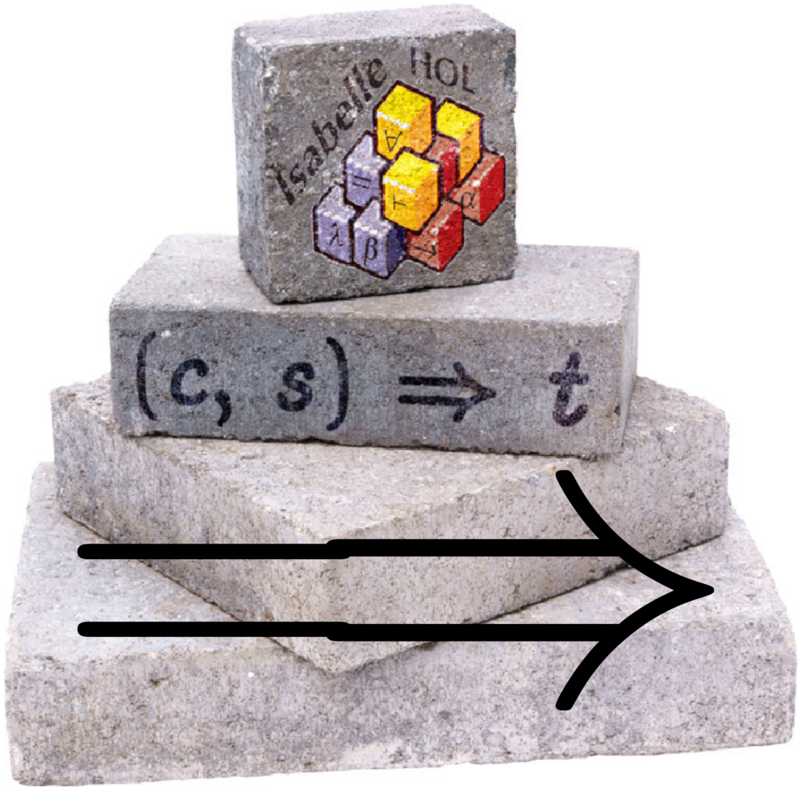
Λ $\Longrightarrow$ λ



Karlsruhe Institute of Technology

Higher-Order Logics and Interactive Theorem Proving with Isabelle/HOL

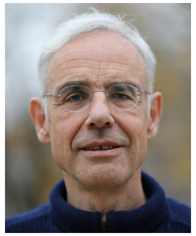
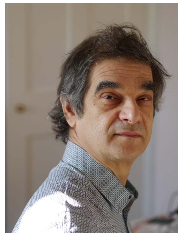
Formal Systems II: Application

Michael Kirsten | Summer Term 2025

KASTEL

Credits

Most material (originally) shamelessly stolen from

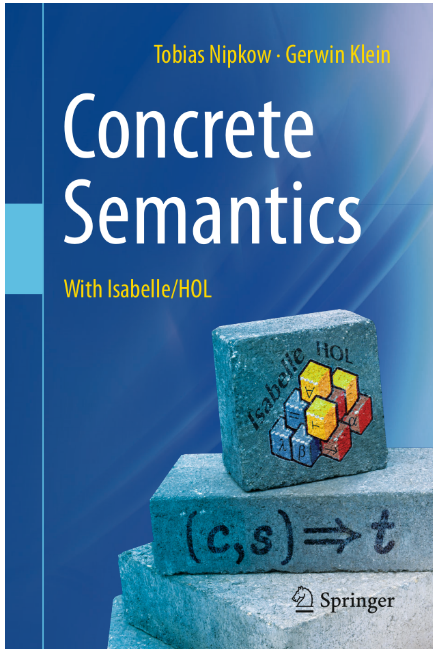
Tobias Nipkow, Lawrence Paulson, Makarius Wenzel

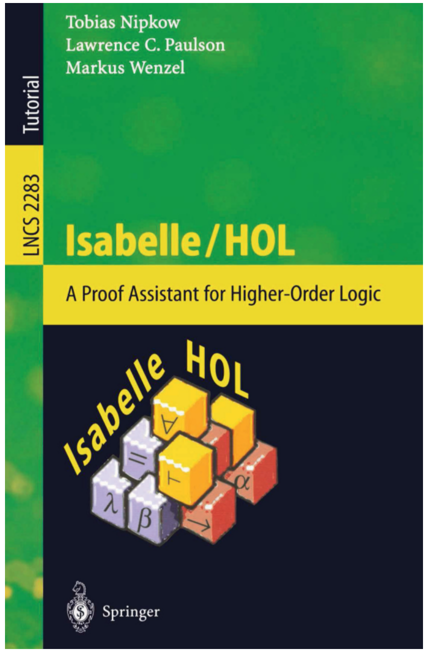
Gerwin Klein, John Harrison, Christian Sternagel, Chelsea Edmonds

Don’t blame them, errors are mine!

Literature

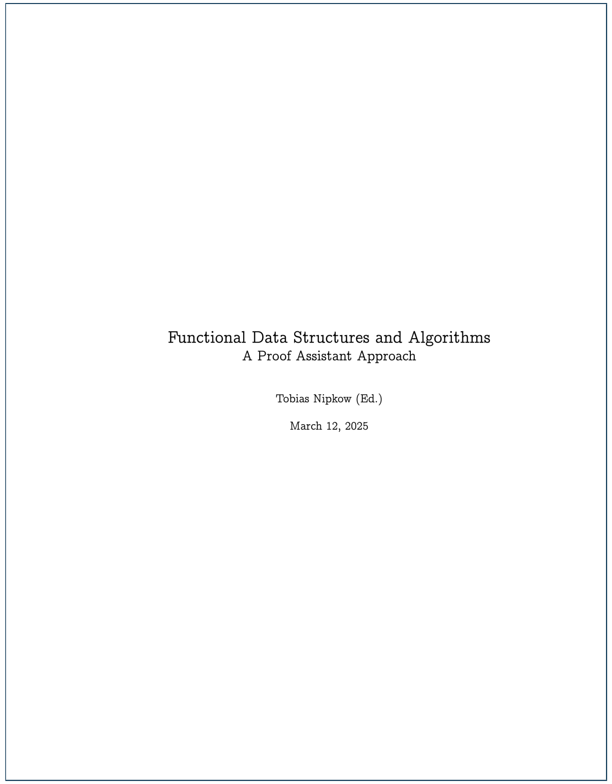


Tobias Nipkow and Gerwin Klein:
Concrete Semantics:
With Isabelle/HOL
Springer International Publishing,
2014
`http://concrete-semantics.org/`



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.)

March 12, 2025

Tobias Nipkow (Editor):
Functional Data Structures and Algorithms:
A Proof Assistant Approach
Under Development, 2025
`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?

What is a Theorem Prover?

Implementation of a formal logic on a computer.

Institute of Information Security
and Dependability (KASTEL)

Karlsruhe Institute of Technology

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, Rocq, HOL Light, HOL4, ProofPower, HOL Zero, IMPS, Isabelle, Metamath, Mizar, Nuprl, 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, Rocq, HOL Light, HOL4, ProofPower, HOL Zero, IMPS, Isabelle, Metamath, Mizar, Nuprl, PVS, Lean, …

There are other (algorithmic) verification tools:

- Model checking, static analysis, …
- Usually do not deliver proofs

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 Rocq

Some Impressive Verifications

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

Xavier Leroy (& Co)
INRIA Paris
using Rocq

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 Rocq

Operating system
microkernel (seL4),
Used in safety-critical applications
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 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

Some Impressive Verifications

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)

Some Impressive Verifications

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

Some Impressive Verifications

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

Choice of Foundations

Usually balancing simplicity against flexibility/expressiveness

Institute of Information Security
and Dependability (KASTEL)

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: Rocq, Lean
- Homotopy type theory (HoTT): Arend
- 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: Rocq, Lean
- Homotopy type theory (HoTT): Arend
- Other typed formalisms: IMPS, PVS

Others

- Primitive recursive arithmetic (no explicit quantifiers): ACL2, NQTHM

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)

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

Institute of Information Security and Dependability (KASTEL)

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 (originally *Logic of Computable Functions* by R. Milner, 1979):**
  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, Rocq

Higher-Order Logic (HOL)

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

Higher-Order Logic (HOL)

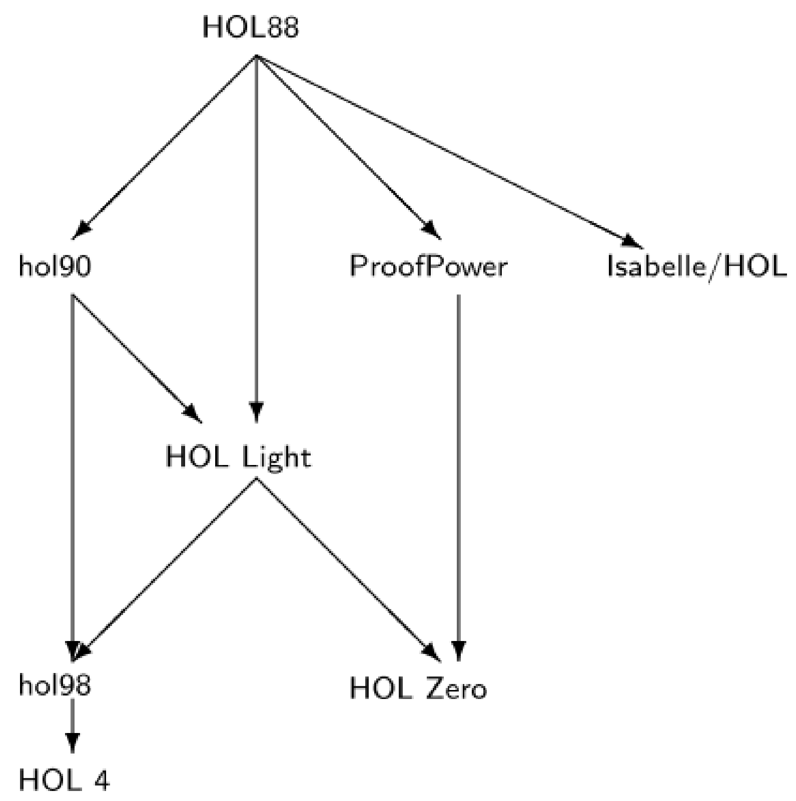
- First order logic extended with functions and sets, i.e.,
  functions are values, too!
- Polymorphic types, including truth value type
- No distinction between terms and formulas
- ML-style functional programming

“HOL = functional programming + logic”

Higher-Order Logic (HOL)

- First order logic extended with functions and sets, i.e.,
  functions are values, too!
- Polymorphic types, including 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 only use HOL in this lecture)

What is Isabelle?

A generic interactive proof assistant

Generic:
Not specialized to one particular logic
(two large developments: HOL and ZF, will only use HOL in this lecture)

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 only use HOL in this lecture)

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

Institute of Information Security
and Dependability (KASTEL)

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!

Institute of Information Security
and Dependability (KASTEL)

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**
- TeXLive (only) for document generation
- Download from `https://isabelle.in.tum.de` with lots of documentation
- Browse `https://isabelle.systems` for community, infrastructure, (more) resources, tools

Let us start with the basics ...

Which of the following three formulas have the same meaning?

1. $A \implies (B \implies C)$
2. $(A \implies B) \implies C$
3. $(A \land B) \implies C$

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 $[\![A; B]\!] \Longrightarrow C$
- $\dfrac{A_1 \quad \ldots \quad A_n}{C}$ means $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow C$

Institute of Information Security and Dependability (KASTEL)

Basic Syntax

Types:

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

Basic Syntax

Types:

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

Terms:

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

Institute of Information Security and Dependability (KASTEL)

Basic Syntax

Types:

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

Terms:

... must be well-typed!

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

Institute of Information Security and Dependability (KASTEL)

Some More Basics

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

Institute of Information Security and Dependability (KASTEL)

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* | *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* | *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 __ , …

Institute of Information Security and Dependability (KASTEL)

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* | *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 __ , …

Institute of Information Security and Dependability (KASTEL)

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* $|$ *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` $T_1 \ldots T_n$
`begin`
(definitions, theorems, proofs, …)*
`end`

Institute of Information Security
and Dependability (KASTEL)
Karlsruhe Institute of Technology

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`

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 (directly) *imported* theories (imports are transitive)

Institute of Information Security and Dependability (KASTEL)

Theory = Isabelle Module

Syntax:

```
theory MyTh
imports T_1 ... T_n
begin
(definitions, theorems, proofs, ...)*
end
```

MyTh: Name of theory. Must live in file *MyTh*`.thy`

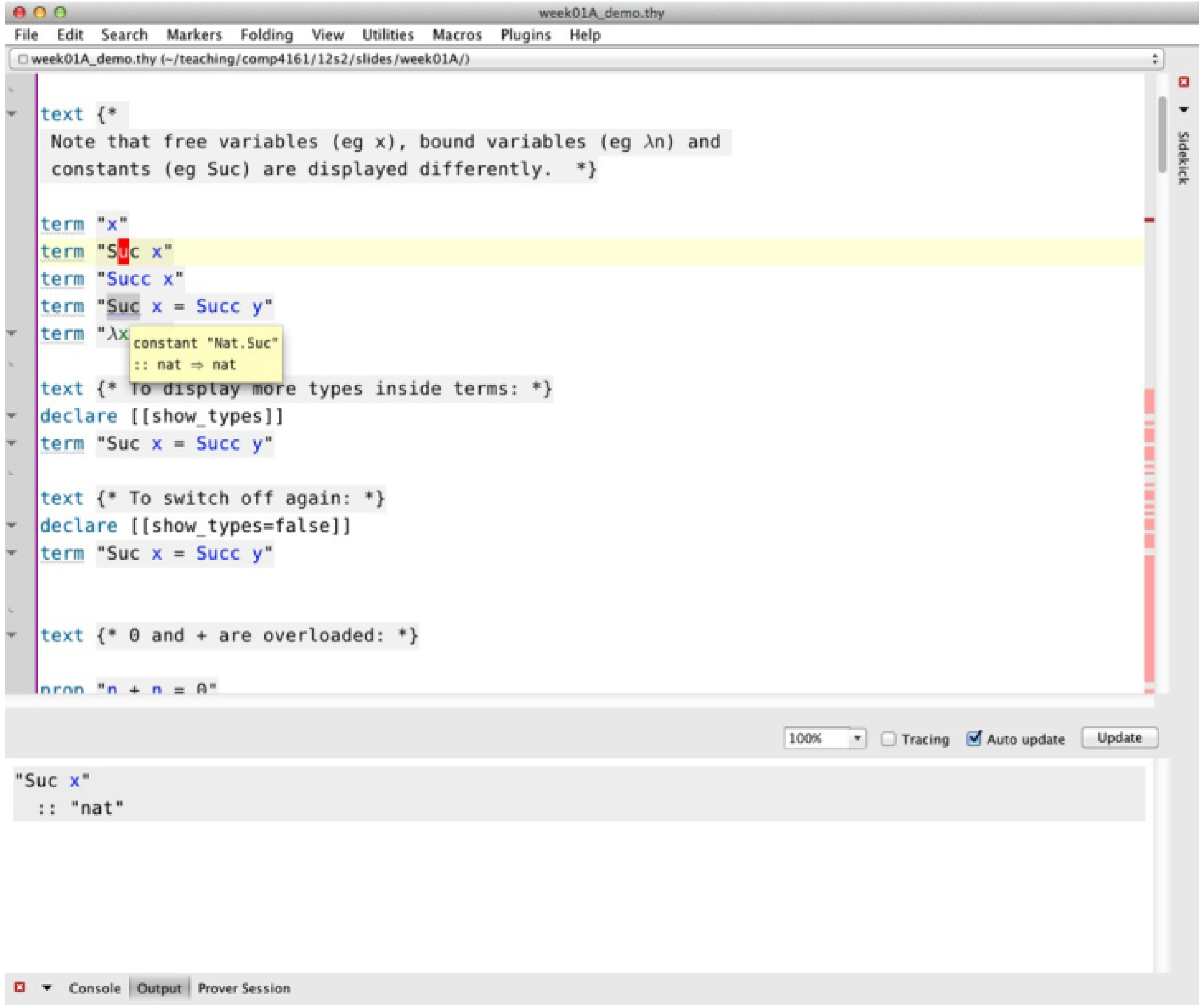
T_i: Names of (directly) *imported* theories (imports are transitive)

Usually: `imports Main`

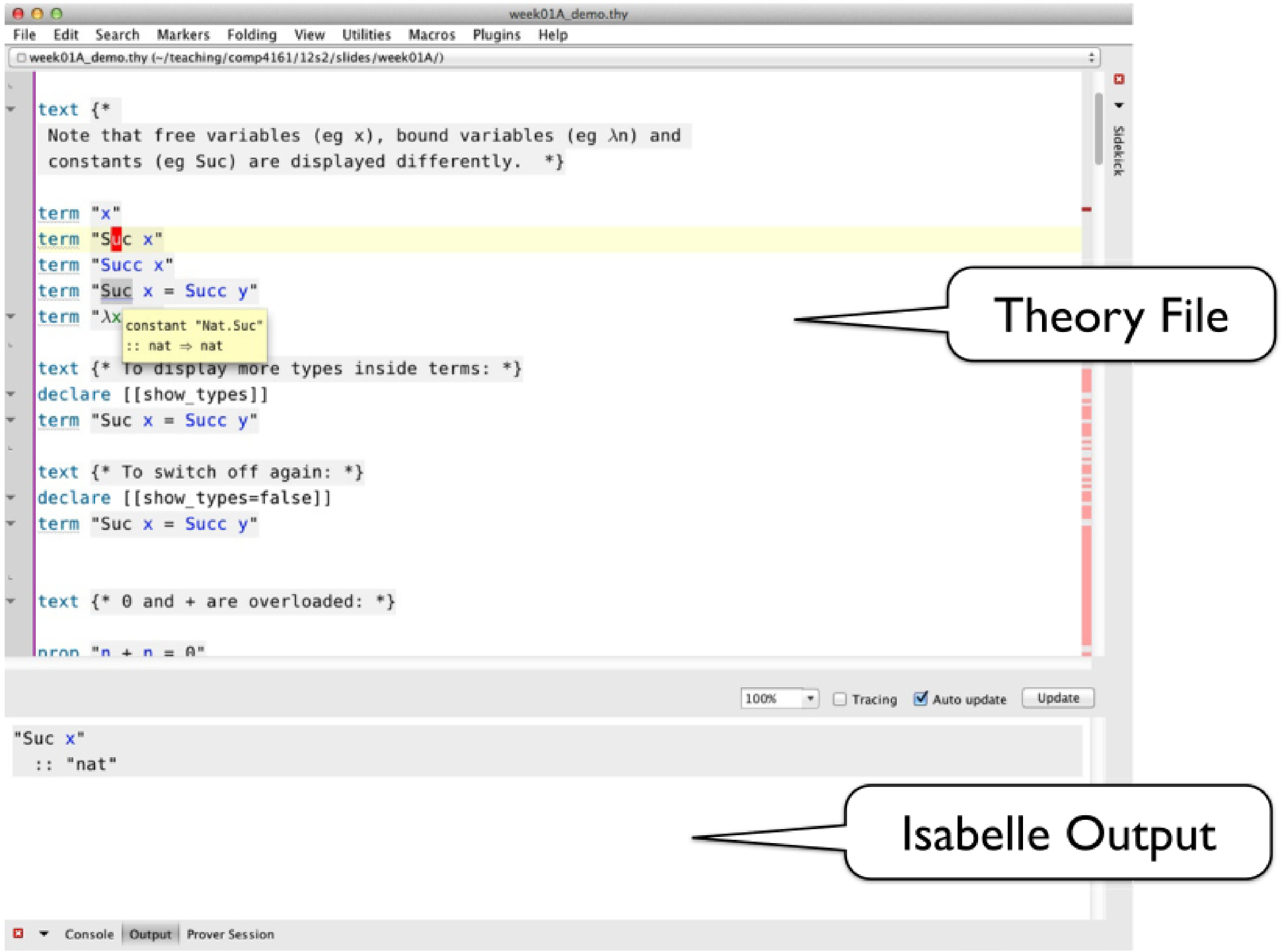
Download

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

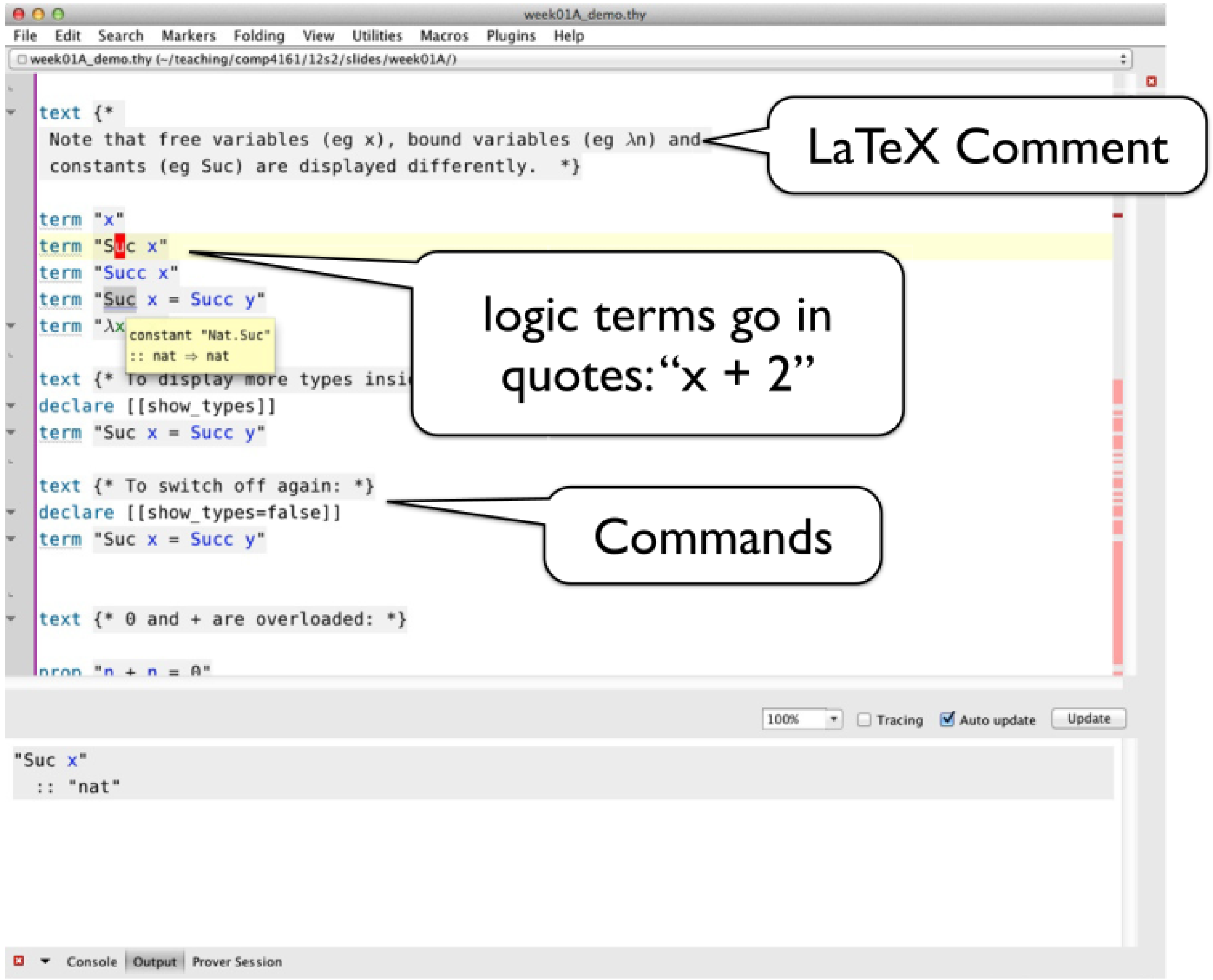
Isabelle jEdit/PIDE



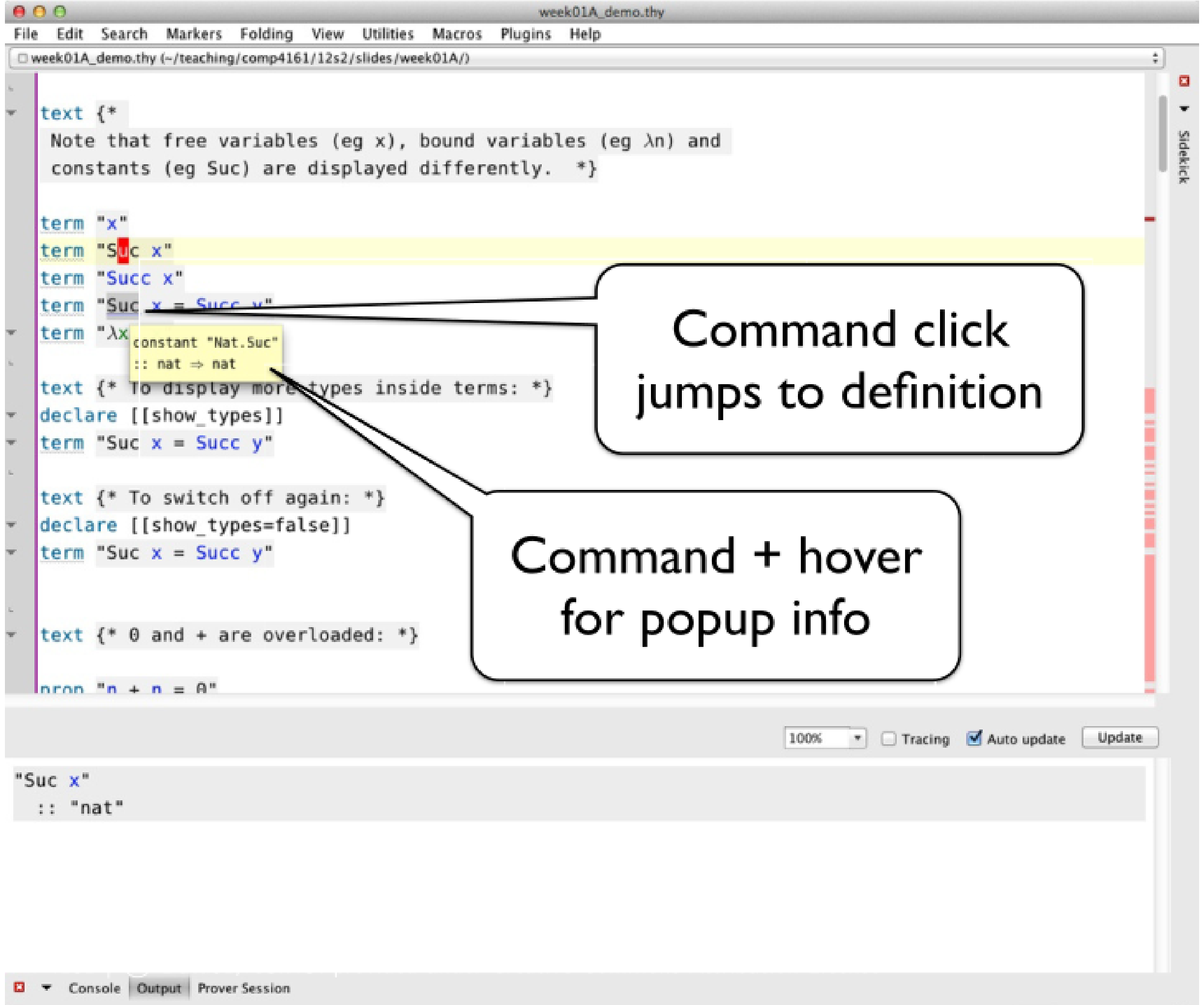
Isabelle jEdit/PIDE



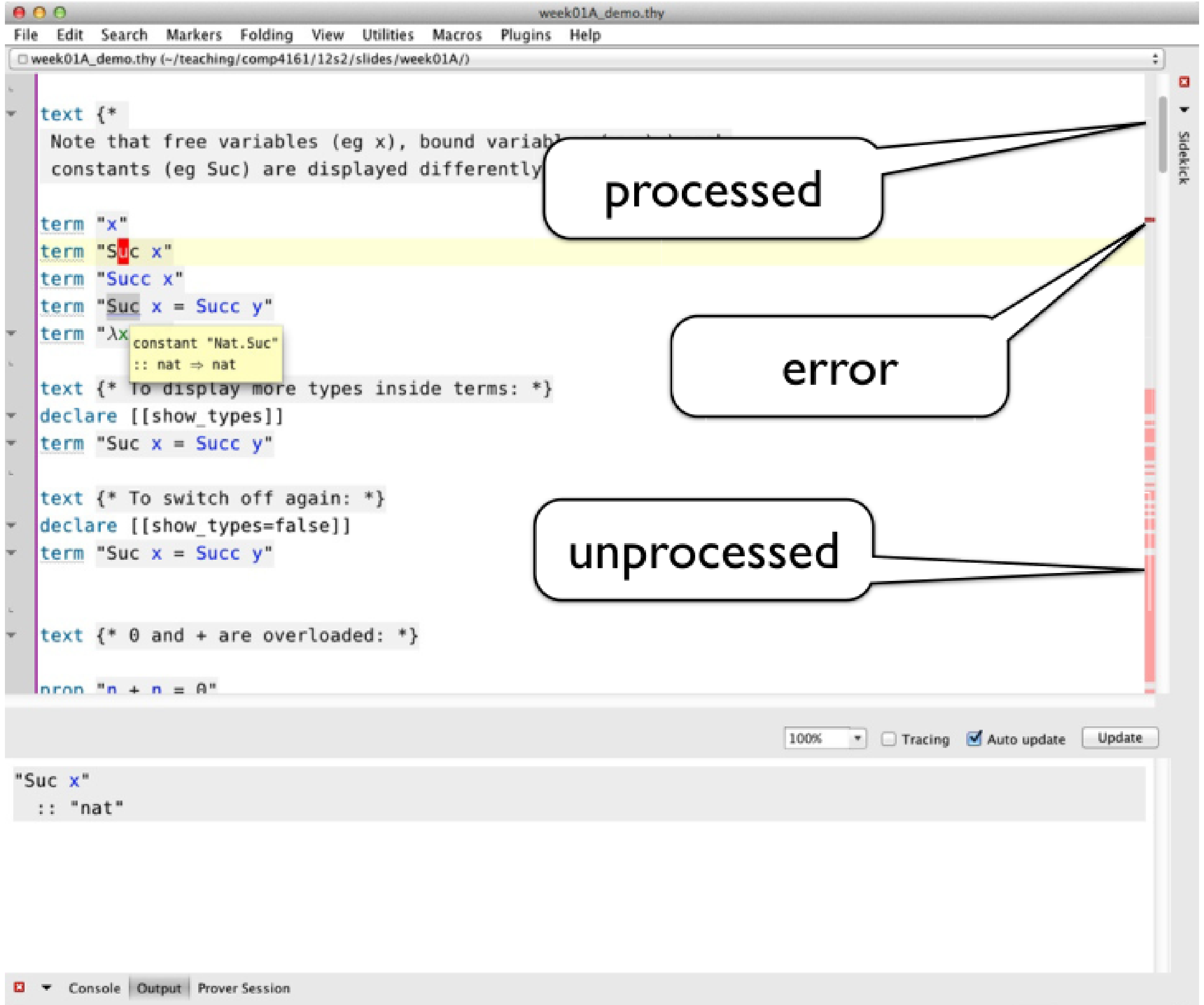
Isabelle jEdit/PIDE



Isabelle jEdit/PIDE



Isabelle jEdit/PIDE



Exploring a Theory in Isabelle

Some Diagnostic Commands

find_theorems ⟨*args*⟩	print all theorems matching ⟨*args*⟩
print_cases	print currently available cases
prop ⟨*formula*⟩	print proposition ⟨*formula*⟩
term ⟨*term*⟩	print term ⟨*term*⟩ and its type
thm ⟨*name*⟩	print theorem called ⟨*name*⟩
typ ⟨*type*⟩	print type ⟨*type*⟩
value ⟨*term*⟩	evaluate and print ⟨*term*⟩

Exploring a Theory in Isabelle

Some Diagnostic Commands

find_theorems ⟨*args*⟩	print all theorems matching ⟨*args*⟩
print_cases	print currently available cases
prop ⟨*formula*⟩	print proposition ⟨*formula*⟩
term ⟨*term*⟩	print term ⟨*term*⟩ and its type
thm ⟨*name*⟩	print theorem called ⟨*name*⟩
typ ⟨*type*⟩	print type ⟨*type*⟩
value ⟨*term*⟩	evaluate and print ⟨*term*⟩

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

Type *nat*

datatype $nat = 0 \mid Suc\ nat$

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

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

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

Institute of Information Security and Dependability (KASTEL)

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 $\mathit{formula}_0$
  have $\mathit{formula}_1$ **by** method_1
  $\vdots$
  have $\mathit{formula}_n$ **by** method_n
  show $\mathit{formula}_{n+1}$ **by** $\ldots$
qed

proves $\mathit{formula}_0 \Rightarrow \mathit{formula}_{n+1}$

Proofs Using Isabelle/Isar

proof
  assume *formula*$_0$
  have *formula*$_1$ **by** *method*$_1$
  $\vdots$
  have *formula*$_n$ **by** *method*$_n$
  show *formula*$_{n+1}$ **by** . . .
qed

proves $\mathit{formula}_0 \Rightarrow \mathit{formula}_{n+1}$

Apply script = assembly language program

Isar proof = structured program with assertions

Proofs Using Isabelle/Isar

proof
  assume $\mathit{formula}_0$
  have $\mathit{formula}_1$ **by** method_1
  $\vdots$
  have $\mathit{formula}_n$ **by** method_n
  show $\mathit{formula}_{n+1}$ **by** $\ldots$
qed

proves $\mathit{formula}_0 \Rightarrow \mathit{formula}_{n+1}$

Apply script = assembly language program

Isar proof = structured program with assertions

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

Institute of Information Security and Dependability (KASTEL)

Type *'a list*

Lists of elements of type $'a$

Type *'a list*

Lists of elements of type $'a$

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

Type *'a list*

Lists of elements of type $'a$

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

Some lists: Nil, $\mathit{Cons}\ 1\ \mathit{Nil}$, $\mathit{Cons}\ 1\ (\mathit{Cons}\ 1\ \mathit{Nil})$, …

Type *'a list*

Lists of elements of type $'a$

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

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 $P(x \# xs)$.

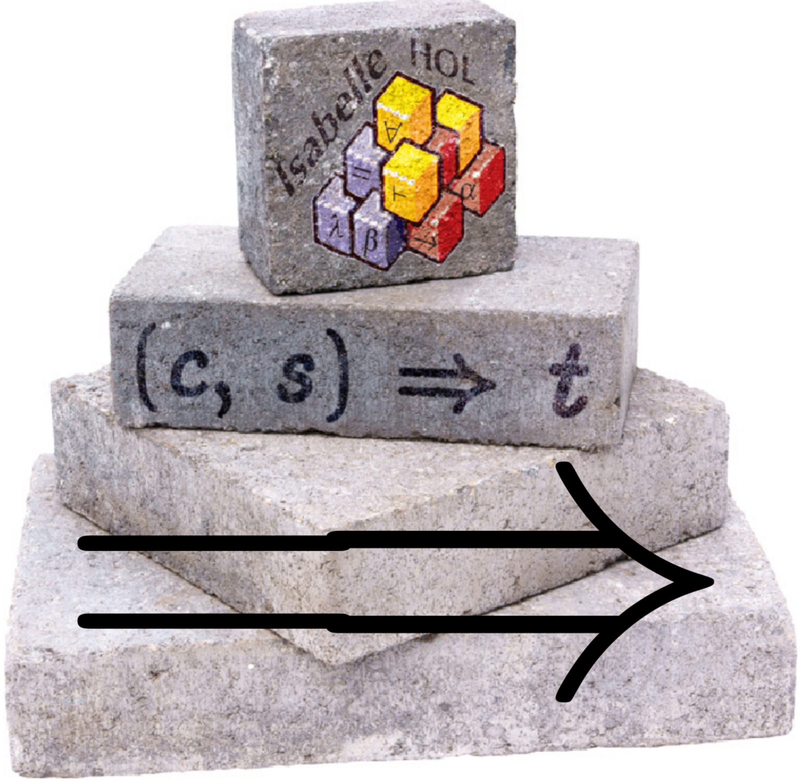
$$\frac{P([]) \qquad \bigwedge x\ xs.\ P(xs) \Longrightarrow P(x \# xs)}{P(xs)}$$

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 (line 120).

Λ



$\Longrightarrow$ λ

Higher-Order Logics and Interactive Theorem Proving with Isabelle/HOL II

Formal Systems II: Application

Michael Kirsten | Summer Term 2025

More on Function Definitions

Non-recursive definitions

- **definition** $sq :: nat \Rightarrow nat$ **where** $sq n = n * n$
- No pattern matching, just $f x_1 \ldots x_n = \ldots$

More on Function Definitions

Non-recursive definitions

- **definition** $sq :: nat \Rightarrow nat$ **where** $sqn = n * n$
- No pattern matching, just $fx_1 \ldots x_n = \ldots$

(Possibly) recursive functions: fun

- Pattern-matching over datatype constructors
- Order of equation matters
- Termination must be provable automatically by size measures
- Proves customized induction schema
- Example: **fun** $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**

$$ack\ 0\ n = Suc\ n\ |$$
$$ack\ (Suc\ m)\ 0 = ack\ m\ (Suc\ 0)\ |$$
$$ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)$$

Proof method *simp*

Goal: 1. $\llbracket\ P_1; \ldots; P_m\ \rrbracket \Longrightarrow C$

apply$(simp\ add\!:\ eq_1\ \ldots\ eq_n)$

Simplify $P_1\ \ldots\ P_m$ and C using

- lemmas with attribute $simp$
- rules from **fun** and **datatype**
- additional lemmas $eq_1\ \ldots\ eq_n$
- assumptions $P_1\ \ldots\ P_m$

Variations:

- $(simp\ \ldots\ del\!:\ \ldots)$ removes $simp$-lemmas
- add and del are optional

auto versus *simp*

- $auto$ acts on all subgoals
- $simp$ acts only on subgoal 1

- $auto$ applies $simp$ and more

- $auto$ can also be modified:

  $(auto\ simp\ add{:}\ \ldots\ \ simp\ del{:}\ \ldots)$

Rewriting with definitions

Definitions (**definition**) must be used explicitly:

$$(\mathit{simp\ add}\colon f\_def \ldots)$$

f is the function whose definition is to be unfolded.

Case Splitting with *simp/auto*

Automatic:

$$P\ (\textit{if}\ A\ \textit{then}\ s\ \textit{else}\ t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

By hand:

$$P\ (\textit{case}\ e\ \textit{of}\ 0 \Rightarrow a \mid Suc\ n \Rightarrow b)$$
$$=$$
$$(e = 0 \longrightarrow P(a)) \wedge (\forall\, n.\ e = Suc\ n \longrightarrow P(b))$$

Proof method: $(simp\ \ split\colon\ nat.split)$
Or $auto$. Similar for any datatype t: $t.split$

More Proof Methods

intro ⟨*intro-rules*⟩	repeatedly applies introduction rules
elim ⟨*elim-rules*⟩	repeatedly applies elimination rules
clarify	applies all safe rules that do not split the goal
safe	applies all safe rules
blast	an automatic tableaux prover (works well on predicate logic)
fast	another automatic search tactic
fastforce	rewriting, logic, sets, relations and a bit of arithmetic
arith	proves linear formulas (no "$*$"), complete for quantifier-free real and Presburger arithmetic

Download

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

$\Longrightarrow$ versus $\longrightarrow$

$\Longrightarrow$ is part of the Isabelle framework. It structures theorems and proof states: $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$

$\longrightarrow$ is part of HOL and can occur inside the logical formulas A_i and A.

Phrase theorems like this $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$

not like this $A_1 \wedge \ldots \wedge A_n \longrightarrow A$

Sets over type $'a$

$$'a\ set$$

- $\{\}$, $\{e_1, \ldots, e_n\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$
- $\ldots$

$\in$	\<in>	:
$\subseteq$	\<subseteq>	<=
$\cup$	\<union>	Un
$\cap$	\<inter>	Int

Set Comprehension

- $\{x.\ P\}$ where x is a variable
- But not $\{t.\ P\}$ where t is a proper term
- Instead: $\{t\ |x\ y\ z.\ P\}$
  is short for $\{v.\ \exists\, x\ y\ z.\ v = t \land P\}$
  where x, y, z are the free variables in t

Isar Core Syntax

$$
\begin{array}{lcl}
\text{proof} & = & \textbf{proof}\ [\text{method}]\ \ \text{step}^*\ \ \textbf{qed} \\
 & | & \textbf{by}\ \text{method}
\end{array}
$$

$$
\text{method} \;=\; (\mathit{simp}\ \ldots) \mid (\mathit{blast}\ \ldots) \mid (\mathit{induction}\ \ldots)
$$

$$
\begin{array}{lcll}
\text{step} & = & \textbf{fix}\ \text{variables} & (\bigwedge) \\
 & | & \textbf{assume}\ \text{prop} & (\Longrightarrow) \\
 & | & [\textbf{from}\ \text{fact}^+]\ \ (\textbf{have} \mid \textbf{show})\ \text{prop}\ \ \text{proof} &
\end{array}
$$

$$
\text{prop} \;=\; [\text{name:}]\ \text{"formula"}
$$

$$
\text{fact} \;=\; \text{name} \mid \ldots
$$

Example: Cantor's Theorem

lemma $\neg\ surj(f :: {'a} \Rightarrow {'a}\ set)$
proof default proof: assume *surj*, show *False*
  assume a: $surj\ f$
  from a **have** b: $\forall\ A.\ \exists\ a.\ A = f\ a$
    by$(simp\ add\!: surj\_def)$
  from b **have** c: $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$
    by $blast$
  from c **show** $False$
    by $blast$
qed

Download

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

Abbreviations

this	$=$	the previous proposition proved or assumed
then	$=$	**from** this
thus	$=$	**then show**
hence	$=$	**then have**

using and with

(**have**|**show**) prop **using** facts

=

from facts (**have**|**show**) prop

with facts

=

from facts *this*

Institute of Information Security
and Dependability (KASTEL)

Structured Lemma Statement

lemma
  fixes $f :: 'a \Rightarrow 'a\ set$
  assumes s: $surj\ f$
  shows $False$
proof $-$ no automatic proof step
  have $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$ **using** s
    by$(auto\ simp{:}\ surj\_def)$
  thus $False$ **by** $blast$
qed

Proves $surj\ f \Longrightarrow False$
but $surj\ f$ *becomes local fact* s *in proof.*

The Essence of Structured Proofs

Assumptions and intermediate facts
can be named and referred to explicitly and selectively

Structured Lemma Statements

fixes $x :: \tau_1$ **and** $y :: \tau_2 \ldots$
assumes *a*: P **and** *b*: $Q \ldots$
shows R

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**

Case Distinction

show R
proof $cases$
  assume P
  $\vdots$
  show R $\langle proof \rangle$
next
  assume $\neg\ P$
  $\vdots$
  show R $\langle proof \rangle$
qed

have $P \vee\ Q$ $\langle proof \rangle$
then show R
proof
  assume P
  $\vdots$
  show R $\langle proof \rangle$
next
  assume Q
  $\vdots$
  show R $\langle proof \rangle$
qed

Institute of Information Security and Dependability (KASTEL)

Set Equality and Subset

show $A = B$
proof
  show $A \subseteq B$ $\langle proof \rangle$
next
  show $B \subseteq A$ $\langle proof \rangle$
qed

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\langle proof \rangle$
qed

Sledgehammer

Architecture:

Isabelle

Goal
& filtered library ↓ ↑ Proof

external
ATPs[1]

Characteristics:

- Sometimes it works,
- sometimes it doesn't.

Do you feel lucky?

[1]Automatic Theorem Provers

Sledgehammer

Sledgehammer:

- ➜ *Connects Isabelle with ATPs and SMT solvers:*
  E, SPASS, Vampire, CVC3, Yices, Z3
- ➜ *Simple invocation:*
  - ➜ *Users don't need to select or know facts*
  - ➜ *or ensure the problem is first-order*
  - ➜ *or know anything about the automated prover*
- ➜ *Exploits local parallelism and remote servers*

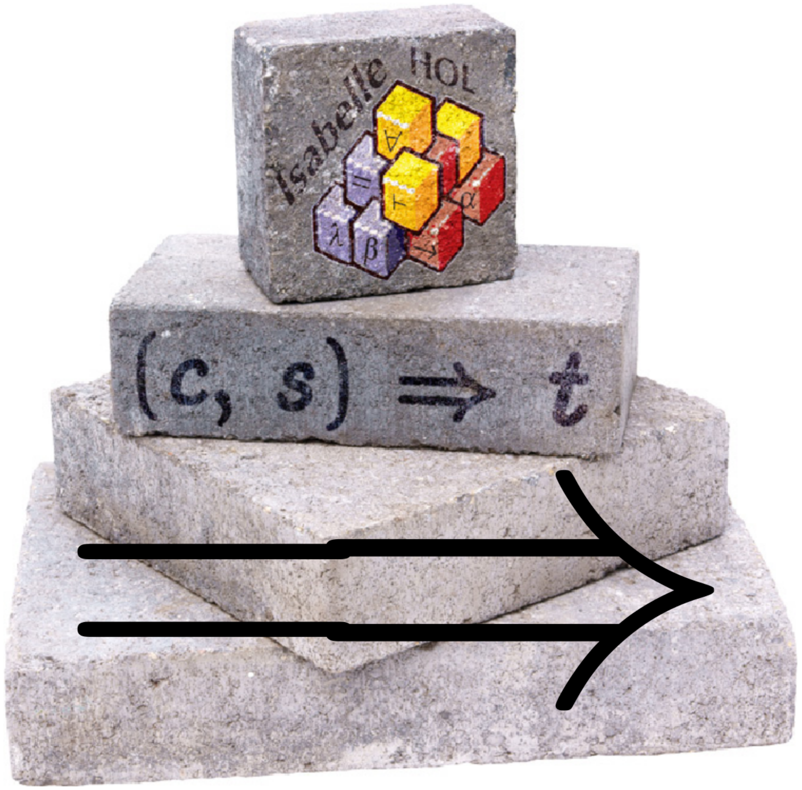
Download

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

Download `Chapter2.thy` from
`http://concrete-semantics.org/Exercises/templates.tar`
and try to do some more exercises.

KIT
Karlsruhe Institute of Technology

Λ



λ

Higher-Order Logics and Interactive Theorem Proving with Isabelle/HOL III

Formal Systems II: Application

Michael Kirsten | Summer Term 2025

KASTEL

Back to HOL

Base: $\quad bool, \Rightarrow, ind \qquad =, \longrightarrow, \varepsilon$

And the rest is definitions:

$$
\begin{array}{lcl}
\text{True} & \equiv & (\lambda x :: bool.\ x) = (\lambda x.\ x) \\
\text{All}\ P & \equiv & P = (\lambda x.\ \text{True}) \\
\text{Ex}\ P & \equiv & \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q \\
\text{False} & \equiv & \forall P.\ P \\
\neg P & \equiv & P \longrightarrow \text{False} \\
P \wedge Q & \equiv & \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R \\
P \vee Q & \equiv & \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R \\
\text{If}\ P\ x\ y & \equiv & \text{SOME}\ z.\ (P = \text{True} \longrightarrow z = x) \wedge (P = \text{False} \longrightarrow z = y) \\
\text{inj}\ f & \equiv & \forall x\ y.\ f\ x = f\ y \longrightarrow x = y \\
\text{surj}\ f & \equiv & \forall y.\ \exists x.\ y = f\ x
\end{array}
$$

The Axioms of HOL

$$\frac{}{t = t}\ \text{refl} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst} \qquad \frac{\bigwedge x.\ f\ x = g\ x}{(\lambda x.\ f\ x) = (\lambda x.\ g\ x)}\ \text{ext}$$

$$\frac{P \Longrightarrow Q}{P \longrightarrow Q}\ \text{impl} \qquad \frac{P \longrightarrow Q \quad P}{Q}\ \text{mp}$$

$$\frac{}{(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)}\ \text{iff}$$

$$\frac{}{P = \text{True} \lor P = \text{False}}\ \text{True\_or\_False}$$

$$\frac{P\ ?x}{P\ (\text{SOME}\ x.\ P\ x)}\ \text{someI}$$

$$\frac{}{\exists f :: \mathit{ind} \Rightarrow \mathit{ind}.\ \text{inj}\ f \land \neg \text{surj}\ f}\ \text{infty}$$

That's it.

- ➜ 3 basic constants
- ➜ 3 basic types
- ➜ 9 axioms

With this you can define and derive all the rest.

Isabelle knows 2 more axioms:

$$\frac{x = y}{x \equiv y}\ \text{eq\_reflection} \qquad \frac{}{(\text{THE } x.\ x = a) = a}\ \text{the\_eq\_trivial}$$

Institute of Information Security and Dependability (KASTEL)

Induction: Example for Even Numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

inductive $ev :: nat \Rightarrow bool$
where
$ev\ 0 \quad |$
$ev\ n \Longrightarrow ev\ (n + 2)$

An easy proof: $ev\ 4$

$$ev\ 0 \Longrightarrow ev\ 2 \Longrightarrow ev\ 4$$

Consider

fun $evn :: nat \Rightarrow bool$ **where**
$evn\ 0 = True\ |$
$evn\ (Suc\ 0) = False\ |$
$evn\ (Suc\ (Suc\ n)) = evn\ n$

A trickier proof: $ev\ m \Longrightarrow evn\ m$

By induction on the *structure* of the derivation of $ev\ m$

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow evn\ m = True$
- rule $ev\ n \Longrightarrow ev\ (n+2)$
  $\Longrightarrow m = n+2$ and $evn\ n$ (IH)
  $\Longrightarrow evn\ m = evn\ (n+2) = evn\ n = True$

Rule induction for *ev*

To prove

$$ev\ n \Longrightarrow P\ n$$

by *rule induction* on $ev\ n$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n{+}2)$

Rule `ev.induct`:

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n.\ [\![\ ev\ n;\ P\ n\ ]\!] \Longrightarrow P(n{+}2)}{P\ n}$$

Format of Inductive Definitions

inductive $I :: \tau \Rightarrow bool$ **where**

$$\llbracket\ I\ a_1;\ \ldots\ ;\ I\ a_n\ \rrbracket \Longrightarrow I\ a \quad |$$

$$\vdots$$

Note:

- I may have multiple arguments.
- Each rule may also contain *side conditions* not involving I.

Institute of Information Security and Dependability (KASTEL)

Rule Induction in General

To prove

$$I\ x \Longrightarrow P\ x$$

by *rule induction* on $I\ x$
we must prove for every rule

$$[\![\ I\ a_1;\ \dots\ ;\ I\ a_n\ ]\!] \Longrightarrow I\ a$$

that P is preserved:

$$[\![\ I\ a_1;\ P\ a_1;\ \dots\ ;\ I\ a_n;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

Inductively Defined Set

inductive_set $I :: \tau \ set$ **where**

$\llbracket \ a_1 \in I; \ \ldots \ ; a_n \in I \ \rrbracket \Longrightarrow a \in I \ \ |$

$\vdots$

Difference to **inductive**:

- arguments of I are tupled, not curried
- I can later be used with set theoretic operators, eg $\ I \cup \ldots$

Named Assumptions

In a proof of

$$I \ldots \implies A_1 \implies \ldots \implies A_n \implies B$$

by rule induction on $I \ldots$:

In the context of

case R

we have

$R.IH$	the induction hypotheses
$R.hyps$	the assumptions of rule R
$R.prems$	the premises A_i
R	$R.IH + R.hyps + R.prems$

Rule Induction

inductive $I :: \tau \Rightarrow \sigma \Rightarrow bool$
where
$rule_1$: $\ldots$
$\vdots$
$rule_n$: $\ldots$

show $I\ x\ y \Longrightarrow P\ x\ y$
proof $(induction\ rule{:}\ I.induct)$
  case $rule_1$
  $\ldots$
  show $?case$
next
$\vdots$
next
  case $rule_n$
  $\ldots$
  show $?case$
qed

Style Remark

- **case** $(Suc\ n)\ \ldots$ **show** $?case$
  is easy to write and maintain
- **fix** n **assume** $formula\ \ldots$ **show** $formula'$
  is easier to read:
  - all information is shown locally
  - no contextual references (e.g. $?case$)

More on (Data) Types

- One common use case of datatypes is an option datatype

```
datatype 'a option = None | Some 'a
```

- Datatypes can be parameterised by multiple types:

```
datatype ('a, 'b, 'c) three = Three 'a 'b 'c
```

- Datatypes can also be annotated:

```
datatype (set: 'a) mylist =
    null: mNil  (‹[]›)
  | mCons (mhd: 'a) (mtl: "'a mylist")  (infixr ‹##› 65)
```

Infix Notation

Discriminator

Constructors

Selectors

- The datatypes (and co-datatypes) tutorial has significantly more information.

Type Synonyms, Declarations, and Definitions

- A type synonym can be useful to make a formalisation more readable/descriptive. E.g.

```
type_synonym 'a edge = "'a set"
```

  - declares a parameterised edge type which is the same as a set
- A type declaration declares a new type without defining it

```
typedecl Test
```

- A type definition allows you to define a new type

```
typedef three = "{0:: nat, 1, 2}"
  apply (intro exI[of _ 0]) (* Goal must show RHS is non-empty *)
  by simp
```

  - You must prove the type is not empty
  - Introduces Rep and Abs properties to convert between reasoning on base type and new type (then you need to establish useful properties)...
  - Or in this case just use a datatype which does the setup for you!

Application: Programming Language Semantics

Language with only arithmetic expressions and assignments:

```
a ::= 6;; b ::= 7;; x = a + b
```


Institute of Information Security
and Dependability (KASTEL)

Application: Programming Language Semantics

Language with only arithmetic expressions and assignments:

```
a ::= 6;; b ::= 7;; x = a + b
```

```
type_synonym vname = string
datatype aexp = Num int | Val vname | Plus aexp aexp

datatype
  com =  Assign vname aexp       (‹_ ::= _› [1000, 61] 61)
      | Seq    com  com          (‹_;;/ _›  [60, 61] 60)
```


Institute of Information Security and Dependability (KASTEL)

Application: Programming Language Semantics

Predicate $(c, s) \Rightarrow s'$: program c executed in state s yields state s'.

```
type_synonym val = int
type_synonym state = "vname ⇒ val"

fun aval :: "aexp ⇒ state ⇒ val" where
"aval (Num n) s = n" |
"aval (Val x) s = s x" |
"aval (Plus a₁ a₂) s = aval a₁ s + aval a₂ s"

inductive
  big_step :: "com × state ⇒ state ⇒ bool" (infix ‹⇒› 55) where
  Assign: "(x ::= a,s) ⇒ s(x := aval a s)" |
  Seq:    "⟦ (c₁,s₁) ⇒ s₂;  (c₂,s₂) ⇒ s₃ ⟧ ⟹ (c₁;;c₂, s₁) ⇒ s₃"
```


Institute of Information Security and Dependability (KASTEL)

Application: Toy Assembly and Compiler

Simple assembly language with four instructions:

```
datatype instr =
  LOADI int | LOAD vname | ADD | STORE vname

fun acomp :: "aexp ⇒ instr list" where
"acomp (Num n) = [LOADI n]" |
"acomp (Val x) = [LOAD x]" |
"acomp (Plus a1 a2) = acomp a1 @ acomp a2 @ [ADD]"

fun ccomp :: "com ⇒ instr list" where
"ccomp (x ::= a) = acomp a @ [STORE x]" |
"ccomp (c1;;c2) = ccomp c1 @ ccomp c2"
```

Application: Execution

Predicate: $instr \vdash (i, s, stk) \rightarrow^{\star} (i', s', stk')$

"*instr* executed on *stk* with state *s* at instruction counter *i* leads to state s' on stk' with new instruction counter i'."

```
type_synonym stack = "val list"
type_synonym config = "int × state × stack"

fun iexec :: "instr ⇒ config ⇒ config" where
"iexec instr (i,s,stk) = (case instr of
  LOADI n ⇒ (i+1,s, n#stk) |
  LOAD x ⇒ (i+1,s, s x # stk) |
  ADD ⇒ (i+1,s, (hd2 stk + hd stk) # tl2 stk) |
  STORE x ⇒ (i+1,s(x := hd stk),tl stk))"
```


Institute of Information Security and Dependability (KASTEL)

IMP: A Small Imperative Language

Commands:

$$
\begin{array}{llll}
\textbf{datatype}\ \mathsf{com} & = & \mathsf{SKIP} & \\
 & \mid & \mathsf{Assign\ vname\ aexp} & (\_ := \_) \\
 & \mid & \mathsf{Semi\ com\ com} & (\_;\ \_) \\
 & \mid & \mathsf{Cond\ bexp\ com\ com} & (\mathsf{IF}\ \_\ \mathsf{THEN}\ \_\ \mathsf{ELSE}\ \_ \\
 & \mid & \mathsf{While\ bexp\ com} & (\mathsf{WHILE}\ \_\ \mathsf{DO}\ \_\ \mathsf{OD})
\end{array}
$$

$$
\begin{array}{lll}
\textbf{type\_synonym}\ \mathsf{vname} & = & \mathsf{string} \\
\textbf{type\_synonym}\ \mathsf{state} & = & \mathsf{vname} \Rightarrow \mathsf{nat}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{type\_synonym}\ \mathsf{aexp} & = & \mathsf{state} \Rightarrow \mathsf{nat} \\
\textbf{type\_synonym}\ \mathsf{bexp} & = & \mathsf{state} \Rightarrow \mathsf{bool}
\end{array}
$$

Example Program

Usual syntax:

$$\begin{array}{l} B := 1; \\ \text{WHILE } A \neq 0 \text{ DO} \\ \quad B := B * A; \\ \quad A := A - 1 \\ \text{OD} \end{array}$$

Expressions are functions from state to bool or nat:

$$\begin{array}{l} B := (\lambda\sigma.\ 1); \\ \text{WHILE } (\lambda\sigma.\ \sigma\ A \neq 0) \text{ DO} \\ \quad B := (\lambda\sigma.\ \sigma\ B * \sigma\ A); \\ \quad A := (\lambda\sigma.\ \sigma\ A - 1) \\ \text{OD} \end{array}$$

Structural Operational Semantics

$$\frac{}{\langle \mathsf{SKIP}, \sigma \rangle \to \sigma}$$

$$\frac{e\ \sigma = v}{\langle \mathsf{x := e}, \sigma \rangle \to \sigma[x \mapsto v]}$$

$$\frac{\langle c_1, \sigma \rangle \to \sigma' \quad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1; c_2, \sigma \rangle \to \sigma''}$$

$$\frac{b\ \sigma = \mathsf{True} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2, \sigma \rangle \to \sigma'}$$

$$\frac{b\ \sigma = \mathsf{False} \quad \langle c_2, \sigma \rangle \to \sigma'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2, \sigma \rangle \to \sigma'}$$

Structural Operational Semantics Cont'd

$$\frac{b\ \sigma = \text{False}}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \to \sigma}$$

$$\frac{b\ \sigma = \text{True} \quad \langle c, \sigma \rangle \to \sigma' \quad \langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma' \rangle \to \sigma''}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \to \sigma''}$$

Isabelle's Module System: Locales

- Locales are Isabelle's module system. From a logical perspective, they are simply persistent contexts.

$$\bigwedge x_1 \dots x_n.\ [\![A_1; \dots; A_m]\!] \Rightarrow C.$$

- Provides fixed type and term variables and contextual assumptions within a local context.
- Type classes use and can interact with the underlying locale infrastructure.

Class

```
locale semigroup_orig =
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 70)
  assumes assoc: "(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
```

```
class semigroup_orig_add = plus +
  assumes add_assoc: "(a + b) + c = a + (b + c)"
begin

sublocale add: semigroup_orig plus
  by standard (fact add_assoc)

end
```

Same params/assumptions as before

Locale inheritance

Locales Cont'd

- Locales allow us to work explicitly with "carrier sets" (if we want to)

Carrier set

```
locale semigroup =
  fixes M and composition (infixl "·" 70)
  assumes composition_closed [intro, simp]: "⟦ a ∈ M; b ∈ M ⟧ ⟹ a · b ∈ M"
  assumes assoc[intro]: "⟦ a ∈ M; b ∈ M; c ∈ M ⟧ ⟹ (a · b) · c = a · (b · c)"
```

- Think of locales as more of a set-based rather than type-based approach.

Interpreting a Locale

- Global theory interpretation:

Label interpretation

Locale being interpreted

```
interpretation ints: semigroup ℤ plus
  by unfold_locales  simp_all
```

locale tactic

Terms to “instantiate” locale parameters with

- Can also now use inherited locale properties outside locale context

```
lemma "(1 + 2) + (3 ::int) = 1 + (2 + 3)"
  using ints.assoc by simp
```

Must reference named interpretation

Institute of Information Security and Dependability (KASTEL)

Other Facets of Isabelle

- Document preparation:
  You can generate LaTeX documents from your theories
- Axiomatic type classes:
  A general approach to polymorphism and overloading when there are shared laws
- Code generation:
  You can generate executable code from the formal functional programs you have verified
  $\Rightarrow$ Algorithms can be verified and then executed (ML, Haskell, Scala, …)
- Archive of formal proofs:
  A massive collection of proven und useful theories, which you can extend!
- More on program verification:
  Imperative HOL and refinement to efficient compiler code using separation logic