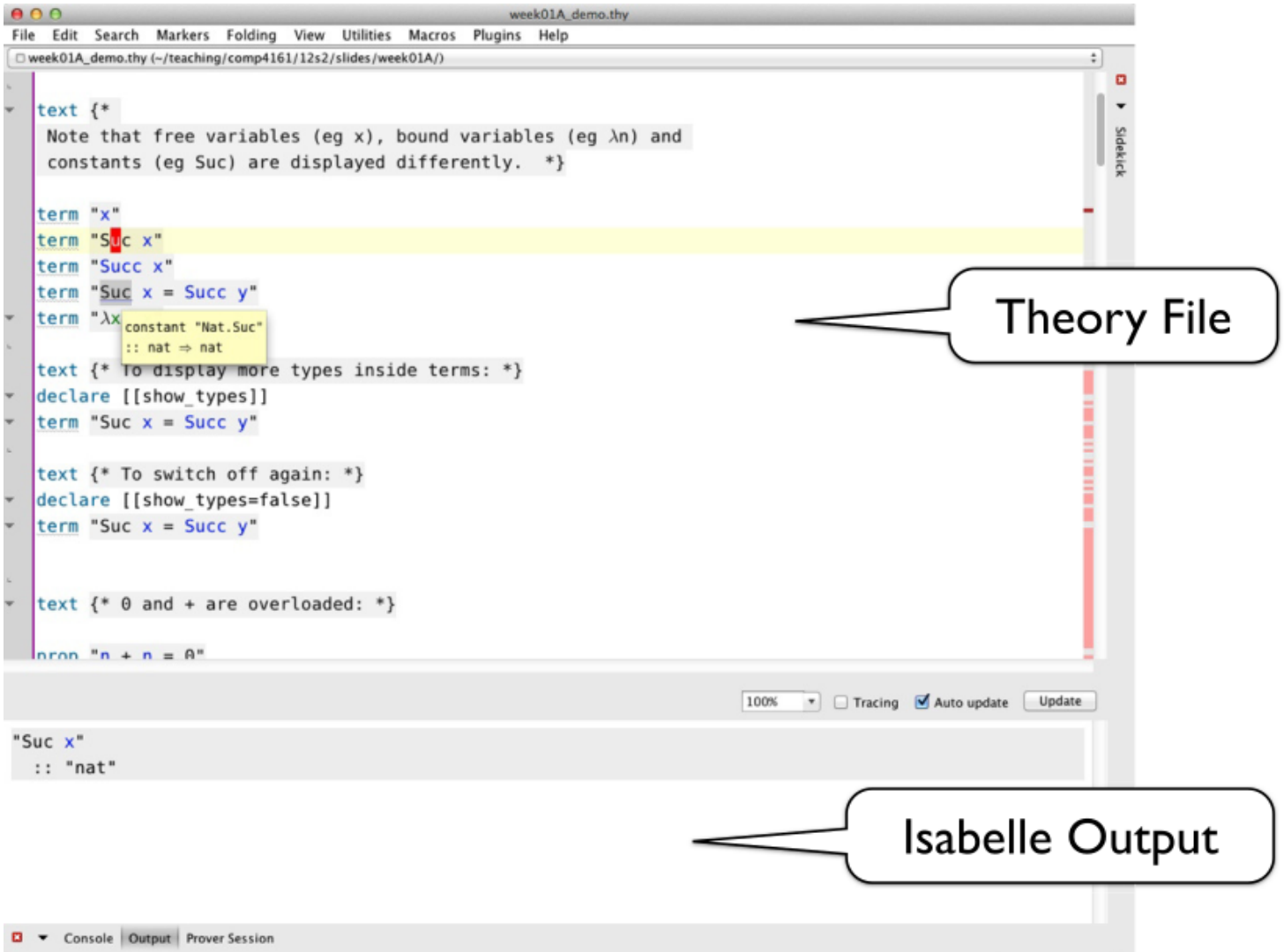
Download

https://www21.in.tum.de/teaching/fds/SS22/assets/Demos/Overview_Demo.thy

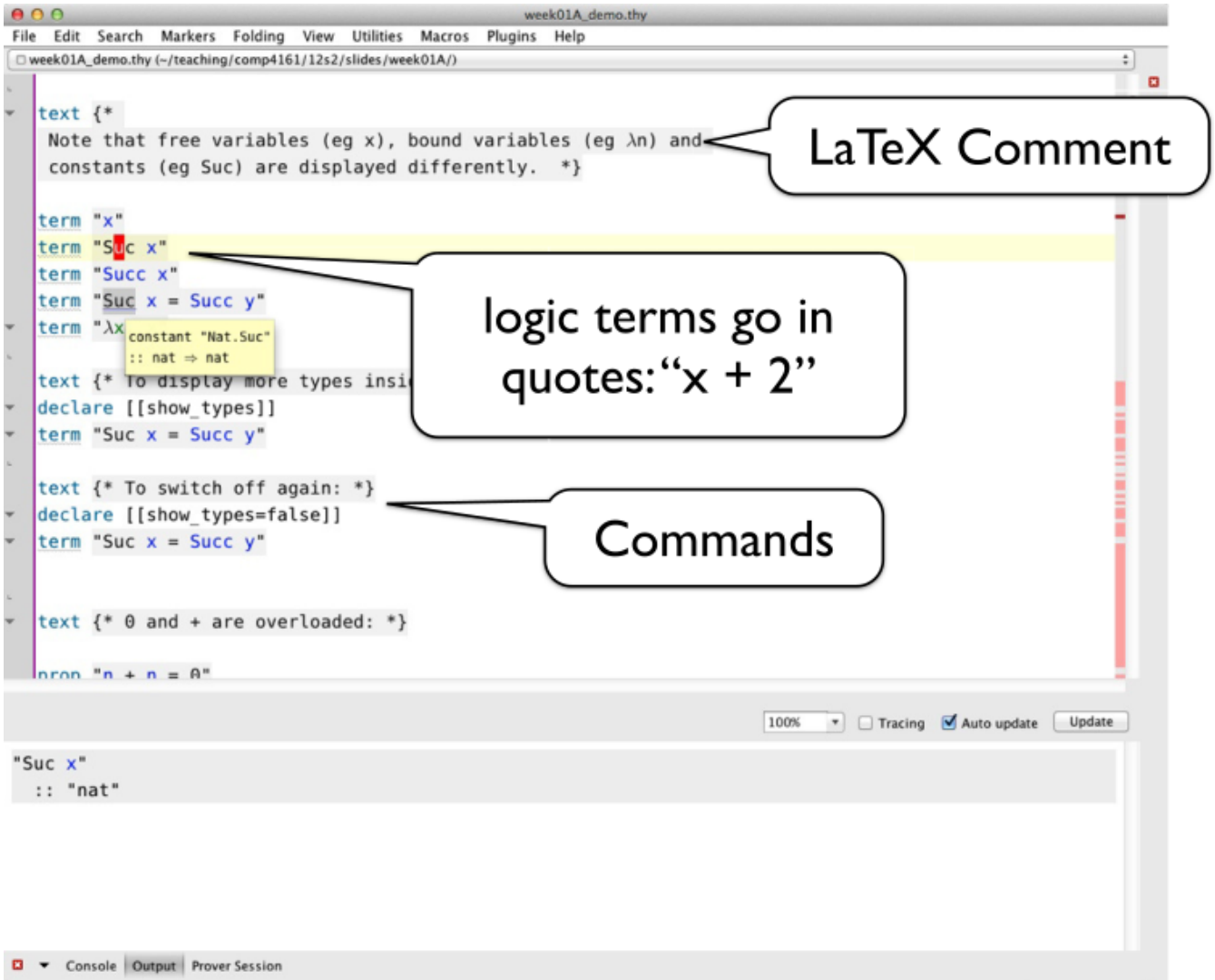
Isabelle jEdit/PIDE



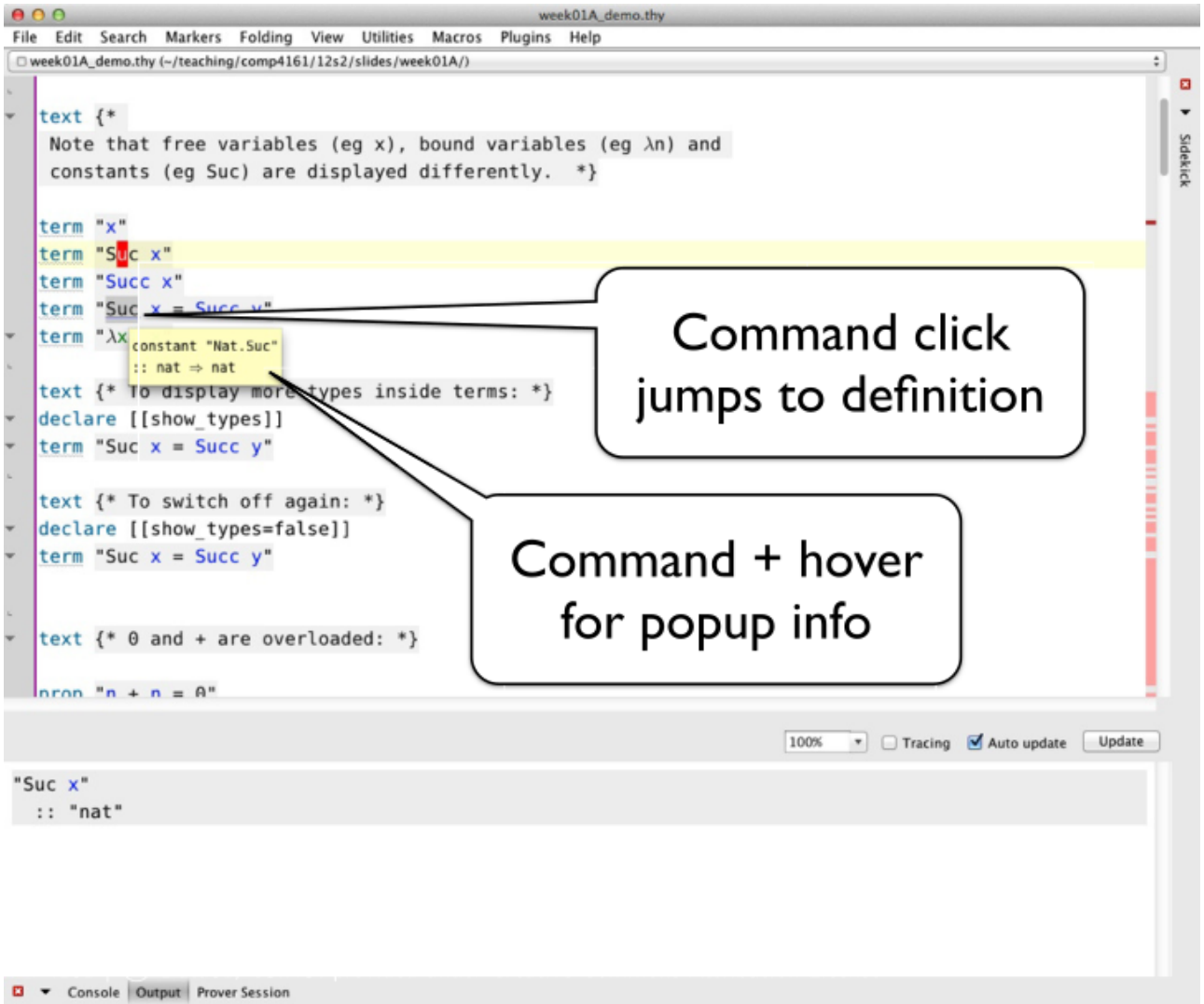
Isabelle jEdit/PIDE



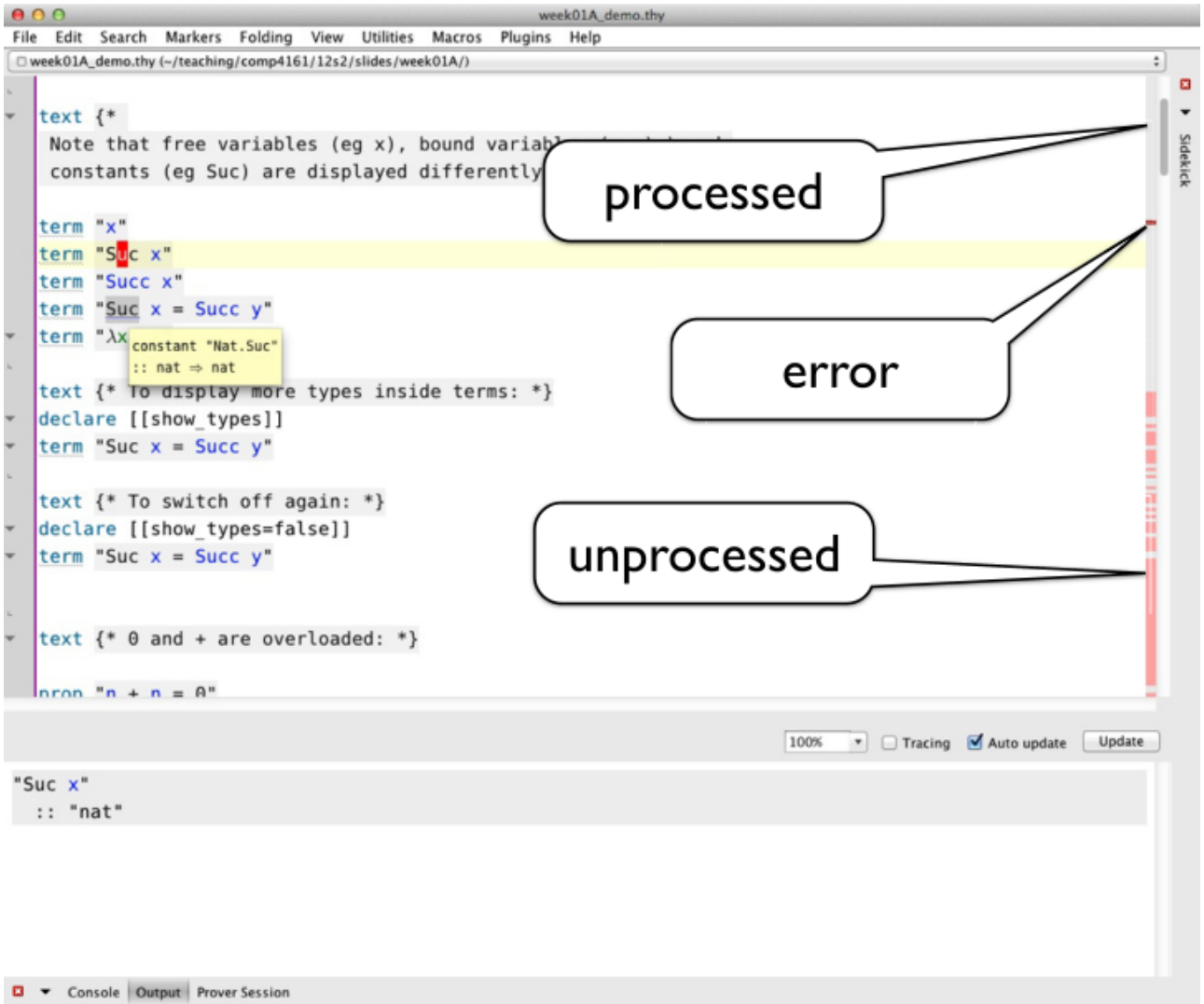
Isabelle jEdit/PIDE



Isabelle jEdit/PIDE



Isabelle jEdit/PIDE



Exploring a Theory in Isabelle

Some Diagnostic Commands

find_theorems $\langle args \rangle$	print all theorems matching $\langle args \rangle$
print_cases	print currently available cases
prop $\langle formula \rangle$	print proposition $\langle formula \rangle$
term $\langle term \rangle$	print term $\langle term \rangle$ and its type
thm $\langle name \rangle$	print theorem called $\langle name \rangle$
typ $\langle type \rangle$	print type $\langle type \rangle$
value $\langle term \rangle$	evaluate and print $\langle term \rangle$

Exploring a Theory in Isabelle

Some Diagnostic Commands

find_theorems $\langle args \rangle$	print all theorems matching $\langle args \rangle$
print_cases	print currently available cases
prop $\langle formula \rangle$	print proposition $\langle formula \rangle$
term $\langle term \rangle$	print term $\langle term \rangle$ and its type
thm $\langle name \rangle$	print theorem called $\langle name \rangle$
typ $\langle type \rangle$	print type $\langle type \rangle$
value $\langle term \rangle$	evaluate and print $\langle term \rangle$

Three Kinds of Variables

- Free variables (blue in Isabelle/jEdit)
- Bound variables (green in Isabelle/jEdit)
- Schematic variables (dark blue with leading ? in Isabelle/jEdit); can be replaced by arbitrary values

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

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

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:
$0, 1, 2, \ldots :: 'a, :: 'a \Rightarrow 'a \Rightarrow 'a$

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:
$0, 1, 2, \ldots :: 'a, :: 'a \Rightarrow 'a \Rightarrow 'a$

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

Download

https://www21.in.tum.de/teaching/fds/SS22/assets/Demos/Nat_Demo.thy

Types, Functions, Lemmas, Proof Methods

- datatype defines (possibly) recursive data types.
- fun defines (possibly) recursive functions by pattern-matching over datatype constructors.
- lemma or theorem defines a theorem (that has to be proven), either as a single formula or broken into fixed variables (**fixes**), assumptions (**assumes**), and proof obligation (**shows**)

Types, Functions, Lemmas, Proof Methods

- datatype defines (possibly) recursive data types.
- fun defines (possibly) recursive functions by pattern-matching over datatype constructors.
- lemma or theorem defines a theorem (that has to be proven), either as a single formula or broken into fixed variables (**fixes**), assumptions (**assumes**), and proof obligation (**shows**)

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]`: "..."

Proofs

General Schema:

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

If the lemma is suitable as a simplification rule: **lemma** *name*`[simp]`: "..."

The Proof State: $\bigwedge x_1 \ldots x_p.\ A \Longrightarrow B$

$x_1 \ldots x_p$	fixed local variables
A	local assumption(s)
B	actual (sub)goal

Proofs

General Schema: for apply scripts

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

If the lemma is suitable as a simplification rule: **lemma** *name*`[simp]`: "..."

The Proof State: $\bigwedge x_1 \ldots x_p.\ A \Longrightarrow B$

$x_1 \ldots x_p$	fixed local variables
A	local assumption(s)
B	actual (sub)goal

Proofs

General Schema: for apply scripts

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

Unreadable, hard to maintain, and does not scale!

If the lemma is suitable as a simplification rule: **lemma** *name*`[simp]`: "..."

The Proof State: $\bigwedge x_1 \ldots x_p.\ A \Longrightarrow B$

$x_1 \ldots x_p$	fixed local variables
A	local assumption(s)
B	actual (sub)goal

Proofs Using Isabelle/Isar

proof
  assume $formula_0$
  have $formula_1$ **by** $method_1$
  ⋮
  have $formula_n$ **by** $method_n$
  show $formula_{n+1}$ **by** ...
qed

Proofs Using Isabelle/Isar

proof
  assume $formula_0$
  have $formula_1$ **by** $method_1$
  ⋮
  have $formula_n$ **by** $method_n$
  show $formula_{n+1}$ **by** ...
qed

proves $formula_0 \Rightarrow formula_{n+1}$

Proofs Using Isabelle/Isar

proof
  assume $formula_0$
  have $formula_1$ **by** $method_1$
  ⋮
  have $formula_n$ **by** $method_n$
  show $formula_{n+1}$ **by** ...
qed

proves $formula_0 \Rightarrow formula_{n+1}$

Apply script = assembly language program

Isar proof = structured program with assertions

Proofs Using Isabelle/Isar

proof
  assume $formula_0$
  have $formula_1$ **by** $method_1$
  ⋮
  have $formula_n$ **by** $method_n$
  show $formula_{n+1}$ **by** ...
qed

proves $formula_0 \Rightarrow formula_{n+1}$

Apply script = assembly language program

Isar proof = structured program with assertions

But: **apply** still useful for proof exploration

Type *'a list*

Lists of elements of type $'a$

Type *'a list*

Lists of elements of type $'a$

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

Type *'a list*

Lists of elements of type $'a$

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

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

Type *'a list*

Lists of elements of type $'a$

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

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

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 $P(x \# xs)$.

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

Download

https://www21.in.tum.de/teaching/fds/SS22/assets/Demos/List_Demo.thy

Download `Chapter2.thy` from

http://concrete-semantics.org/Exercises/templates.tar

and try to (re-)do exercises up to summation formula.