On Exams with the Isabelle Proof Assistant

Frederik Krogsdal Jacobsen Jørgen Villadsen

Technical University of Denmark

Introduction

- Our graduate course in automated reasoning needs an evaluation of student outcomes
- We use Isabelle throughout the course
- The exam is also in Isabelle
- How can we test that students know what is going on when Isabelle has so much automation?
- How can we make the grading as easy as possible?

Overview of our exam questions

1. Isabelle proofs without automation
2. Verification of functional programs in Isabelle/HOL
3. Natural deduction proofs
4. Sequent calculus proofs
5. General proofs in Isabelle/HOL with Isar

Isabelle/Pure

- Students are asked to prove simple logical formulas
- Higher-order logic
- No automation
- Proof rules must be applied manually
- Isabelle can infer which rule to use in many, but not all, cases

Example

```
subsection ‹Question 1›

text ‹ Replace "by blast" with a proof in Pure_True (if possible omit names of Pure_True rules). ›

proposition ‹p ⟷ ¬ ¬ p›
  by blast
```

Example

```
subsection ‹Question 1›

text ‹ Replace "by blast" with a proof in Pure_True (if possible omit names of Pure_True rules). ›

proposition ‹p ⟷ ¬ ¬ p›
proof
  assume p
  show ‹¬ ¬ p›
  proof
    assume ‹¬ p›
    from this and ‹p› show ⊥ ..
  qed
next
  assume ‹¬ ¬ p›
  show p
  proof (rule ccontr)
    assume ‹¬ p›
    with ‹¬ ¬ p› show ⊥ ..
  qed
qed
```

Programming and proving

- Simple functional programs
- Simple proofs by induction
- Automation is allowed
- Finding the right level of complexity is hard

Example

```
subsection ‹Question 1›

text ‹ Using only the constructors 0 and Suc and no arithmetical operators, define a recursive function
triple :: ‹nat ⇒ nat› and prove ‹triple n = 3 * n›. ›
```

Example

```
subsection ‹Question 1›

text ‹ Using only the constructors 0 and Suc and no arithmetical operators, define a recursive function
triple :: ‹nat ⇒ nat› and prove ‹triple n = 3 * n›. ›

fun triple :: ‹nat ⇒ nat› where
  ‹triple 0 = 0› |
  ‹triple (Suc n) = Suc (Suc (Suc (triple n)))›

lemma ‹triple n = 3 * n›
  by (induct n) simp_all
```

Natural Deduction Assistant

- Graphical web application for natural deduction proofs
- Classical first-order logic
- Impossible to apply proof rules wrong, but students still have to choose which ones to use
- Students export proofs to Isabelle

Example

```
subsection ‹Question 1›

text ‹ Use NaDeA to prove the following formula and include the load lines as a (*...*) comment. ›

proposition ‹((A ⟶ B) ⟶ A) ⟶ A› by metis
```

Example

Natural Deduction Assistant Load Code Help ProofJudge 36/36 Stop Undo

1	Imp_I	$[]\ ((A \to B) \to A) \to A$
2	Boole	$[(A \to B) \to A]\ A$
3	Imp_E	$[A \to \bot, (A \to B) \to A]\ \bot$
4	Assume	$[A \to \bot, (A \to B) \to A]\ A \to \bot$
5	Imp_E	$[A \to \bot, (A \to B) \to A]\ A$
6	Assume	$[A \to \bot, (A \to B) \to A]\ (A \to B) \to A$
7	Imp_I	$[A \to \bot, (A \to B) \to A]\ A \to B$
8	Boole	$[A, A \to \bot, (A \to B) \to A]\ B$
9	Imp_E	$[B \to \bot, A, A \to \bot, (A \to B) \to A]\ \bot$
10	Assume	$[B \to \bot, A, A \to \bot, (A \to B) \to A]\ A \to \bot$
11	Assume	$[B \to \bot, A, A \to \bot, (A \to B) \to A]\ A$

Example

```
proposition "((A --> B) --> A) --> A"
  by blast

theorem "semantics e f g (Imp (Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])) (Pre 0 []))"
proof (rule soundness)
  show "OK (Imp (Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])) (Pre 0 [])) []"
  proof (rule Imp_I)
    show "OK (Pre 0 []) [Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
    proof (rule Boole)
      show "OK Falsity [Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
      proof (rule Imp_E)
        show "OK (Imp (Pre 0 []) Falsity) [Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
          by (rule Assume) simp
      next
        show "OK (Pre 0 []) [Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
        proof (rule Imp_E)
          show "OK (Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])) [Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
            by (rule Assume) simp
        next
          show "OK (Imp (Pre 0 []) (Pre 1 [])) [Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
          proof (rule Imp_I)
            show "OK (Pre 1 []) [Pre 0 [], Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
            proof (rule Boole)
              show "OK Falsity [Imp (Pre 1 []) Falsity, Pre 0 [], Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
              proof (rule Imp_E)
                show "OK (Imp (Pre 0 []) Falsity) [Imp (Pre 1 []) Falsity, Pre 0 [], Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
                  by (rule Assume) simp
              next
                show "OK (Pre 0 []) [Imp (Pre 1 []) Falsity, Pre 0 [], Imp (Pre 0 []) Falsity, Imp (Imp (Pre 0 []) (Pre 1 [])) (Pre 0 [])]"
                  by (rule Assume) simp
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

Sequent Calculus Verifier

- Text-based web application for sequent calculus proofs
- Classical first-order logic
- Users must manually specify the result of applying each proof rule
- Students export proofs to Isabelle

Example

```
subsection ‹Question 2›

text ‹Construct a SeCaV proof for the following formula and add (* … *) with the shortened proof›

proposition ‹(∀x. p x) ∧ q ⟶ r ∨ (∃x. p x)› by metis
```

Example

Sequent Calculus Verifier | Help and Input Examples | 27:6 | Copy Output to Clipboard | SeCaV Unshortener 1.A

```
Imp (Con (Uni p[0]) q) (Dis r (Exi p[0]))

AlphaImp
  Neg (Con (Uni p[0]) q)
  Dis r (Exi p[0])
AlphaCon
  Neg (Uni p[0])
  Neg q
  Dis r (Exi p[0])
GammaUni
  Neg p[a]
  Neg q
  Dis r (Exi p[0])
Ext
  Dis r (Exi p[0])
  Neg p[a]
AlphaDis
  r
  Exi p[0]
  Neg p[a]
Ext
  Exi p[0]
  Neg p[a]
GammaExi
  p[a]
  Neg p[a]
Basic
```

```
proposition ‹((∀x. (p x)) ∧ q) ⟶ (r ∨ (∃x. (p x)))› by metis

text ‹
  Predicate numbers
    0 = p
    1 = q
    2 = r
  Function numbers
    0 = a
  ›

lemma ‹⊩
  [
    Imp (Con (Uni (Pre 0 [Var 0])) (Pre 1 [])) (Dis (Pre 2 []) (Exi (Pre 0 [Var 0])))
  ]
  ›
proof -
  from AlphaImp have ?thesis if ‹⊩
    [
      Neg (Con (Uni (Pre 0 [Var 0])) (Pre 1 [])),
      Dis (Pre 2 []) (Exi (Pre 0 [Var 0]))
    ]
    ›
    using that by simp
  with AlphaCon have ?thesis if ‹⊩
    [
      Neg (Uni (Pre 0 [Var 0])),
      Neg (Pre 1 []),
      Dis (Pre 2 []) (Exi (Pre 0 [Var 0]))
    ]
    ›
    using that by simp
  with GammaUni[where t=‹Fun 0 []›] have ?thesis if ‹⊩
    [
      Neg (Pre 0 [Fun 0 []]),
      Neg (Pre 1 []),
      Dis (Pre 2 []) (Exi (Pre 0 [Var 0]))
    ]
    ›
```

More problems with complexity

- The full power of Isabelle is too much for an exam situation
- Extremely difficult to find theorems which are too difficult for automation but possible to solve within 15–30 minutes
- Solution: make students “fix” proofs

Example

```
section ‹Problem 5 - 20%›

subsection ‹Question 1›

text ‹ Replace \<proof> with the "proof ... qed" lines in the following comment and correct the errors
such that the structured proof is a proper proof in Isabelle/HOL (do not alter the lemma text). ›

lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
  \<proof>

(*

proof (rule ccontr)
  assume ‹∃x. ¬ p x›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p x›
    proof (rule ccontr)
      show ‹¬ p x›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› assume ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊤ ..
qed

*)
```

Example

```
lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
proof (rule ccontr)
  assume ‹∄x. ¬ p x›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p x›
    proof (rule ccontr)
      show ‹¬ p x›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› assume ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊤ ..
qed
```

Example

```
lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
proof (rule ccontr)
  assume ‹∃x. ¬ p x›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p a›
    proof (rule ccontr)
      show ‹¬ p a›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› assume ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊤ ..
qed
```

Example

```
lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
proof (rule ccontr)
  assume ‹∄x. ¬ p x›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p a›
    proof (rule ccontr)
      assume ‹¬ p a›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› show ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊤ ..
qed
```

Example

```
lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
proof (rule ccontr)
  assume ‹¬ (∃x. ¬ p x)›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p a›
    proof (rule ccontr)
      assume ‹¬ p a›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› show ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊤ ..
qed
```

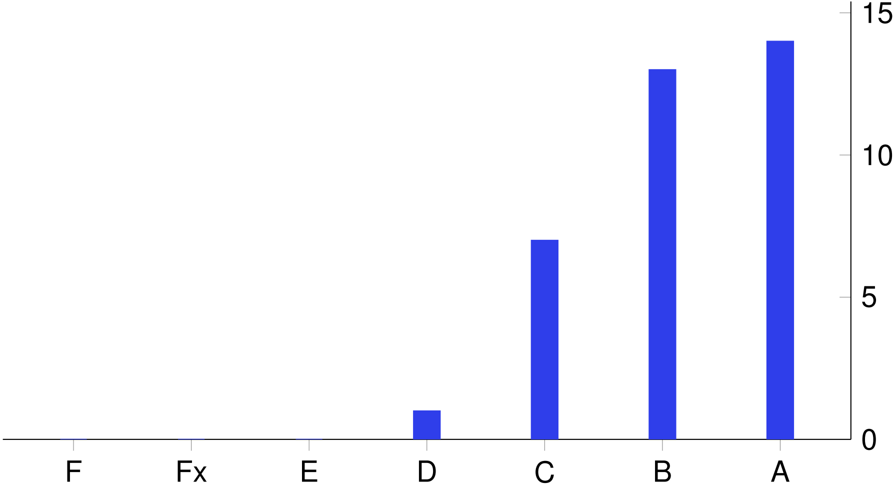
Example

```
lemma Foobar:
  assumes ‹¬ (∀x. p x)›
  shows ‹∃x. ¬ p x›
proof (rule ccontr)
  assume ‹¬ (∃x. ¬ p x)›
  have ‹∀x. p x›
  proof
    fix a
    show ‹p a›
    proof (rule ccontr)
      assume ‹¬ p a›
      then have ‹∃x. ¬ p x› ..
      with ‹¬ (∃x. ¬ p x)› show ⊥ ..
    qed
  qed
  with ‹¬ (∀x. p x)› show ⊥ ..
qed
```

Our experiences with the approach

- Difficult to come up with problems of the right complexity
- Relatively easy to grade submissions
- Students seem to have no problem understanding how to fill in answers and hand in

Grades



Automated grading?

- Difficult because students hand in with many syntax errors
- Some work on this has been done for Coq
- In an exam situation it seems unfeasible

Future work

- How do we design problems with a good level of complexity?
  - Auxiliary tools can help mitigate complexity issues, but require a lot of work to create
  - Project-based exams may be easier to design, but take a long time to create and are difficult to scale up to many students
- Can we make grading more automated?
- Are there other things we should test?