University of Sheffield

LECTURE 1: MODULAR PROOFS IN ISABELLE HOL

CHELSEA EDMONDS | c.l.edmonds@sheffield.ac.uk

Midlands Graduate School 2025

University of Sheffield

COURSE OVERVIEW

A practical course on effective use of the Isabelle/HOL proof assistant in mathematics and programming languages

Lectures:

- **Introduction to Proof Assistants**
- **Formalising the basics in Isabelle/HOL**
- Introduction to Isar, more types, Locales and Type-classes
- Case studies:
  - Formalising Mathematics: Combinatorics & advanced locale reasoning patterns
  - Program Verification: Formalising semantics, program properties, and introducing modularity/abstraction.

Example Classes:

- Isabelle exercises based on the previous lecture
- Will be drawing from the existing Isabelle tutorials/Nipkow's Concrete Semantic Book, as well as custom exercises (e.g. for locales).

Acknowledgement: Slides partially inspired by slides/notes by Larry Paulson, Tobias Nipkow, Gerwin Klein, Clemens Ballarin, Georg Struth, Andrei Popescu (and many more who've come before me!)

PRE-REQUISITE KNOWLEDGE

- No prior proof assistance is assumed:
  - If you've used Isabelle before, perhaps this will offer a new perspective/closer look at certain features
  - If you've used other proof assistants before, there'll be plenty of Isabelle specific concepts as well as more familiar ones.
  - We'll discuss topics that are both Isabelle specific and more general in the proof assistant landscape.
- What is assumed:
  - Some familiarity with functional programming
  - Basic logic, discrete maths, some semantics (for the last lecture).

A DISCLAIMER ....

This course IS...

...unashamedly a course on the practical use of proof assistants and in particular, Isabelle/HOL

Main course goals:

- Be able to use Isabelle to start your own project/keep learning yourself.
- Understand the importance of modularity in formal proof and use important tools/advanced proof techniques in Isabelle/HOL to manage such modularity
- Understand the role proof assistants can play in several areas of foundations research

This course IS NOT:

- A type theory course
- A course on the details of all proof assistants (or for that matter, even all the details of Isabelle/HOL!).
- An introduction to a particular foundational concept which only uses Isabelle for exercises

COURSE RESOURCES

- Documentation
  - See the course website for slides, notes, and exercises:
  - https://cledmonds.github.io/mgs2025/
  - Will be updated throughout this week!
- Other useful resources:
  - The official documentation (particularly prog-prove & locales tutorials): Comes with Isabelle distribution
  - Tobias Nipkow's Concrete Semantics Book: http://concrete-semantics.org/
  - Machine Logic Blog: Interesting exploration of Isabelle and history by Larry Paulson - https://lawrencecpaulson.github.io/

LECTURE 1 OVERVIEW

- Introduction to Proof Assistants
  - History, major developments, motivation
- Introduction to Isabelle/HOL
- A fast-paced “tour” through key basic concepts
  - The editors
  - Some logical proofs
  - Functions, datatypes, tactics.
  - More examples!
  - Isabelle Infrastructure: AFP, automation, search, etc
  - Summary of other advanced features

INTRODUCTION TO PROOF ASSISTANTS

PROOF ASSISTANTS

- Interactive proof assistants allow us to prove theorems in a logical formalism:
  - With precise definitions of concepts
  - A formal deductive system
  - And (hopefully) automated tools
- We can create hierarchies of definitions and proofs
  - Specifications of components and properties
  - Proofs that designs meet their requirements.
- Interactive = “guided” by a human user to produce a formalisation or mechanisation.

WHY FORMALISE?

WHY FORMALISE?

A very simple example ....

Are the proofs below correct? Are they valid theorems to begin with?

$(P \to Q), (Q \to R) \vdash R$

1.	$(P \to Q)$	hyp
2.	$(Q \to R)$	hyp
3.	P	hyp
4.	Q	$(\to E)$, 1, 3
5.	R	$(\to E)$, 2, 4
6.	$P \to R$	$(\to I)$ 3-5
7.	R	$(\to E)$ 6,3

$\forall x \exists y P(x, y) \vdash \exists x \forall y P(x, y)$

1.	$\forall x \exists y P(x, y)$	hyp
2.	$\exists y P(a, y)$	$(\forall E)$ 1
3.	$P(a, b)$	$(\exists E)$ 2
4.	$\forall x P(x, b)$	$(\forall I)$ 3
5.	$\exists y \forall x P(x, y)$	$(\exists I)$ 4

$(P \wedge Q) \to R \vdash P \to (Q \to R)$

1.	$(P \wedge Q) \to R$	hyp
2.	P	hyp
3.	Q	hyp
4.	$P \wedge Q$	$(\wedge E_I)$ 2, 3
5.	R	$(\to E)$ 1, 4
6.	$Q \to R$	$(\to I)$ 3-5
7.	$P \to Q \to R$	$(\to I)$ 2-6

WHY FORMALISE?

A very simple example ....

$(P \to Q), (Q \to R) \vdash R$

1.	$(P \to Q)$	hyp
2.	$(Q \to R)$	hyp
3.	P	hyp
4.	Q	$(\to E)$, 1, 3
5.	R	$(\to E)$, 2, 4
6.	$P \to R$	$(\to I)$ 3-5
7.	R	$(\to E)$ 6,3

NOT A THEOREM! $(\to E)$ at 7

$\forall x \exists y P(x, y) \vdash \exists x \forall y P(x, y)$

1.	$\forall x \exists y P(x, y)$	hyp
2.	$\exists y P(a, y)$	$(\forall E)$ 1
3.	$P(a, b)$	$(\exists E)$ 2
4.	$\forall x P(x, b)$	$(\forall I)$ 3
5.	$\exists y \forall x P(x, y)$	$(\exists I)$ 4

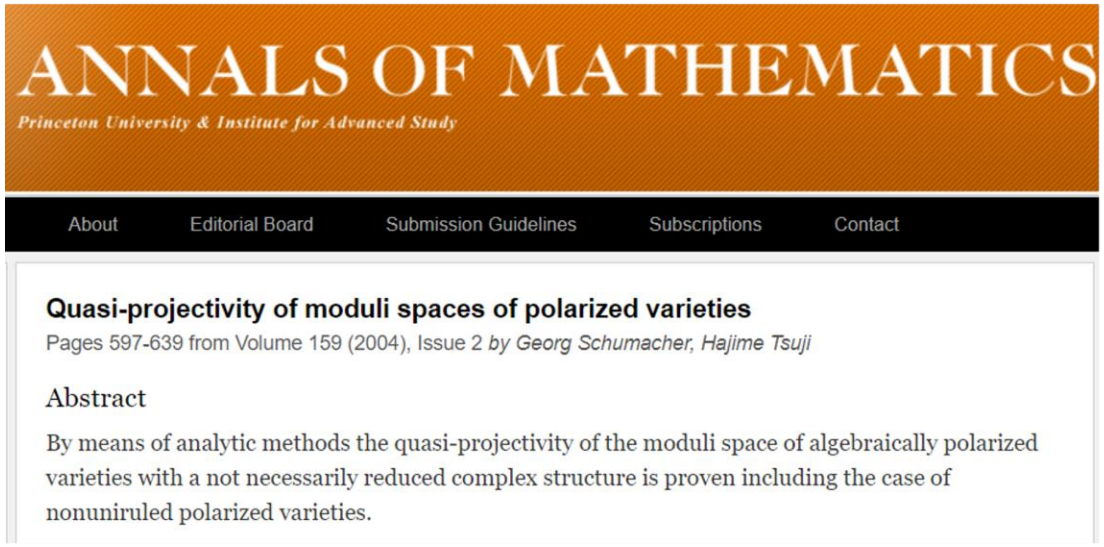
NOT A THEOREM! $(\exists E)$ at 3

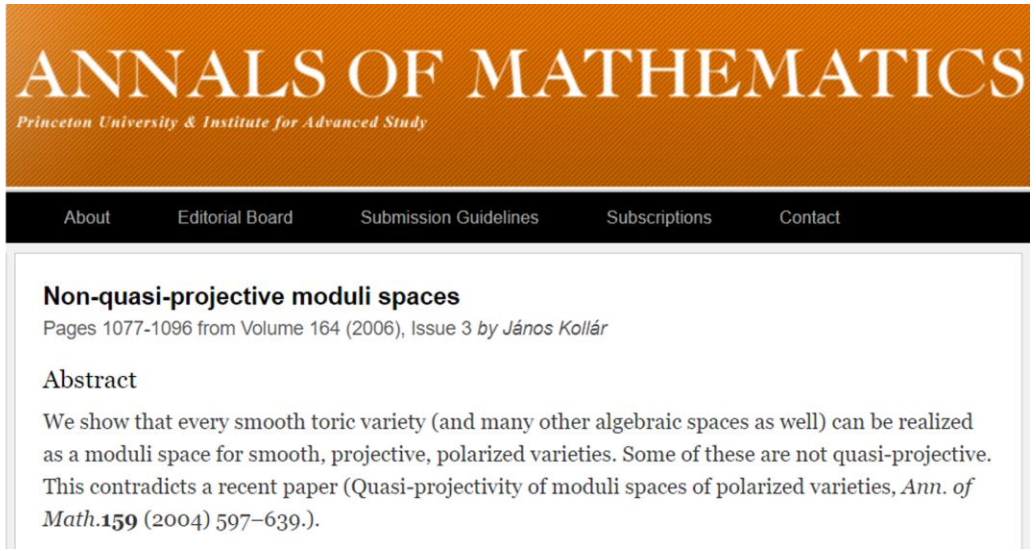
$(P \wedge Q) \to R \vdash P \to (Q \to R)$

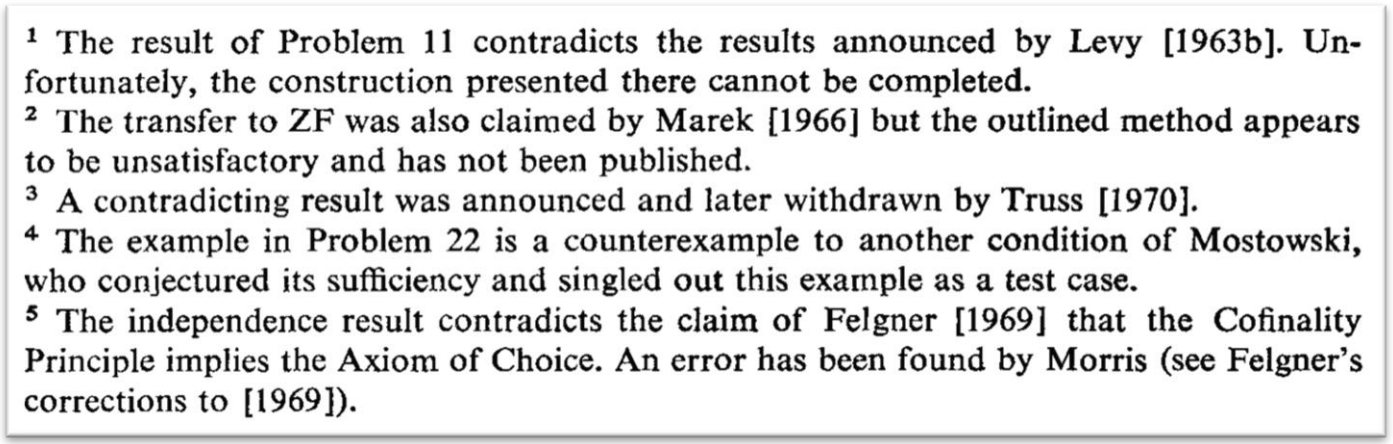
1.	$(P \wedge Q) \to R$	hyp
2.	P	hyp
3.	Q	hyp
4.	$P \wedge Q$	$(\wedge E_I)$ 2, 3
5.	R	$(\to E)$ 1, 4
6.	$Q \to R$	$(\to I)$ 3-5
7.	$P \to Q \to R$	$(\to I)$ 2-6

PROOF ERROR: $(\wedge I)$ at 4

WHY FORMALISE?







[1] The result of Problem 11 contradicts the results announced by Levy [1963b]. Unfortunately, the construction presented there cannot be completed.
[2] The transfer to ZF was also claimed by Marek [1966] but the outlined method appears to be unsatisfactory and has not been published.
[3] A contradicting result was announced and later withdrawn by Truss [1970].
[4] The example in Problem 22 is a counterexample to another condition of Mostowski, who conjectured its sufficiency and singled out this example as a test case.
[5] The independence result contradicts the claim of Felgner [1969] that the Cofinality Principle implies the Axiom of Choice. An error has been found by Morris (see Felgner's corrections to [1969]).

*Footnotes on page 118 of Jech's *The Axiom of Choice (1973)*

WHY FORMALISE?

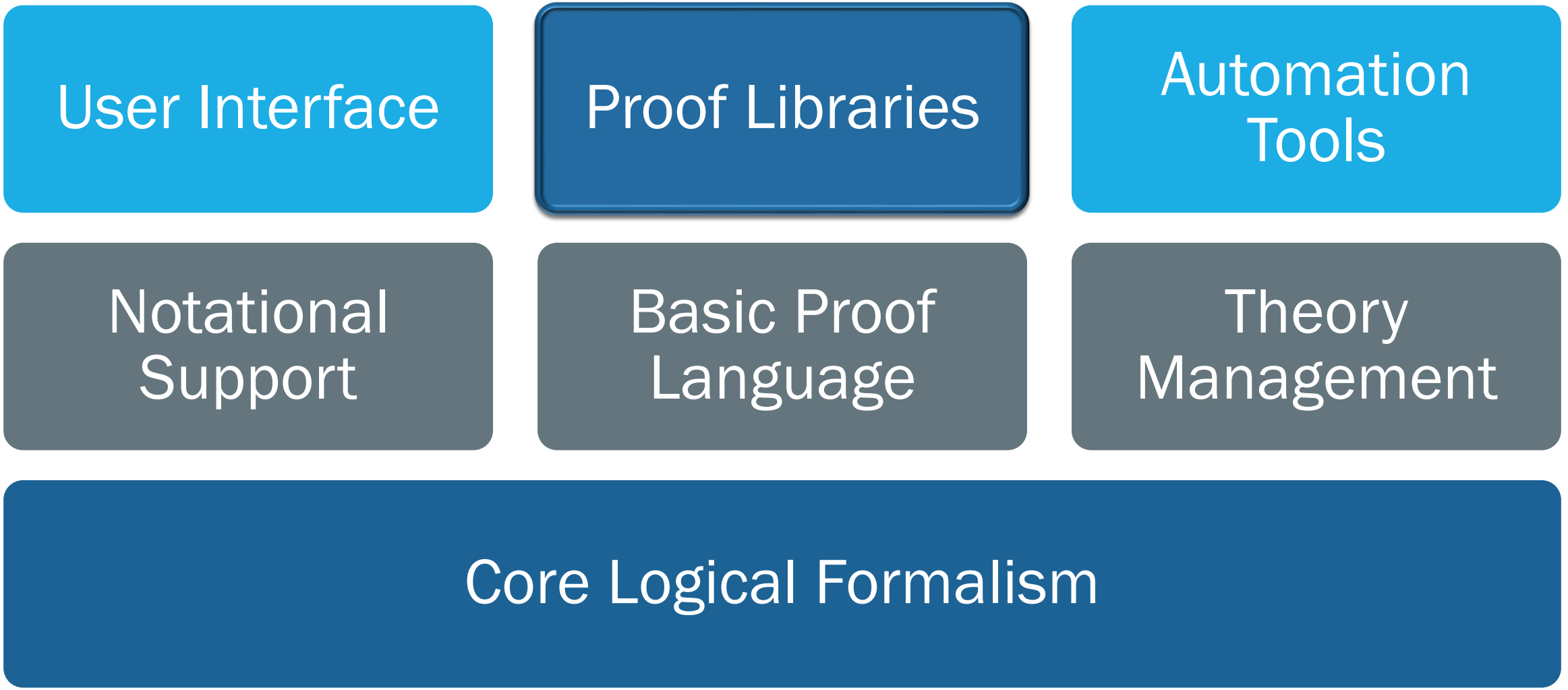
To validate complex proofs

To reveal hidden assumptions & proof steps

To create central libraries of verified mathematical/CS knowledge

To benefit from advances in automation and technology

PROOF ASSISTANT COMPONENTS



SOME HISTORY

- **Automath** (de Bruijn, 1968): The first! Novel type theory. Formalised the construction of the reals.
- **Mizar** (Trybulec, 1973): Set theory with “soft typing”. Structured formal language
- **Rocq (Coq)** (Coquand and Huet et al, 1984): Dependent type theory.
- **HOL [Light]** (Gorden, 1988, Harrison, 1992): Simple type theory/Higher-order logic. First to verify real analysis.
- **Isabelle[HOL]** (Paulson, 1986)**:** Isabelle is a generic proof assistant. Its main instance is simple type theory/higher order logic.
- **Agda** (Coquand, 1999, Ulf, 2007): A dependently typed functional programming language, that is also a proof assistant. Based on Intuitionistic type theory.
- **Lean** (de Moura et al, 2015)**:** Dependent type theory. Has a strong community for formalised maths.
- **And many more …**

THE ISABELLE PROOF ASSISTANT

THE ISABELLE PROOF ASSISTANT

Isabelle

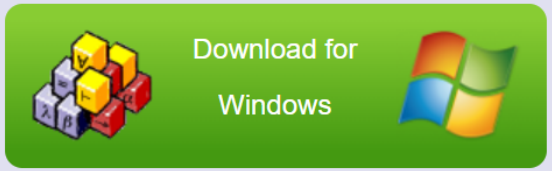
UNIVERSITY OF CAMBRIDGE
Computer Laboratory

TECHNISCHE UNIVERSITÄT MÜNCHEN

What is Isabelle?

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. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. See the Isabelle overview for a brief introduction.

Now available: Isabelle2025 (March 2025)



Download for Linux (Intel) - Download for Linux (ARM) - Download for Windows - Download for macOS

Hardware requirements:

- *Small experiments:* 4 GB memory, 2 CPU cores
- *Medium applications:* 8 GB memory, 4 CPU cores
- *Large projects:* 16 GB memory, 8 CPU cores
- *Extra-large projects:* 64 GB memory, 16 CPU cores

ISABELLE OVERVIEW

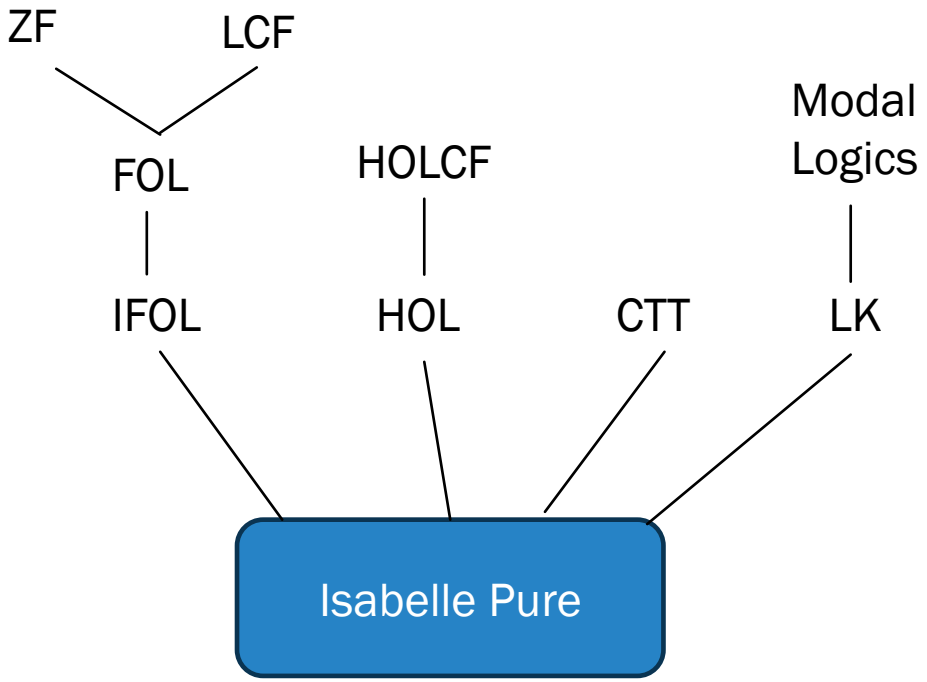
- Simple type theory/HOL
- Sledgehammer – automated proof search.
- Counter-example generators
- Search tools: Query Search, Find Facts, SErAPIS
- The Isar structured proof language
- Jedit/VS Codium IDE
- Extensive existing libraries in Maths & Computer Science (AFP)
- Additional features: Code generation, documentation generation …

```
theorem  assumes "prime p"  shows "sqrt p ∉ ℚ"
proof
  from <prime p> have p: "1 < p" by (simp add: prime_def)
  assume "sqrt p ∈ ℚ"
  then obtain m n :: nat where
      n: "n ≠ 0" and sqrt_rat: "¦sqrt p¦ = m / n"
    and "coprime m n" by (rule Rats_abs_nat_div_natE)
  have eq: "m² = p * n²"
  proof -
    from n and sqrt_rat have "m = ¦sqrt p¦ * n" by simp
    then show "m² = p * n²"
      by (metis abs_of_nat of_nat_eq_iff of_nat_mult power2_eq_square real_sqrt_abs2 rea
  qed
  have "p dvd m ∧ p dvd n"
  proof
    from eq have "p dvd m²" ..
    with <prime p> show "p dvd m" by (rule prime_dvd_power_nat)
    then obtain k where "m = p * k" ..
    with eq have "p * n² = p² * k²" by (auto simp add: power2_eq_square ac_simps)
    with <prime p> show "p dvd n"
      by (metis dvd_triv_left nat_mult_dvd_cancel1 power2_eq_square prime_dvd_power_nat
  qed
  then have "p dvd gcd m n" by simp
  with <coprime m n> have "p = 1" by simp
  with p show False by simp
qed
```

sledgehammer proofs

ISABELLES FAMILY OF LOGICS

- Isabelle is a *generic* theorem prover
- Overtime, several different logics have been developed – Isabelle/HOL is by far the most widely used.



ISABELLE/HOL FOUNDATIONS

- Isabelle/HOL is based on a Higher-Order logic (i.e. simple type theory)
  - First order logic extended with functions and sets.
  - Extended to also incorporate rank-1 polymorphism (we'll get to type classes later!).
  - ML-style functional programming.
- Often introduced as HOL
- Variation of Gordon's HOL (also led to the logic behind HOL4/HOL Light)

BASIC TYPES / TERMS / FUNCTIONS

$$
\begin{array}{rcl}
\tau & ::= & (\tau) \\
& | & bool \mid nat \mid int \mid \ldots \\
& | & 'a \mid 'b \mid \ldots \\
& | & \tau \Rightarrow \tau \\
& | & \tau \times \tau \\
& | & \tau\ list \\
& | & \tau\ set \\
& | & \ldots
\end{array}
$$

- Base types
- Type variables
- Function types
- Pairs
- Lists
- Sets
- User defined types

-Postfix types have precedence over function types (i.e. $'a \Rightarrow 'b\ list$ means $'a \Rightarrow ('b\ list)$)

TERMS

$$
\begin{array}{rcl}
t & ::= & (t) \\
 & | & a \\
 & | & t\ t \\
 & | & \lambda x.\ t \\
 & | & \ldots
\end{array}
$$

Terms (follow the typed λ calculus)

- Constants, *c* and Variables, *x*
- Function applications $t\ u$
- Abstractions $\lambda x\ .\ t$
- Lots of syntactic sugar

- i.e. The language of terms is a simply type $\lambda - \text{calculus}$, noting Isabelle performs β-reduction ($(\lambda x.\ t)\ u$ to $t[u/x]$) automatically.
- Terms must be **well-typed** ($t\ ::\ \tau$)
- Isabelle automatically computers the type of each variable in a term (type inference), except for overloaded functions where type annotations can be useful.

ISABELLE'S META LOGIC

- **Implication: $\Longrightarrow$**
  - For separating premises and conclusions of theorems
- **Equality $\equiv$**
  - For definitions
- **Universal Quantifier $\bigwedge$**
  - For binding local variables

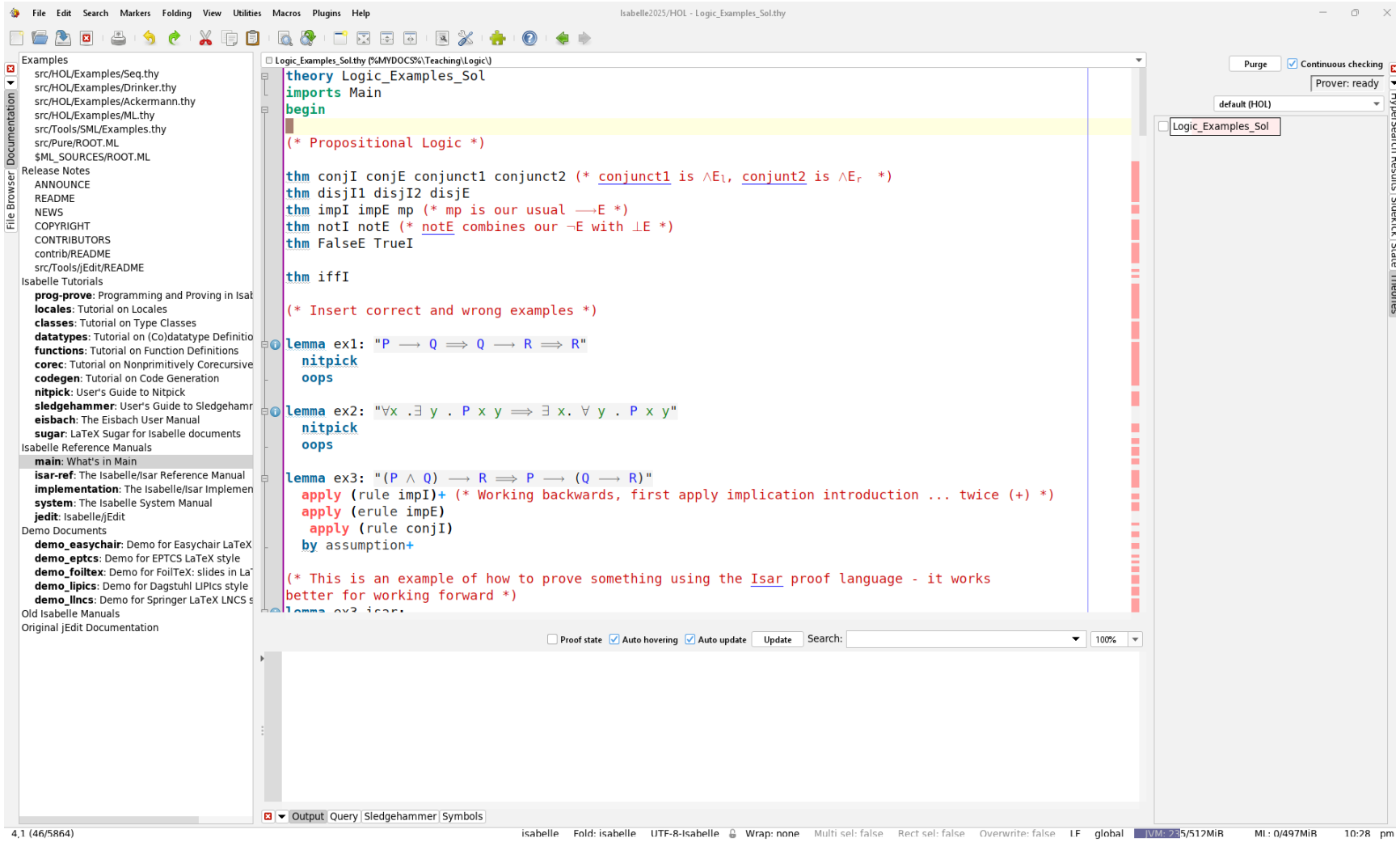
Do not use inside HOL formula!

Logically the same meaning, but differences is usability/automation

NB: The Metalogic, has itself been formalised! https://www.isa-afp.org/entries/Metalogic_ProofChecker.html

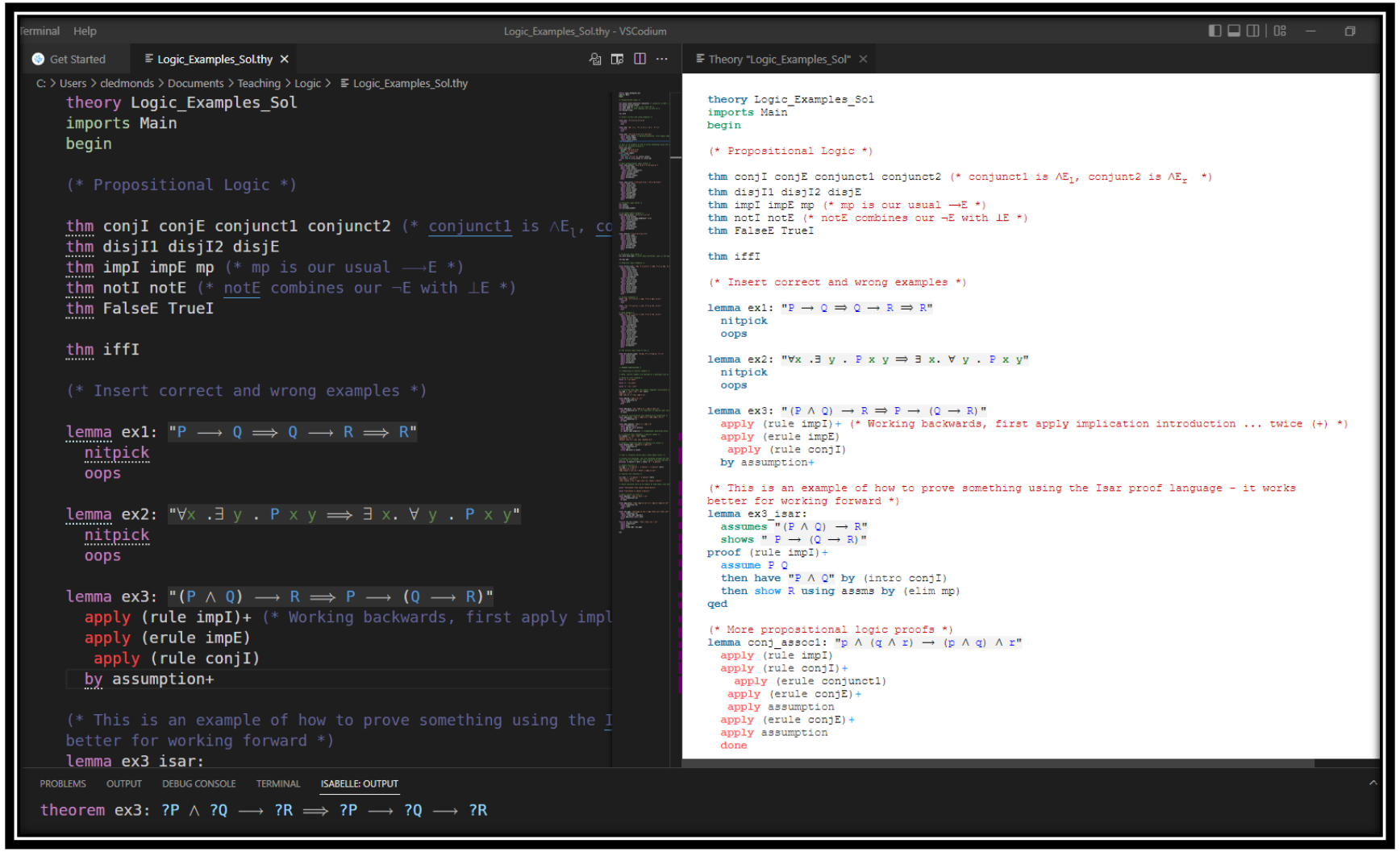
EDITORS

ISABELLE JEDIT



Includes the most customised support for Isabelle developments

ISABELLE VSCODE



New VSCode based editor

- Must use instance in the Isabelle download
- Start via:

  “isabelle vscode”
- Nice html preview
- Many less Isabelle features than jedit
- Don’t use the old VSCode extension

INTRODUCTION BY EXAMPLE

1. BOOLEAN LOGIC AND FUNCTIONS

FUNCTIONS/DATATYPES

DATATYPES

- Functional style datatypes
- Generates lots of useful facts/properties:
  - distinctness and injectivity (applied automatically).
  - Induction (needs to be applied)

```
datatype 'a mylist = Nill | Consl 'a " 'a mylist"

thm mylist.induct
thm mylist.case
```

FUNCTIONS & DEFINITIONS

- All Functions must be total!
- Fun – termination proved automatically (most things we'll deal with),

```
fun app :: "'a mylist ⇒ 'a mylist ⇒ 'a mylist" where
"app Nil ys = ys" |
"app (Consl x xs) ys = Consl x (app xs ys)"
```

- Function – user supplied termination proof.
- Definition: non-recursive definitions

```
definition prime :: "nat ⇒ bool" where
"prime p = (1 < p ∧ (∀ m. m dvd p ⟶ m = 1 ∨ m = p))"
```

- Recursive functions have more built in facts that are useful in proofs than a definition.

TACTICS

AUTO VS SIMP

Auto

- `auto` applies `simp` rules + all obvious logical steps, e.g.:
  - Splitting conjunctive goals and disjunctive assumptions
  - Performing obvious quantifier removal
- It operates on *all* subgoals
- Designated intro and elimination rules included in this

Simp

- `Simp` performs *rewriting* (along with simple arithmetic simplification)
- It only operates on the *first* subgoal
- Some facts are included in the simplifier
- Other facts are often useful, e.g. for arithmetic, consider trying the following:
  - `algebra_simps`
  - `field_simps`
  - `divide_simps`

MORE REWRITING

- Simp rules work left to right, i.e. at each step transform the LHS into the RHS
- Isabelle enables you to add rules to the simplifier by declaring them as such
- Rewrite rules can be conditional (and are applied if the conditions can themselves be recursively proved via simplification)
- But! We need to be careful to avoid loops.
  - The following pair of “simp” rules would cause issues:

$$f(x) = h\big(g(x)\big), g(x) = f(x + 2)$$

  - Permutative rewrite rules (e.g. $x + y = y + x$) are applied but only if they make the term “lexicographically smaller”

VARIATIONS ON SIMP/AUTO

- Add a fact (once-off) to be used for simplification: `simp add: app_assoc`
- Omit a fact (once-off) from simplification: `simp del: rev_rev`
- Don’t simplify the assumptions: `simp (no_asm_simp)`
- Ignore the assumptions: `simp (no_asm)`
- Simplify all the subgoals: `simp_all`
- Add rewriting rules/introduction rules etc to auto: `auto simp add: … intro: …`
- You can combine many of these!

SIMP TRACE

- Insert: `using [[simp_trace]]` (inline proof) or `declare [[simp_trace]]` (theory wide)

```
lemma ordered_merge[simp]: "ordered (merge xs ys) = (ordered xs ∧ ordered ys)"
  apply (induct xs ys rule: merge.induct)
   apply simp_all
 using  [[simp_trace]]
  apply (auto split: list.split simp del: ordered.simps(2))
  done
```

```
Proof state  Auto hovering  Auto update  Updat
[0]Adding rewrite rule "triv_forall_equality":
(⋀x. PROP ?V) ≡ PROP ?V
[1]SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
⋀x xs y ys.
   (x ≤ y ⟹
    ordered (merge xs (y # ys)) = (ordered xs ∧ (case ys of [] ⇒ True | ya # xs ⇒ y
   (¬ x ≤ y ⟹
    ordered (merge (x # xs) ys) = ((case xs of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered
   (x ≤ y ⟶
    (case merge xs (y # ys) of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) =
    ((case xs of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) ∧
     (case ys of [] ⇒ True | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs)))) ∧
   (¬ x ≤ y ⟶
    (case merge (x # xs) ys of [] ⇒ True | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs)) =
    ((case xs of [] ⇒ True | y # xs ⇒ x ≤ y ∧ ordered (y # xs)) ∧
     (case ys of [] ⇒ True | ya # xs ⇒ y ≤ ya ∧ ordered (ya # xs))))
[1]Adding rewrite rule "??.??.unknown":
¬ xa ≤ ya ⟹
ordered (merge (xa # xsb) ysb) ≡ (case xsb of [] ⇒ True | y # xs ⇒ xa ≤ y ∧ ordered
[1]Adding rewrite rule "??.??.unknown":
xa ≤ ya ≡ True
[1]Applying congruence rule:
ysb ≡ ?list' ⟹
Output  Query  Sledgehammer  Symbols
```

MORE TACTICS

- Basic tactics such as `rule, erule, assumption, intro, elim`, used in conjunction with a *known fact*
- These can often be *combined* with auto/simp (like other variations of simp)
- We also have other automated tactics:
  - `force, fastforce`
  - `blast`: uses intro + elimination rules with powerful search heuristics (not simplification/arithmetic reasoning) and won't terminate if it doesn't work
  - Arithmetic tactics: `arith, linarith`
  - Use of tactics like "`metis`" and "`smt`" often indicate use of sledgehammer
- Other good tactics for starting a proof (less powerful, but safer): `safe, clarify, standard`
- And many more tactics: `cases, split` …
- Tactics can be combined e.g. `by (induction) (blast | fastforce)+` applies induction then repeatedly shows the subgoals using either blast or fastforce

INDUCTION

- Inductive tactics are well-developed with many options for application.
- The `induction` tactic tries to figure out what to do automatically:

```
lemma app_assoc: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induction xs)
   apply auto
  done
```

- Sometimes it can't, and we need to be more specific

```
lemma "itlen xs n = size xs + n"
  apply (induct xs arbitrary: n rule: list.induct)
   apply auto
  done
```

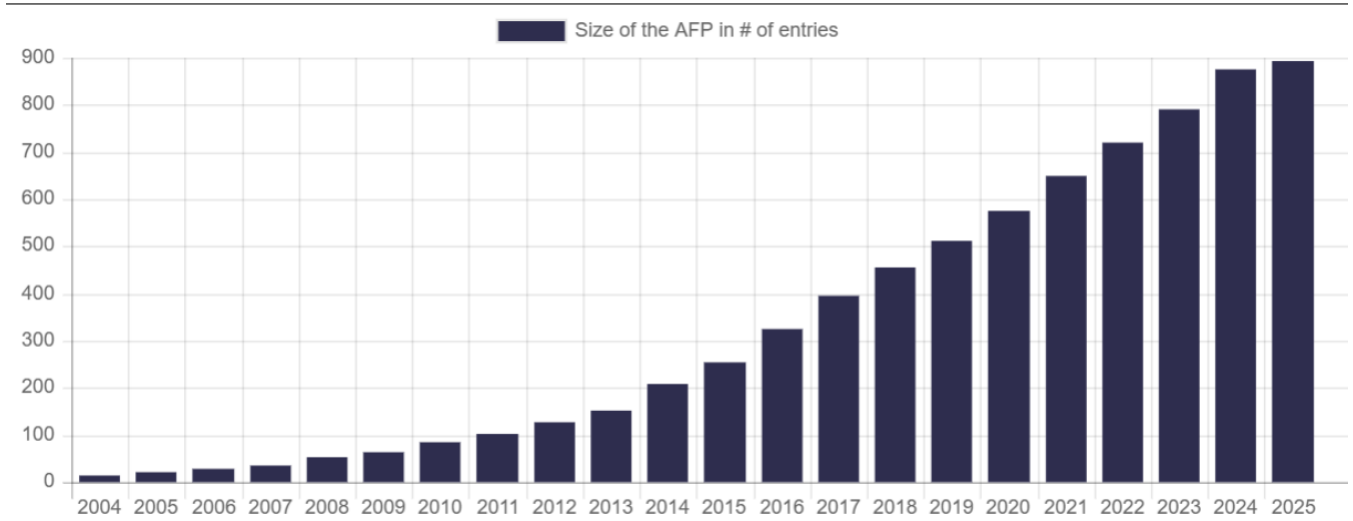
Specify n should be universally quantified in induction

Specify induction rule to use (unnecessary in this case)

USEFUL FEATURES

THE ISABELLE AFP

- A significant archive of (refereed) formalised mathematics and computer science concepts.
  - More of an “archive” than a constantly modified “library”
- https://www.isa-afp.org/
- It can be easily imported into a local instance of Isabelle by adding it as a component, see here: https://www.isa-afp.org/help/
- Over 4.5 million lines of code across 894 entries – and still growing!



SLEDGEHAMMER

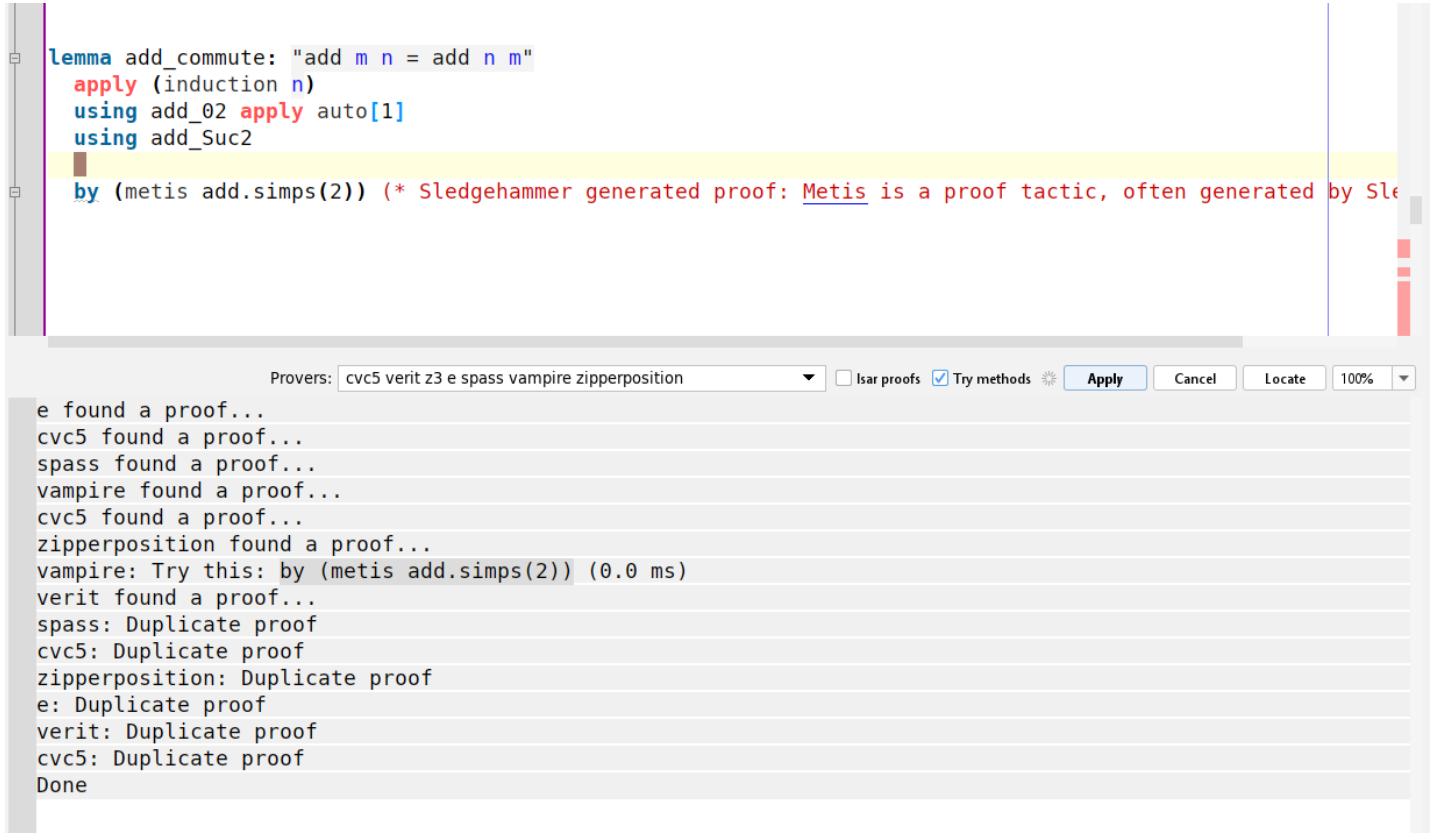
Problem + 1000s of facts/thms

AUTOMATED THEOREM PROVERS

E
SPASS
Vampire
Z3
Cvc
...

Generated Proof(s)

SLEDGEHAMMER



- Simplify the goal and break down into pieces
- Sledgehammer doesn't prove the goal, but returns a "proof" which is a call to metis, smt, blast, auto etc…
- Translations are not sound, hence sledgehammer provided proof *may not work* when inserted.
- Generated proofs can be ugly/messy – there are usually cleaner ways!

- For more history: https://lawrencecpaulson.github.io/2022/04/13/Sledgehammer.html
- For a more technical overview: https://www.cl.cam.ac.uk/~lp15/papers/Automation/paar.pdf (or many of Jasmin Blanchette's papers for more recent work).

COUNTER EXAMPLE

Nitpick

```
lemma ex2: "∀x .∃ y . P x y ⟹ ∃ x. ∀ y . P x y"
  nitpick
  oops
```

Proof state | Auto hovering

```
Nitpicking formula...
Nitpick found a counterexample for card 'b = 3 and card 'a = 2:
  Free variable:
    P = (λx. _)
        (a₁ := (λx. _)(b₁ := False, b₂ := False, b₃ := True),
           a₂ := (λx. _)(b₁ := True, b₂ := True, b₃ := False))
  Skolem constants:
    λx. y = (λx. _)(a₁ := b₃, a₂ := b₂)
    λx. y = (λx. _)(a₁ := b₁, a₂ := b₃)
```

Output | Query | Sledgehammer | Symbols

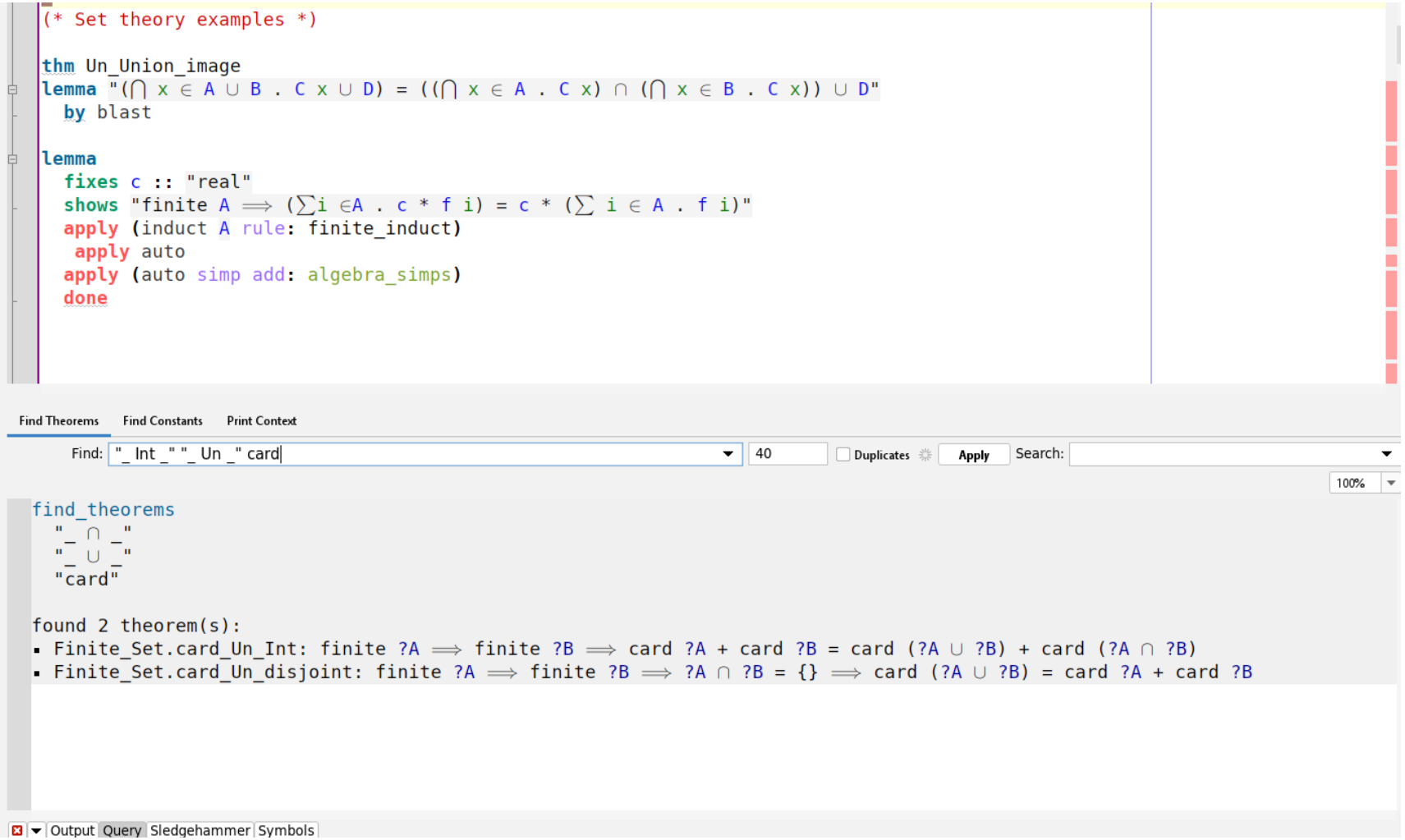
Quickcheck

```
lemma ex2: "∀x .∃ y . P x y ⟹ ∃ x. ∀ y . P x y"
  quickcheck
  oops
```

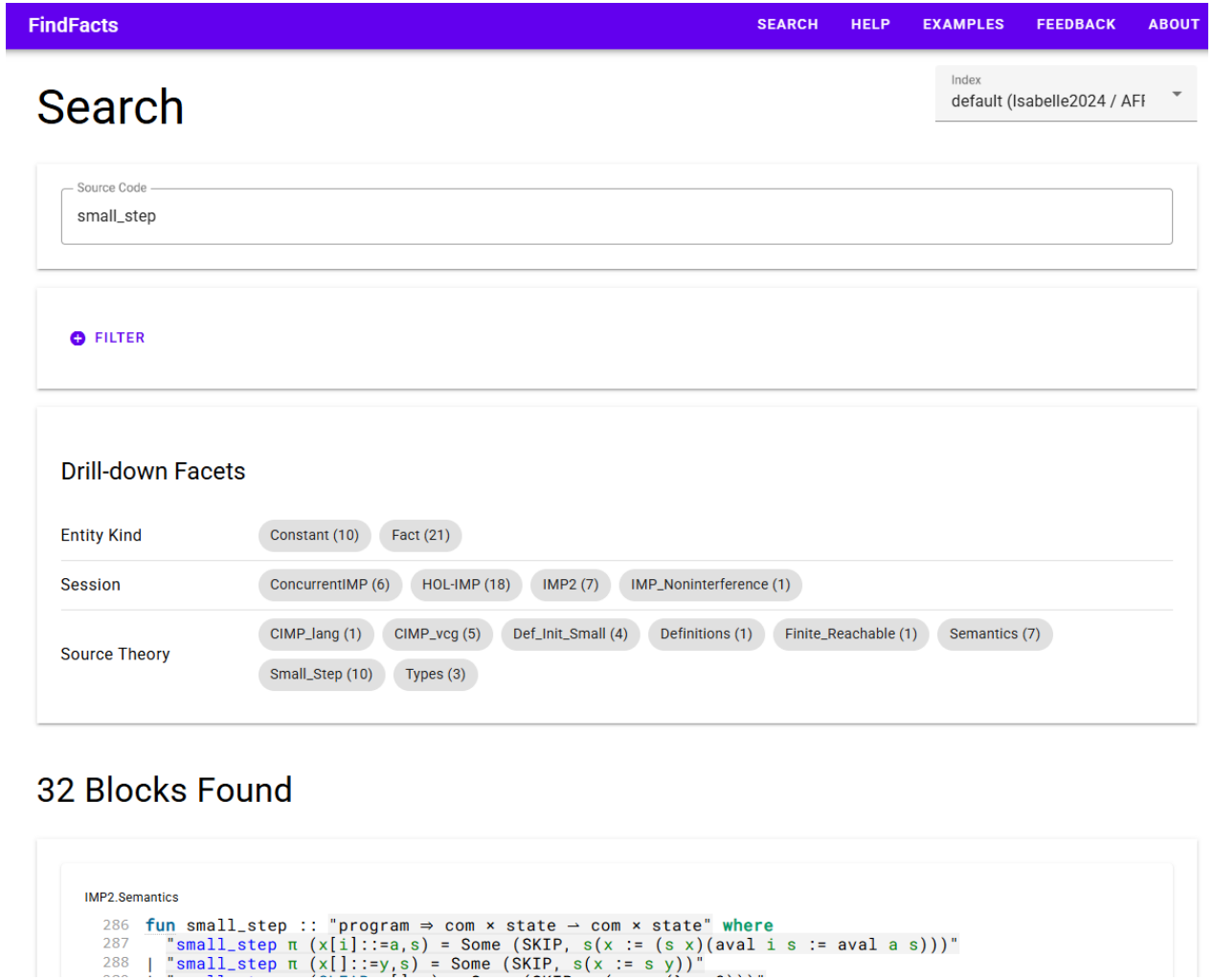
```
Testing conjecture with Quickcheck-exhaustive...
Quickcheck found a counterexample:
  P = (λx. undefined)(a₁ := {a₂}, a₂ := {a₂})
```

Output | Query | Sledgehammer | Symbols

SEARCH: QUERY



SEARCH: FINDFACTS



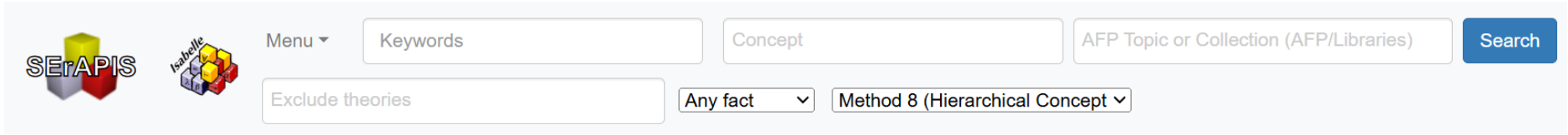
https://search.isabelle.in.tum.de/

OR

Local Database with Isabelle2025

isabelle find_facts_server -p 8080 -o find_facts_database_name=isabelle

SEARCH: SERAPIS



Welcome to SErAPIS

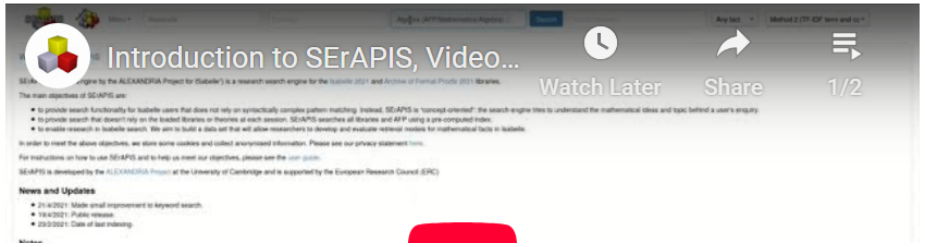
SErAPIS (“Search Engine by the ALEXANDRIA Project for ISabelle”) is a research search engine for the Isabelle 2021 and Archive of Formal Proofs 2021 libraries.

The main objectives of SErAPIS are:

- to provide search functionality for Isabelle users that does not rely on syntactically complex pattern matching. Instead, SErAPIS is “concept-oriented”: the search engine tries to un the mathematical ideas and topic behind a user's enquiry.
- to provide search that doesn't rely on the loaded libraries or theories at each session. SErAPIS searches all libraries and AFP using a pre-computed index.
- to enable research in Isabelle search. We aim to build a data set that will allow researchers to develop and evaluate retrieval models for mathematical facts in Isabelle.

In order to meet the above objectives, we store some cookies and collect anonymised information. Please see our privacy statement here.

We have prepared two short videos to get you started with using SErAPIS:

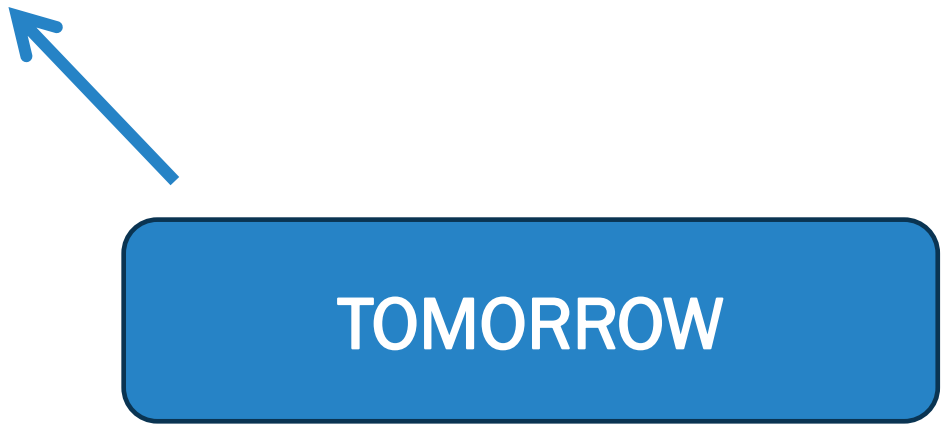


https://behemoth.cl.cam.ac.uk/search/

Note: Last AFP Index was in 2021

OTHER COOL FEATURES

- Code Generation
- Document Preparation
- Lifting and Transfer
- Eisbach => Proof Method language
- **Polymorphism (Type classes) and a powerful module system (Locales)**



NEXT TIME…

- Example Class:
  - Get started with Isabelle: Logic and function proofs
  - Test out sledgehammer for yourself
  - Try out different tactics
  - Gain familiarity with Isabelle tools
- Next Lecture
  - Starting on modularity!
  - Finish off your “tour” overview of Isabelle with the Isar proof language and more advanced types
  - Introducing type classes and locales
- To come… more advanced case studies in mathematics and program verification!