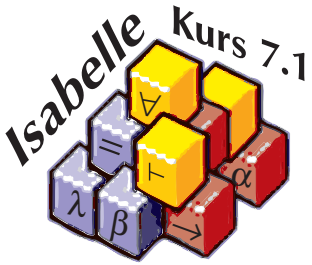
Teaching Isabelle to Secondary-School Students
A Case Study

Paul Baumgartner Daniel Luckhardt

ThEdu'26



An intensive 53-hour course culminating in a formal proof of RSA correctness

Motivation and setting

Why teach proof assistants at school?

- At German secondary schools, proof-writing is usually limited to
  - algebraic transformations and elementary geometry,
  - induction-like or informal arguments, and
  - results obtained with computer algebra systems.
- Isabelle/HOL occupies an interesting middle ground:
  - the computer provides feedback and automation,
  - but students must expose hypotheses, goals, definitions, and proof steps.

Can secondary-school students learn genuine formalization—not just toy proofs?

A case study with two viewpoints

- Course at the Deutsche SchülerAkademie (DSA)
- intensive summer program for highly talented and motivated students
- authors' perspectives:
  - one co-designed and taught the course,
  - one participated as a student.

16 students • ages 16–18 • 2 instructors
53 course hours • 16-day block course

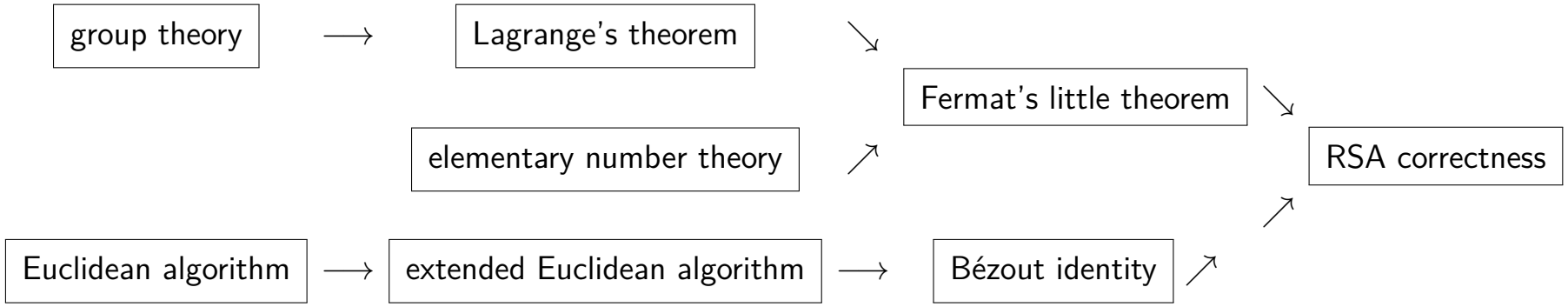
The format shaped the course

DSA constraint	Design consequence
Preparatory work at home	Students first encountered Isabelle alone/in pairs
Different prior knowledge	Pairing used a poll/quiz (besides geographic proximity)
Mid-course presentation	Groups had to explain their project to peers
Student-written documentation	required time/intro to Isabelle document preparation
Bring-your-own hardware	The project had to be modest in resource requirements

Central tension: early independent work was required, although beginners benefit most from immediate in-person support.

Course design

Leitmotif: verify RSA cryptosystem



- unifying goal
- practical motivation
- Two products:
  1. a machine-checked correctness proof of RSA,
  2. bonus: executable encryption and decryption functions.

Resulting tasks for groups

Groups	Project	Main activity
1–3	Selected chapters from *Programming and Proving in Isabelle/HOL*	remember, understand
4	Elementary proof of Fermat's little theorem	apply, analyze
5	Extended Euclidean algorithm: define, prove, export	understand, apply, create
6	Elementary group theory	analyze, create, evaluate
7	Lagrange's theorem and Fermat as a corollary	apply, analyze
8	RSA definition and correctness, using Groups 4 and 5	analyze, create
4, 5, 8	Bonus: combine and export RSA	understand, create

Methodical outline

- Beginning: short introduction to mathematical logic, etc.
- Afterwards: mostly independent and group work
- Each group received a theory file with
  - instructions,
  - definitions and theorem statements with `sorry`s,
  - selected hints.
- Instructors remained available while groups worked and crossed group boundaries.

Sample project: Group 5

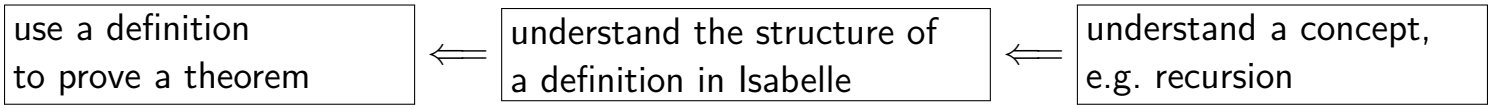
- build on Group 4's congruence results,
- define the Euclidean and extended Euclidean algorithms,
- prove divisibility, greatest-common-divisor, symmetry, and Bézout properties,
- export and run executable code,
- prepare a short presentation and exercises for peers.

The assignment mixed completion, investigation, proof, creation, and communication.

```
imports Main "4_littleFermat_binomialCoeff"

begin

section ‹Informal introduction and instructions›

text ‹This project aims at
  ▪ defining the Euclidean algorithm and the extended Euclidean algorithm,
  ▪ proving their properties,
  ▪ exporting both  to an executable programme›

subsection ‹Connection to Project 4, little Fermat theorem›

text ‹In this project you may use results from Project 4.
    This is to say that there is no need to bother about the sorries therein and you only must work
    in this file. ›

subsection ‹Tasks›

text ‹
  ▸ Replace all sorries in this theory by a proof. This may include adding new lemmas.
  ▸ Do the other tasks indicated throughout this document.
  presentation:
  ▸ Prepare a short presentation regarding the content of your project and your progress.
  ▸ Present around 3 exercise regarding the content of your project.
    After your presentation the participants are invited to work on these exercises under your
    guidance.›

section ‹Connection of Project 4 with the mod function›

text ‹The Euclidean algorithm is built base on the mod function.
  To see its definition press Ctrl and klick on any "mod".
  The crucial connection between the mod function and the congruence relation examined in the
  Project 4 is as expressed by the following lemma.›

lemma mod_cogruent: "(a ≡ b ⟨n⟩)  ⟷ (a mod n) = (b mod n)"
text ‹(*delete ! ! ! ›
  by (simp add: "4_littleFermat_binomialCoeff.congruent_def" mult.commute nat_mod_eq_iff)


section ‹The simple Euclidean algorithm›

text ‹The Euclidean algorithm calculates the greatest common divisor (gcd, Germ. größter gemeinsamer
  Teiler, ggT) of two given natural numbers.
  It may be defined as:›

fun gcd :: "nat ⇒ nat ⇒ nat" where
"gcd m 0 = m" |
"gcd m n = gcd n (m mod n)"

subsection ‹Properties›

lemma divides_0[simp]:"n divides 0"
  sorry

theorem gcd_divides_both: "(gcd m n divides m) ∧ (gcd m n divides n)"
  text ‹Hint: Consult 🗎‹prog-prove›, 4.4.3, and also 2.3.4›
  sorry

theorem gcd_greatest: "k divides m ⟶ k divides n ⟶ k divides gcd m n"
  text ‹Hint: Consult 🗎‹prog-prove›, 4.4.3, and also 2.3.4›
  sorry

proposition gcd_sym:"gcd a b = gcd b a"
  sorry

subsection ‹Task: Now export!›

text ‹
  ▸ export the function to a programming language of your choosing.
    To this end consult the 🗎‹codegen› tutorial.
  ▸ Run the code on your machine.
›

section ‹The extended algorithm›

text ‹For applications one often does not only require only the gcd, but also a concrete way
  to represent it as a linear combination of its arguments.
  This linear combination is called Bézout identity.
  There are many resources online available, let's say at proof wiki
  🌐‹https://proofwiki.org/wiki/Extended_Euclidean_Algorithm#google_vignette›.›

subsection ‹Tasks›

text ‹
  ▸ Replace the definition of gcdExt below by one resembling the extended Euclidean algorithm
    such that for gcdExt(a,b) = (l,m,n) the Bézout identity l = ma + nb holds.
  ▸ Poof the lemmas stated below.
  ▸ Export the algorithm like in the case of the simple one
  ›

subsection ‹Templates›

fun gcdExt :: "nat ⇒ nat ⇒ nat × int × int" where
  "gcdExt a b = (gcd a b, 0, 0)"

lemma gcd_extension: "gcd a b = fst(gcdExt a b)"
  sorry

lemma gcd_Bezout: "fst(gcdExt a b) = (fst (snd t(gcdExt a b)))*a + (snd (snd t(gcdExt a b)))*b  "
  sorry

section ‹Lowest common multiple›

fun lcm :: "nat ⇒ nat ⇒ nat" where
  "lcm a b = a*b div (gcd a b) "

lemma lcm_divided_by_first: "a divides lcm a b"
  sorry

proposition lcm_sym:"lcm a b = lcm b a"
  sorry

theorem lcm_divided_by_both: "(a divides lcm a b) ∧ (b divides lcm a b)"
  sorry
end
```

What happened?

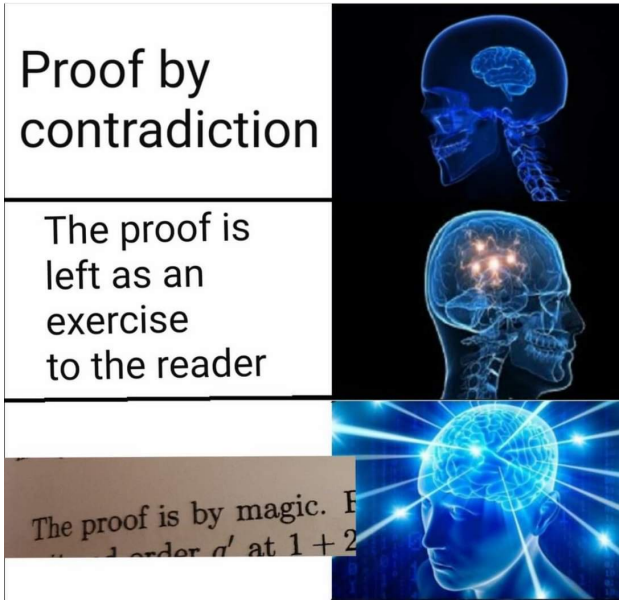
Challenge 1: the prerequisite stack



- Formal definitions expose details that school mathematics often suppresses.
- The official *prog-prove* manual named what can be used, but not always explained what it is or why it is designed that way.
- While the actual math including proofs was accessible to the participants, the surrounding conceptual stack that popped up was not.
- Result: more time needed, and substantial frustration for participants.

Challenge 2: "proof by magic"

- `auto` and Sledgehammer could solve large parts of introductory exercises.
- Students saw that a goal vanished—but not necessarily which mathematical ideas did the work.
- Their phrase "proof by magic" captured the didactic cost of opaque automation.
- Yet automation was useful when students were stuck and when checking a transition they understood.



What helped students move forward

1. Proving in groups
   Pool practical knowledge, explain ideas, ask better questions—and share frustration.
2. Extending the curriculum when needed
   Self-study missing mathematics and computer science at the point of use.
3. Writing proof sketches by hand first
   Decide on intermediate mathematical steps before asking automation to check them.
4. Creating a shared cheat sheet
   Keep the most useful propositional rules and lemmas readily available.

Isabelle's explicit subgoal display and lemma search were frequently experienced as helpful.

Outcome and evaluation

The students completed the Isabelle formalization of RSA correctness.

The planned export of the combined cryptosystem remained unfinished.

- 14 of 16 students acquired foundational problem-solving competence in Isabelle; some progressed substantially further.
- Two students experienced significant difficulties and disengaged.
- The long block format enabled a coherent project instead of isolated demonstrations.

Important qualification:

Some qualifications

- qualitative observations, not measured learning gains.
- No grades, pre/post test, control group, or systematic quantitative assessment
- highly selected, motivated participaunts limit generalization

Conclusion

Learning to prove and learning the prover simultaneously

- added complexity
- gave students clear subgoals to prove
- provided them with immediate feedback
- enabled them to search for simplification rules and lemmas
- obscured some proofs when using sledgehammer
- was overshadowed by some (unnecessary) prerequisites

Though basically all students at some points experienced some frustration, the great majority achieved some proof competency in Isabelle. The main goal of verifying the RSA cryptosystem was achieved.

Lessons for the next course

- More time allocated for dynamically accommodating differing student prerequisites through targeted supplementary instruction and self-study.
- Focus on group work.
- A didactical version of Isabelle.
- Exclusion of any rigorous treatment of mathematical logic.
- A script tailored to the level of students nearing the end of high school. Outline:

  1. instructions for installing the software
  2. Isar proofs
  3. basic arguments that students are likely to know from school mathematics
  4. ...

Further scope: Tools like ProofBuddy, AI, etc.

Thank you for your attention

Questions?