

Teaching Functional Programmers Logic and Metatheory

Frederik Krogsdal Jacobsen Jørgen Villadsen

DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark

We present a novel approach for teaching logic and the metatheory of logic to students who have some experience with functional programming. We define concepts in logic as a series of functional programs in the language of the proof assistant Isabelle/HOL. This allows us to make notions which are often unclear in textbooks precise, to experiment with definitions by executing them, and to prove metatheoretical theorems in full detail. We have surveyed student perceptions of our teaching approach to determine its usefulness and found that students felt that our formalizations helped them understand concepts in logic, and that they experimented with them as a learning tool. However, the approach was not enough to make students feel confident in their abilities to design and implement their own formal systems. Further studies are needed to confirm and generalize the results of our survey, but our initial results seem promising.

1 Introduction

Logic is the foundation on which all of mathematics and computer science rests, and many undergraduate computer science programs therefore include an introductory course on logic. The logical systems introduced in such a course can be applied to databases, domain-specific languages, artificial intelligence, computer security, formal verification, and many other topics. At the Technical University of Denmark (DTU) we teach logic alongside logic programming in Prolog in a late-stage undergraduate course in addition to discrete mathematics taught in the first semester of the study program. Undergraduate courses like ours give students a basic understanding of logic and just enough knowledge to start applying basic logical systems in their work. But for students who really need to work with logic and design their own systems, this is not enough: they also need a good understanding of the metatheory of logic, i.e. why the systems work and how to prove that they do. Proofs about logical systems are fraught with possibilities for subtle mistakes of understanding, and it is our experience that many students never really “get” how the logical systems, and the proofs about them, work. Additionally, many textbooks define logical systems in terms of informal set theory and omit parts of their proofs (sometimes sweeping major complexities such as binders and substitution under the rug), which leads to difficulties in actually applying the theory when implementing real systems. For many students, implementing logical systems in their projects (and making sure that they are correct) can thus seem like an insurmountable challenge.

We have recently begun giving a graduate course (course number 02256) on automated reasoning with course material implemented in the proof assistant Isabelle/HOL. This year the course started with 80 students. The official course listing is available at <https://kurser.dtu.dk/course/02256>.

Isabelle/HOL is the higher-order logic version of the generic proof assistant Isabelle. Briefly, we can consider higher-order logic as the sum of functional programming and logic. Isabelle/HOL allows us to write definitions as functional programs and to formally prove properties of these programs. The proof assistant continuously checks that our proofs are correct, thus making it impossible to neglect any complexities. Additionally, Isabelle/HOL allows us to automatically export appropriate definitions into “real” functional languages such as Haskell, OCaml, Scala and Standard ML, whereby they can be integrated into larger systems.

By defining the logical systems we teach within Isabelle/HOL, we are thus able to give precise and executable definitions of every aspect of the systems, and our proofs of metatheoretical properties such as soundness and completeness of systems relate directly to these precise definitions and are verified by the proof assistant. Our experience is that students appreciate this: if they are in doubt about what something means, they can look up a precise definition and even execute it on examples of their own choosing to gain further understanding, and this should also make it clear how to implement the concepts in practice. To determine whether students actually find our approach useful, we survey student perceptions of the components of our course and of their own abilities and behaviour. Our hypotheses are that:

1. Concrete implementations in a programming language aid understanding of concepts in logic.
2. Students experiment with definitions to gain understanding.
3. Our formalizations make it clear to students how to implement the concepts in practice.
4. Our course makes students able to design and implement their own logical systems.
5. Prior experience with functional programming is useful for our course.
6. Our course helps students gain proficiency in functional programming.

In summary, we contribute:

- functional implementations of several logical systems which can be used to teach topics such as sequent calculus, natural deduction, de Bruijn indices, and algorithms for automatic theorem proving.
- formally verified proofs of common properties such as soundness and completeness for the systems mentioned above.
- an evaluation of the usefulness of our approach based on surveying student perceptions.

In the next section, we survey related work. In Section 3 we explain our teaching approach using a small verified automated theorem prover as an example. In Section 4 we survey student perceptions of our teaching approach to evaluate its usefulness, before concluding in Section 5.

2 Related work

There are of course numerous textbooks on logic, and several which include implementations of many of their definitions. Examples include books by Harrison [12], Ben-Ari [3], and Doets and van Eijck [6], which all contain implementations in various programming languages. Unfortunately, these implementations are not connected directly with the definitions and proofs in the books, and it is thus sometimes unclear how the programs really relate to the definitions in the text. Additionally, the proofs in these books are not verified by a proof assistant, and are about the informal definitions, not the programs themselves. Language, Logic and Proof [2] is a textbook and accompanying software package containing a number of small proof assistants designed to aid in teaching logic. While this means that students can rest assured that their exercise proofs are formally verified, the proofs in the book itself are not, and it is again sometimes unclear exactly how the software packages relate to the notions introduced in the text. In our approach, every logical system and notion is formally defined, and every theorem is proven in Isabelle/HOL, which allows students to see exactly how to implement concepts and inspect every detail of the proofs.

Some textbooks on logic do have formally verified proofs of their theorems, but these are typically not introductions to logic, but instead introductions to proof assistants that just happen to introduce some

Weeks	Topics
1 – 2	Basic set theory, propositional logic, sequent calculus, automatic theorem proving
3 – 4	Syntax and semantics of first-order logic, natural deduction, the LCF approach
5 – 6	Isar, intuitionistic logic, foundational systems
7 – 8	Proof by contradiction, classical logic, higher-order logic, type theory
9 – 10	Proofs in sequent calculus
11 – 13	Metatheory, prover algorithms, program verification

Table 1: Week plan and topics of our course.

particular logic they are working in. One example is *Coq’Art* [4], which is an introduction to the Coq proof assistant and contains many formally verified proofs about logic and other topics. Unfortunately, such books, being written for a much more advanced audience, are not by themselves good introductions to logic, since they presuppose knowledge beyond the usual computer science program. In our approach, on the other hand, we teach an introduction to logic without presupposing much beyond basic knowledge of functional programming.

The approach of formalizing definitions as functional programs has successfully been used for a number of other topics in computer science. The *Software Foundations* series [22] covers some basic logic, programming languages, formal verification, functional algorithms and separation logic with definitions and proofs in the Coq proof assistant. *Concrete Semantics* [20] is a textbook on programming language semantics with definitions and proofs in Isabelle/HOL. *Functional Algorithms, Verified!* [19] is an introduction to functional data structures and algorithms with definitions and proofs in Isabelle/HOL. *Verified Functional Programming in Agda* [25] is an introduction to functional programming itself, as well as functional algorithms, using the Agda programming language, which supports proofs about the programs written in it.

We have previously written about the contents of our courses [29], about teaching with Isabelle/Pure and Isabelle/HOL [8,9,28], and about individual formalizations [10,24,26,27], but not about the specifics and benefits of teaching logic and metatheory using functional programming to aid understanding, and we have not previously surveyed student perceptions about our approach in a rigorous manner.

3 A glimpse of the teaching approach

As mentioned, our approach is to define the logical systems which are our objects of study as functional programs in Isabelle/HOL. Having done this, we can then define e.g. automatic theorem provers and other derived programs. Isabelle/HOL then allows us to prove results about the implementations directly, e.g. that our automatic theorem provers are sound and complete.

3.1 The structure of the course

Table 1 contains a general overview of the topics we went through in our course in spring 2022. Note that functional programming is integrated in more or less all topics. For a more detailed explanation of the contents of the 2020 and 2021 versions of our course and its place in the overall curriculum, see [29]. For a discussion of our approach to teaching formal methods with Isabelle/HOL, see [14].

The course consists of lectures, exercise sessions, and assignments. This year, the lectures were in-person, but some parts were recorded for later use by the students. The exercise sessions were supervised by teaching assistants who were only available in-person. The lectures primarily covered theoretical topics, but also included tutorials on Isabelle/HOL and related software. The exercise sessions consisted primarily of programming and proving exercises in Isabelle/HOL, with occasional use of other software. There were 6 assignments during the course, and students worked on them individually. The assignments consisted of larger exercises as well as exercises similar to those in the exam.

This year, our graduate course on automated reasoning started with 80 students, but only 43 were left at the time of the survey. This is not unexpected, since our university allows students to deregister for courses within the first month with no consequences. This means that many students spend the first few weeks auditing many courses, and then choose which courses to stay registered to. Almost all of the students registered at the time of the survey stayed on for the rest of the course and registered for the exam. If a student registers for the exam then the course becomes binding for the student and the student must pass the course in order to graduate. Three exam attempts are allowed.

The exam consisted of five problems each comprising two questions, and the students were given two hours to complete it. One of the problems consisted of writing simple functions and proving properties about them in Isabelle/HOL. The two questions were as follows:

- Using only the constructors `0` and `Suc` and no arithmetical operators, define a recursive function `triple :: nat ⇒ nat` and prove `triple n = 3 * n`.
- Using the usual operators `+` and `-`, define two recursive functions `add42 :: int list ⇒ int list` and `sub42 :: int list ⇒ int list` that adds 42 to each integer and subtracts 42 from each integer, respectively, and prove `sub42 (add42 xs) = xs ∧ add42 (sub42 xs) = xs`.

Both questions can be solved by defining the functions and instructing Isabelle to perform induction. While these questions are not very difficult to solve, being able to prove properties of basic functional programs provides the basis for formalizing concepts in logic. From these simple beginnings, we can build an understanding of more complicated functions and proofs such as the example we introduce in the next section. Along the way we of course need to introduce both the logical concepts and the language and proof methods of the Isabelle proof assistant itself.

For a concrete example of the types of exam problems used, in particular on logic and the use of Isabelle, see the extended abstract [15].

The course evaluations and the grade history can be found on the Information tab on the official course listing available at <https://kursen.dtu.dk/course/02256>.

3.2 Logical systems and provers as functional programs

As a simple example, Figure 1 contains a definition of classical propositional logic. We define a datatype of formulas (this is the so-called deep embedding approach), then introduce semantics as a function from truth value assignments (interpretations) and formulas to truth values (we also define a semantics function for sequents, *sc*). We can then define an automatic theorem prover (function *mp* for micro prover) for the system which works by breaking formulas up using a system similar to a sequent calculus. All of these definitions are simple functional programs, and students should thus be able to understand them quite quickly when the ideas behind them are explained.

The programs can be executed directly inside of Isabelle/HOL, but can also be exported to standard functional programming languages, which allows us to use the concepts in practice. The Haskell program in Figure 2 has been automatically generated by Isabelle/HOL from the definitions in Figure 1.

```

datatype 'a form
  = Pro 'a (·) | Falsity (⊥) | Imp 'a form 'a form (infixr (→) 0)

primrec semantics where
  ⟨ semantics i (·n) = i n ⟩ |
  ⟨ semantics - ⊥ = False ⟩ |
  ⟨ semantics i (p → q) = (semantics i p → semantics i q) ⟩

abbreviation ⟨ sc X Y i ≡ (∀ p ∈ set X. semantics i p) → (∃ q ∈ set Y. semantics i q) ⟩

primrec member where
  ⟨ member - [] = False ⟩ |
  ⟨ member m (n # A) = (m = n ∨ member m A) ⟩

lemma member-iff [iff]: ⟨ member m A ⟷ m ∈ set A ⟩
by (induct A) simp-all

primrec common where
  ⟨ common - [] = False ⟩ |
  ⟨ common A (m # B) = (member m A ∨ common A B) ⟩

lemma common-iff [iff]: ⟨ common A B ⟷ set A ∩ set B ≠ {} ⟩
by (induct B) simp-all

function mp where
  ⟨ mp A B (·n # C) [] = mp (n # A) B C [] ⟩ |
  ⟨ mp A B C (·n # D) = mp A (n # B) C D ⟩ |
  ⟨ mp - - (⊥ # ·) [] = True ⟩ |
  ⟨ mp A B C (⊥ # D) = mp A B C D ⟩ |
  ⟨ mp A B ((p → q) # C) [] = (mp A B C [p] ∧ mp A B (q # C) []) ⟩ |
  ⟨ mp A B C ((p → q) # D) = mp A B (p # C) (q # D) ⟩ |
  ⟨ mp A B [] [] = common A B ⟩
by pat-completeness simp-all

termination
by (relation ⟨ measure (λ(-, -, C, D). ∑ p ← C @ D. size p) ⟩) simp-all

theorem main: ⟨ (∀ i. sc (map · A @ C) (map · B @ D) i) ⟷ mp A B C D ⟩
by (induct rule: mp.induct) (simp-all, blast, meson, fast)

definition ⟨ prover p ≡ mp [] [] [p] ⟩

corollary ⟨ prover p ⟷ (∀ i. semantics i p) ⟩
unfolding prover-def by (simp flip: main)

```

Figure 1: An example Isabelle/HOL development defining a propositional logic and an automatic theorem prover for it. We begin by defining formulas with propositions, falsity, and implication as a datatype, then define semantics of formulas as a function. We then define an automatic theorem prover (function *mp*) for the system and prove that it terminates and is sound and complete. Note how every definition is a simple functional program and that Isabelle/HOL allows us to prove properties of these programs directly. Our students study this example very early on in our graduate course.

```

{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Scratch(Form, prover) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^~), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

data Form a = Pro a | Falsity | Imp (Form a) (Form a);

member :: forall a. (Eq a) => a -> [a] -> Bool;
member uu [] = False;
member m (n : a) = m == n || member m a;

common :: forall a. (Eq a) => [a] -> [a] -> Bool;
common uu [] = False;
common a (m : b) = member m a || common a b;

mp :: forall a. (Eq a) => [a] -> [a] -> [Form a] -> [Form a] -> Bool;
mp a b (Pro n : c) [] = mp (n : a) b c [];
mp a b c (Pro n : d) = mp a (n : b) c d;
mp uu uv (Falsity : uw) [] = True;
mp a b c (Falsity : d) = mp a b c d;
mp a b (Imp p q : c) [] = mp a b c [p] && mp a b (q : c) [];
mp a b c (Imp p q : d) = mp a b (p : c) (q : d);
mp a b [] [] = common a b;

prover :: forall a. (Eq a) => Form a -> Bool;
prover p = mp [] [] [] [p];

}

```

Figure 2: Haskell code generated by Isabelle/HOL from the example in Figure 1. Note that it is essentially a direct translation of the relevant Isabelle/HOL definitions, and that the proofs are not included. While the code is not pretty, a module such as this one can easily be integrated with other (handwritten) code to create a full application. This provides an example of how to implement and use the logical systems we cover in our course in practice.

Note that only the functions themselves, and not the proofs about them, are present in the Haskell code. The idea behind this approach is that the difficult parts of a program can be formally verified using Isabelle/HOL, then exported to a “normal” programming language as modules ready for integration into the overall program. This is similar to the LCF approach [13, 21] to theorem proving and the “hexagonal architecture” pattern common in object-oriented and functional programming [5], both of which advocate for programs consisting of a core application which interacts with users and other programs through an outer layer. In this case, the core can be implemented and verified in Isabelle/HOL, then exported, while the communication layer can be implemented in the target programming language.

3.3 Metatheory as properties of programs

Once we have defined a logical system in Isabelle/HOL, we can begin proving results about it. In Figure 1, we first prove that the functions *member* and *common* are correct by simple structural induction. Note that the proofs very much resemble the classic “The proof is by induction” often found in textbooks, but that the proof has been verified by Isabelle/HOL. This allows the reader to rest easy knowing that there is no hidden complexity in the proof.

We then prove termination of our prover by noting that the sum of the sizes of the two last arguments decrease. Isabelle/HOL requires that all functions are total, and in this case the proof is complicated enough that we have to prove it manually, but simple enough that we can do so easily. Next, we prove that our automatic theorem prover returns true if and only if the provided sequent is valid (theorem *main*). This proof is by induction, and the automation in Isabelle/HOL is enough to handle all cases. Finally, we conclude that our prover returns true for a formula exactly when it is valid (true in all interpretations), which means that the prover is sound and complete.

In this fashion, we can prove metatheoretical results about several logical systems used in our course, and thus showcase the proof techniques needed. We are also able to explore and prove equivalences between different logical systems by proving that the programs implementing them return the same results.

4 Student perceptions of the teaching approach

To attempt to shed some light on whether our approach is actually useful, we have conducted a survey study to analyze student perceptions about our course. The survey consisted of a single self-administered questionnaire, which all students following the course were invited to fill in during the tenth week of the course.

4.1 Methods

The study was designed to address the following six hypotheses:

1. Concrete implementations in a programming language aid understanding of concepts in logic.
2. Students experiment with definitions to gain understanding.
3. Our formalizations make it clear to students how to implement the concepts in practice.
4. Our course makes students able to design and implement their own logical systems.
5. Prior experience with functional programming is useful for our course.
6. Our course helps students gain proficiency in functional programming.

To address these hypotheses, we developed 12 questions to measure student perception of the course and their own behaviour and abilities. The questions can be seen on the left in Figure 3 or in the appendix. The questions were developed following evidence-based best practices for questionnaire item question design as described in [16], but the consistency and reproducibility of answers to the questionnaire items have not been tested.

Questions were close-ended and answers were given in terms of perception on one of three Likert-type [17] scales (multiple answers not possible) based on established best practice response options [16]: **An importance scale** consisting of the levels: "Not important", "Slightly important", "Moderately important", "Quite important", and "Essential".

A frequency scale consisting of the levels: "Almost never", "Once in a while", "Sometimes", "Often", and "Almost always".

A confidence scale consisting of the levels: "Not at all confident", "Slightly confident", "Moderately confident", "Quite confident", and "Completely confident".

Since the questionnaire consisted almost exclusively of attitudinal questions, we did not include a "Don't know" option for any question since this incentivizes partial responses [11]. "Don't know" responses were thus not possible, and students were required to fill in the entire questionnaire to submit it, which means that partial responses were not possible.

Since the survey consisted of a single questionnaire which was administered only once, we have only one sample (but note that question 10 asks about previous perception, which we analyze as a separate sample). The survey population was all 43 students following the course during course week 10. To preserve student anonymity, the sampling frame does not include auxiliary information such as gender, age, grades, etc. Specifically, this means that we cannot connect student answers to the questionnaire with measures of learning outcomes. We expect that there is some self-selection bias, since some students officially follow the course but do not actually show up to class, and are thus unlikely to answer the questionnaire. Additionally, we expect that students who either really like or really dislike the course will be more likely to answer the questionnaire, which could also result in some self-selection bias.

The questionnaire was administered using an online surveying solution provided by our university. This let us ensure that each student was only able to fill in the questionnaire once, and that participating students were anonymous. Participation in the survey was optional and not compensated financially or otherwise. The students were asked to participate in several lectures and through email.

4.2 Data analysis

The questionnaire has been designed to answer the hypotheses listed above. In this section, we will describe how we intend to confirm or reject each of the hypotheses based on the answers to the questions. Since we did not allow partial responses, we do not need to handle missing data for individual questions.

All of our data analysis was performed using the R environment for statistical computing [23]. The data from the questionnaire was pre-processed using the readr package [30]. The analysis script is available at <https://github.com/fkj/tfpie-2022-statistics>.

4.2.1 Concrete implementations in a programming language aid understanding of concepts in logic

Questions 1–3 ask about the perceived importance of the three elements of the course. We would like to compare the relative importance of the three elements to determine whether the implementations play an important part in gaining understanding.

If we had a measure of student learning outcomes, it would be interesting to perform a relative importance analysis on the three factors, but due to the anonymity of the questionnaire, we are not able to couple it to e.g. grades. Instead, we simply qualitatively compare the results of the questionnaire.

4.2.2 Students experiment with definitions to gain understanding

Question 4 asks whether students think it is important to experiment with their own examples when learning about a new concept. Question 5 asks whether students actually evaluate concrete examples to gain understanding when they are using Isabelle. We can look at the frequencies and median answers to gain an understanding of student behaviour.

4.2.3 Our formalizations make it clear to students how to implement the concepts in practice

Question 6 asks directly about how the students perceive their abilities to do this. We can look at the frequencies and median answer to gain an understanding of student perceptions.

4.2.4 Our course makes students able to design and implement their own logical systems

The meaning of this hypothesis is a bit vague, so we divide it into multiple concrete questions. Question 7 asks about perception of ability to design a formal system to solve a practical problem. Question 8 asks about perception of ability to implement a formal system in any programming language (without proofs). Question 9 asks about perception of ability to prove a formal system correct. We can look at the frequencies and median answers to gain an understanding of student perceptions.

4.2.5 Prior experience with functional programming is useful for our course

Question 10 asks about perceived ability with functional programming before starting the course and we will measure the association with perceived ability during the exercises and assignments in the course, which we ask about in question 11. Since both categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in question 10 as a predictor for question 11, we will use Somers' Delta statistic [1] to measure the association [18]. We can also look at the frequencies and median answers to gain an understanding of student perceptions.

4.2.6 Our course helps students gain proficiency in functional programming

Question 10 asks about perceived ability before starting the course, while question 12 asks about perceived ability towards the end of the course. We can test whether the probability of high perceived ability has increased during the course by comparing the two answers for each student. Since we have two groups with paired observations (i.e. the same students twice), we can perform a two-sample paired signed-rank test [18] (i.e. a paired Wilcoxon signed-rank test) to determine whether there is a significant difference in perceived functional programming ability before and after the course [18]. The effect size of the paired Wilcoxon signed-rank test can be measured by the matched-pairs rank biserial correlation coefficient [18].

4.3 Results

21 students completed the entire questionnaire (48.84% response rate). The margin of error is thus below 15.48% (95% CI, p set to 0.5 to obtain worst-case margin). A summary of the answers can be seen in Figure 3, and the full data set is available in the appendix. Based on these answers, we will present the result of the analyses mentioned for each hypothesis above.

4.3.1 Concrete implementations in a programming language aid understanding of concepts in logic

Comparing the answers to question 1–3, we see that the median student thinks that:

1. the implementations in Isabelle are “Quite important” for their understanding of the course topics
2. the reading material is “Moderately important” for their understanding of the course topics
3. the lectures are “Slightly important” for their understanding of the course topics

This indicates that students believe that the implementations in Isabelle are the most important component of the course when it comes to their understanding of the course topics. While we do not have access to an association between these beliefs and actual learning outcomes, this data suggests that the hypothesis is plausible.

4.3.2 Students experiment with definitions to gain understanding

Looking at the answers to question 4, we see that all students find experiments with their own examples at least slightly important, while most students find them at least quite important. The median student finds experiments with their own examples “Essential” when learning about a new concept.

Looking at the answers to question 5, we see that there seems to be two distinct groups: one group which “Almost never” evaluates concrete examples in Isabelle, and a group which does so quite often. The median student evaluates concrete examples in Isabelle “Quite often”, which confirms our hypothesis.

4.3.3 Our formalizations make it clear to students how to implement the concepts in practice

Looking at the answers to question 6, we see that most students are not at all, or only slightly confident, but that there are also a number of students who are quite, or completely confident. The median student, however, is only “Slightly confident” that they could implement the concepts in practice. This suggests to reject the hypothesis.

4.3.4 Our course makes students able to design and implement their own logical systems

Looking at the answers to question 7, we see that the general trend is that student do not feel that they have the ability to design formal systems to prove practical problems. The median student is only “Slightly confident” in their ability to do this. This suggests to reject the hypothesis.

Looking at the answers to question 8, we see that the general trend is that students do not feel that they have the ability to implement their own logical systems (in any programming language). The median student is only “Slightly confident” in their ability to do this. This suggests to reject the hypothesis.

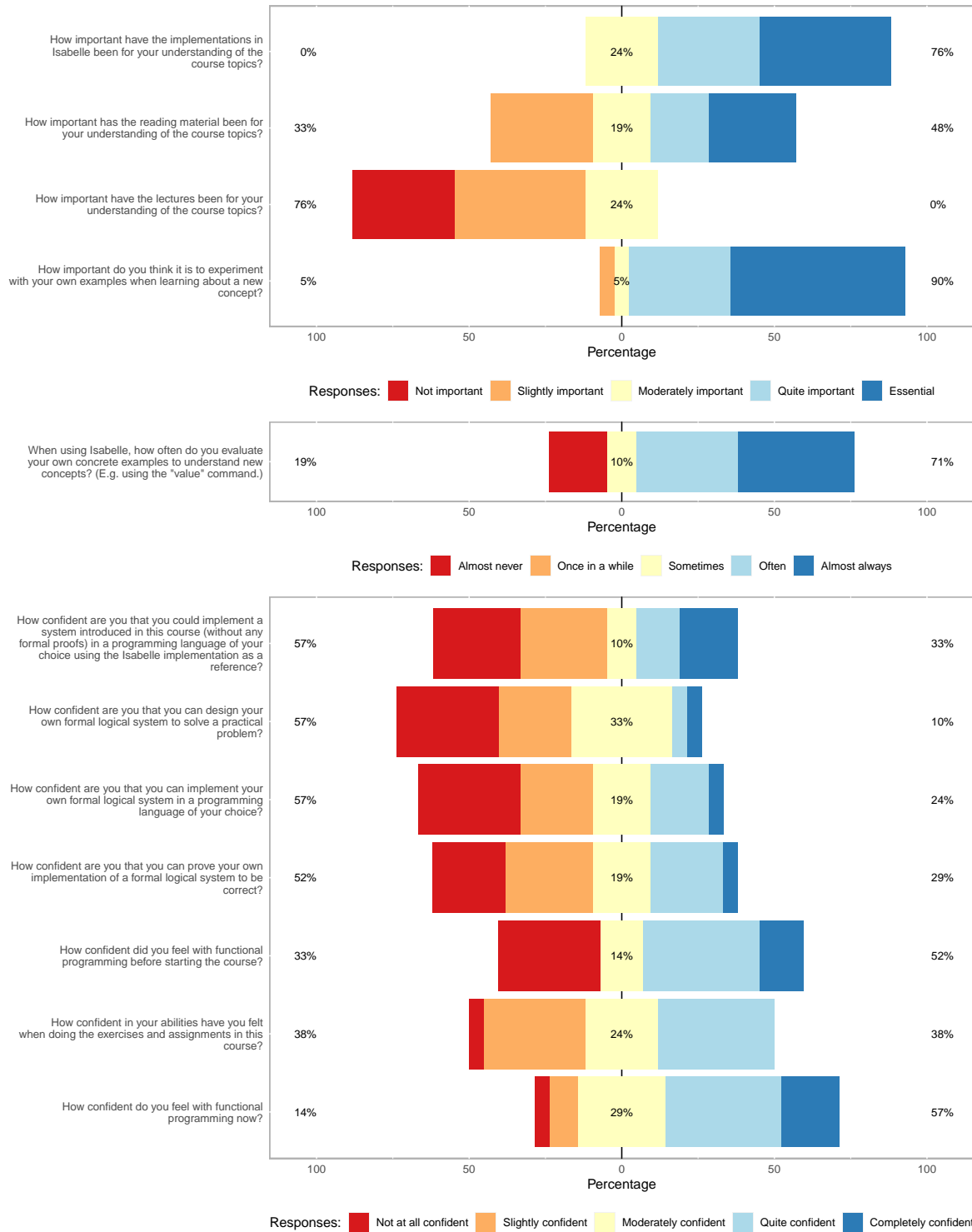


Figure 3: Summary of answers to the student self-evaluation questionnaire (n = 21). See the appendix for the full data set.

Looking at the answers to question 9, we see that the general trend is that students are only slightly to moderately confident that they have the ability to prove a formal system correct. The median student is only “Slightly confident” in their ability to do this. This suggests to reject the hypothesis.

Overall, the data suggests to reject the hypothesis.

4.3.5 Prior experience with functional programming is useful for our course

Looking at the answers to question 10, we see that there is an approximately normal distribution of students who felt moderately to completely confident as well as a second mode of students who were not at all confident with functional programming before starting the course. We know that this second mode consists of students who did not have any experience with functional programming prior to the course (although it was an explicit prerequisite). The median student felt “Quite confident” with functional programming before starting the course.

Looking at the answers to question 11, we see that nobody felt completely confident, while most students felt slightly to quite confident when doing the exercises and assignments in the course. The median student felt “Moderately confident” when doing the exercises and assignments.

Somers’ Delta statistic for the association of question 10 as a predictor for question 11 is 0.3642 (95% CI [0.0038,0.7247]), which suggests a small to moderate association between students who felt confident with functional programming before starting the course and students who felt confident during the exercises and assignments [7]. This suggests to confirm the hypothesis.

4.3.6 Our course helps students gain proficiency in functional programming

Looking at the answers to question 10, we see that a number of students had no confidence with functional programming before starting the course (presumably due to them missing the functional programming prerequisite). Looking at the answers to question 12, we see that after the course, we obtain what looks more like a normal distribution centered around the “Quite confident” answer (although it is quite skewed).

The two-sample paired signed-rank test gives $V = 2$ (95% CI $[-3.5, 0.00]$), with $p = 0.0498$. Since $p \leq 0.05$, we conclude that there is a statistically significant difference in perceived functional programming ability before and after the course [18]. The effect size as measured by the matched-pairs rank biserial correlation is -0.857 (95% CI $[-1.00, -0.333]$), which suggests that the course has a large positive effect on perceived functional programming ability [18]. This suggests to confirm the hypothesis.

4.4 Discussion

We begin with a summary of our results:

Plausible: Concrete implementations in a programming language aid understanding of concepts in logic.

Confirmed: Students experiment with definitions to gain understanding.

Rejected: Our formalizations make it clear to students how to implement the concepts in practice.

Rejected: Our course makes students able to design and implement their own logical systems.

Confirmed: Prior experience with functional programming is useful for our course.

Confirmed: Our course helps students gain proficiency in functional programming.

Our first result suggests that our students feel that our teaching approach does help them understand concepts in logic. Unfortunately, our study does not have the data to confirm that they actually do, since we do not have access to assessments of learning outcomes linked to student responses. We also confirm that students do in fact use the formal definitions as learning tools by experimenting with examples and definitions within Isabelle.

On the other hand, students did not feel that our formalizations were enough to make it clear how to design and implement their own logical systems, or the concepts in the course in general. While we did not expect that students would feel very confident doing this (since formal verification is generally very difficult), we had still hoped for more confident answers. This suggests that we should attempt to find ways to improve our course in this respect. We also note that students do not seem to think that our lectures are very useful, which suggests that we should consider whether they are necessary or if even more time should be spent on exercises. By reducing the time spent on lectures, it may also be possible to obtain time for some project work, which may help students become more confident in designing and implementing their own logical systems.

We also found that students with more functional programming experience before starting the course felt more confident during the course, which is as expected. Our data also suggests that our course has a large positive effect on self-perceived functional programming confidence (though this may be a result of some students having no prior functional programming experience, see Section 4.4.1).

Our study is quite limited in scope, and the confidence intervals on our results are thus not very narrow. A larger study is needed to obtain precise knowledge of effect sizes. Since students participated in the survey of their own volition and with no compensation, we expect that our results also suffer from some self-selection bias. Further studies are needed to determine whether our results generalize.

4.4.1 Post hoc exploratory analyses

Having seen the answers to our survey, we notice some phenomena that could be interesting to explore further. Note that the analyses in this section are post hoc, i.e. they have been formulated after seeing the data, and that further studies are thus necessary to confirm any effects described in this section.

In Section 4.3.2, we noted that the answers to question 5 seem to contain two distinct groups. We can think of two possible reasons that a group of students almost never experiment with concrete examples in Isabelle: either there is a group of students who simply had not understood that this is possible, or some students do not find experimentation valuable enough to actually do it outside of the abstract. To investigate this further, we can look at the distribution of answers to whether students think that experimentation is important for students who answered that they almost never evaluate concrete examples in Isabelle and students who answered otherwise. We can split the data set in "Almost never" and "Other" and look at the distribution of how important students think experimentation is for each category. By doing this, we see that the shapes of the two distributions do not seem to differ significantly.

We can also measure the association between students who think that experimentation is important and students who often evaluate concrete examples in Isabelle. Since both categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in question 4 as a predictor for question 5, we will use Somers' Delta statistic [1] to measure the association [18]. The statistic is -0.1039 (95% CI $[-0.4777, 0.2699]$), which shows a small negative association between the variables, i.e. that students who find experimentation more important evaluate concrete examples slightly less often than those who find it less important. This is surprising. If we assume that the answers of "almost never" are due to students not knowing about the feature, and thus remove them from the data set as

special cases, the association becomes -0.0814 (95% CI $[-0.5046, 0.3418]$), which again shows a small negative association. While there is not enough data to give good confidence intervals, it seems like there is no statistically significant association between the perceived importance of experimentation and the frequency of evaluating concrete examples in Isabelle. Note, however, that students still generally tend to believe that experimentation is important, and that they generally tend to evaluate their own concrete examples in Isabelle. One possible explanation is that some students who believe that experiments are essential are used to doing experiments with pen and paper, and thus do this instead of using Isabelle to evaluate examples, but we do not have any data to investigate this hypothesis in this study.

In Section 4.3.3, we noted that the answers to question 6 seem to contain two distinct groups. The bimodality is interesting, since we know that many students did not have experience with functional programming prior to our course (even though it was an explicitly stated prerequisite). We can test whether there is an association between perceived functional programming ability and perceived ability to implement the concepts in practice. We have data for perceived programming ability both before and after the course, so we can analyze both. Since all categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in questions 10 and 12 as predictors for question 6, we will use Somers' Delta statistic [1] to measure the associations [18]. For previous functional programming ability, the statistic is 0.194 (95% CI $[-0.139, 0.527]$). This suggests that there is an association, but that it is barely significant [7]. For current functional programming ability, the statistic is 0.394 (95% CI $[0.0807, 0.707]$). This suggests a small to moderate, but statistically significant association [7].

We conclude that it seems that our formalizations may help students understand how to implement the concepts in practice, but only significantly so if they were confident functional programmers after following the course. Further studies are needed to confirm this effect, since the analysis above is post hoc.

In Section 4.3.4, we concluded that students do not generally feel confident in their abilities to implement their own logical systems. It could be interesting to measure the association between perceived functional programming ability and perceived ability to implement logical systems. We have data for perceived programming ability both before and after the course, so we can analyze both. Since both categories are ranked from low to high, there is no possibility of multiple choices, and we are interested in questions 10 and 12 as predictors for question 8, we will use Somers' Delta statistic [1] to measure the associations [18]. For previous functional programming ability, the statistic is 0.108 (95% CI $[-0.259, 0.474]$). This suggests that there is no statistically significant association [7]. For current functional programming ability, the statistic is 0.419 (95% CI $[0.0445, 0.794]$). This suggests a small to moderate association [7].

We conclude that it seems that our formalizations may help students understand how to implement logical systems, but only if they are confident functional programmers after the course, and only slightly so.

In Section 4.3.5, we noted that the answers to question 10 seem to have two modes. If we look at the two modes independently, the students with the functional programming prerequisite tend to feel moderately to quite confident while most students without the functional programming prerequisite felt only slightly confident. This also supports the original hypothesis.

We can try to remove the students with no functional programming experience and measure the association between perceived ability with functional programming before starting the course and perceived ability during the exercises and assignments in the course again to see whether the course requires advanced functional programming skills. We will again use Somers' Delta statistic to measure association [18]. The value is 0.0000 (95% CI $[-0.5019, 0.5019]$), which we interpret as meaning that only basic knowledge of functional programming seems to be needed in our course. This tracks well with our design choices in our systems, where we deliberately avoid "advanced" concepts such as folds in our implementations and exercises in an effort to obtain this result.

In Section 4.3.6, we found that the course has a significant positive effect on perceived functional programming ability. We can try to remove the students with no functional programming experience to see whether there is still a significant difference for those students who already have functional programming experience. We again perform a two-sample paired signed-rank test (i.e. a paired Wilcoxon signed-rank test) to determine whether there is a significant difference in perceived functional programming ability before and after the course [18]. The test shows $V = 1$, with $p = 1$. This means there is no statistically significant difference in perceived functional programming ability for the group of students who already have functional programming experience. In fact, by inspection of the data set we see that every student has answered exactly the same for both questions, except one student who moved from feeling "Quite confident" to feeling only "Moderately confident" after having taken the course. The lack of effect on perceived functional programming ability for students who already have some experience is to be expected since our course does not utilize any advanced concepts in functional programming.

4.5 Future work

Our results suggest that our approach seems to aid student understanding of logic, but that students do not feel that our formalizations are enough to make it clear how to implement the concepts in practice, and that students do not feel capable of designing and implementing their own logical systems after the course. This suggests that we should attempt to find ways to make it more clear how to do this in future iterations of the course. One possible way to do this would be to introduce a project about implementing some system to solve a practical problem, but this would almost certainly require us to extend the course load. It may also be possible to replace some assignments and exercises during the course with more practically oriented ones. Additionally, we may be able to reduce the time spent on lectures, since students do not seem to find them very important for their learning outcomes.

Our survey consisted almost entirely of attitudinal questions, which means that we are forced to believe student self-perceptions to some degree. We could improve this aspect by assessing e.g. understanding of logic and experience with functional programming by learning outcome measures such as previous grades or tests during the survey. Adding tests of aptitude to the survey would make completing it a much larger endeavour, however, which might decrease the number of students willing to participate and introduce additional self-selection bias. Including measures such as previous grades would require us to handle non-anonymous data, which makes the survey a much larger undertaking, since we would then need to collect informed consent letters, obtain institutional review board approval, etc. The same is true for demographic data such as gender, age, etc.

The generalizability of our survey may be doubted, since it may have issues of self-selection bias, and since our sample size was not large enough to obtain a narrow margin of error. Self-selection bias may be decreased by screening participants to ensure that our sample is representative of the population, but this will also require collection of demographic data and possibly previous grades, and will most likely

reduce the sample size. Another way to reduce self-selection bias is to collect data on every student, but this has obvious ethical issues. In the future, our survey could be repeated on other cohorts to increase the generalizability of our results.

Since our course is new, we have no data from previous pen-and-paper based courses at our university to compare our approach to. Performing such a comparison in a rigorous way would also be difficult, since it is not obvious how to choose a test of student learning outcomes that is comparable for both pen-and-paper approaches and our approach.

Finally, it may be interesting to conduct further studies to determine whether the effects discussed in Section 4.4.1 actually exist. We have noted the following interesting questions which arise from phenomena in our sample, but which, being the result of post hoc analyses, cannot be confirmed from the answers to the present survey:

- Why do students who think experimentation is important do it less? Do they do it on paper instead?
- Does functional programming experience play a significant role in understanding of how to implement concepts in practice?
- Does functional programming experience play a significant role in understanding of how to design and implement one's own logical systems?
- Does our course have a positive effect on functional programming skill for students who are already confident functional programmers?

5 Conclusion

We have presented our approach for teaching logic and metatheory using functional programming and demonstrated how logical concepts can be defined as simple functional programs about which we can prove metatheoretical results in a natural way. We have surveyed our students to determine their perceptions of our teaching approach. Our survey shows that students feel that our formalizations do aid them in understanding concepts in logic. We also confirm that students do in fact use the formal definitions as learning tools by experimenting with examples and definitions within Isabelle. On the other hand, students did not feel that our formalizations were enough to make it clear how to design and implement their own logical systems, or the concepts in the course in general. This suggests that we should attempt to find ways to make this clearer in future iterations of the course. Our survey also revealed a number of interesting phenomena which may be interesting to pursue further in future studies. Future studies are also needed to improve the generalizability of our results. We believe that our study design can serve as a useful methodology for such further studies.

For now, our course material consists of a series of lecture notes and corresponding formalizations and student exercise templates in Isabelle/HOL, and we combine this with excerpts from textbooks and tutorials. It would be great to have a comprehensive and coherent textbook written with this style of teaching in mind, but our course material has not yet stabilized enough for us to consider writing one. We would like to make clear that a book containing nothing but programs is not what we advocate; instead, we would like a textbook containing explanations based on, and corresponding to, the precise definitions in the functional programs, such that readers can always clarify any doubts about the “intuitive” explanations given in the text by referring to the programs implementing the definitions. Similarly, we do not believe that textbooks should contain meticulous proofs going into every detail of every case, but that such proofs should be available to the reader if they are not convinced by the abbreviated arguments given in the text.

Acknowledgements

We thank Agnes Moesgård Eschen, Asta Halkjær From, Alceste Scalas, Michael Reichhardt Hansen and Simon Tobias Lund for comments on drafts. We also thank the anonymous reviewers for their useful suggestions.

References

- [1] Signorell Andri et mult. al. (2021): *DescTools: Tools for Descriptive Statistics*. Available at <https://cran.r-project.org/package=DescTools>. R package version 0.99.44.
- [2] David Barker-Plummer, Jon Barwise & John Etchemendy (2011): *Language, Proof and Logic*, second edition. Center for the Study of Language and Information.
- [3] Mordechai Ben-Ari (2012): *Mathematical Logic for Computer Science*. Springer, doi:10.1007/978-1-4471-4129-7.
- [4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development*. Springer, Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.
- [5] Alistair Cockburn (2005): *Hexagonal architecture*. Available at <https://alistair.cockburn.us/hexagonal-architecture/>. Blog article.
- [6] Kees Doets & Jan van Eick (2012): *The Haskell Road to Logic, Maths and Programming*, second edition. College Publications.
- [7] Christopher J. Ferguson (2009): *An Effect Size Primer: A Guide for Clinicians and Researchers*. *Professional Psychology: Research and Practice* 40, pp. 532–538, doi:10.1037/a0015808.
- [8] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen (2019): *Teaching a Formalized Logical Calculus*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019, EPTCS 313*, pp. 73–92, doi:10.4204/EPTCS.313.5.
- [9] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020, EPTCS 328*, pp. 18–34, doi:10.4204/EPTCS.328.2.
- [10] Asta Halkjær From, Anders Schlichtkrull & Jørgen Villadsen (2021): *A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL*. In Stefania Monica & Federico Bergenti, editors: *Proceedings of the 36th Italian Conference on Computational Logic - CILC 2021, Parma, Italy, September 7-9, 2021, CEUR Workshop Proceedings 3002*, CEUR-WS.org, pp. 107–121. Available at <http://ceur-ws.org/Vol-3002/paper7.pdf>.
- [11] Jason Harlacher (2016): *An educator’s guide to questionnaire development (REL 2016-108)*. Technical Report, Washington, DC: U.S. Department of Education, Institute of Education Sciences, National Center for Education Evaluation and Regional Assistance, Regional Educational Laboratory Central. Available at <https://ies.ed.gov/ncee/edlabs/regions/central/resources/pemtoolkit/pdf/module-6/CE5.3.2-An-Educators-Guide-to-Questionnaire-Development.pdf>.
- [12] John Harrison (2009): *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, doi:10.1017/CBO9780511576430.
- [13] John Harrison, Josef Urban & Freek Wiedijk (2014): *History of Interactive Theorem Proving*. In Jörg H. Siekmann, editor: *Computational Logic, Handbook of the History of Logic 9*, North-Holland, pp. 135–214, doi:10.1016/B978-0-444-51624-4.50004-6. Available at <https://www.sciencedirect.com/science/article/pii/B9780444516244500046>.

- [14] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *Lessons of Teaching Formal Methods with Isabelle*. Isabelle Workshop. Available at https://files.sketis.net/Isabelle_Workshop_2022/Isabelle_2022_paper_9.pdf.
- [15] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *On Exams with the Isabelle Proof Assistant*. 11th International Workshop on Theorem proving components for Educational software. Available at <https://www.uc.pt/en/congressos/thedu/ThEdu22>. Extended Abstract.
- [16] Anthony R. Artino Jr., Jeffrey S. La Rochelle, Kent J. Dezee & Hunter Gehlbach (2014): *Developing questionnaires for educational research: AMEE Guide No. 87*. *Medical Teacher* 36(6), pp. 463–474, doi:10.3109/0142159X.2014.889814.
- [17] Rensis Likert (1932): *A Technique for the Measurement of Attitudes*. *Archives of Psychology* 140, pp. 1–55.
- [18] Salvatore S. Mangiafico (2016): *Summary and Analysis of Extension Program Evaluation in R*, 1.19.10 edition. Rutgers Cooperative Extension. Available at <https://rcompanion.org/handbook>.
- [19] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer & Bohua Zhan (2021): *Functional Algorithms, Verified!* Available at <https://functional-algorithms-verified.org/>.
- [20] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics*. Springer, Cham, doi:10.1007/978-3-319-10542-0.
- [21] Lawrence C. Paulson, Tobias Nipkow & Makarius Wenzel (2019): *From LCF to Isabelle/HOL*. *Formal Aspects of Computing* 31(6), pp. 675–698, doi:10.4204/EPTCS.118.4.
- [22] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg & Brent Yorgey (2017): *Software Foundations*. Electronic textbook. <http://www.cis.upenn.edu/~bcpierce/sf>.
- [23] R Core Team (2022): *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. Available at <https://www.R-project.org/>.
- [24] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2018): *Students’ Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018, EPTCS 290*, pp. 1–13, doi:10.4204/EPTCS.290.1.
- [25] Aaron Stump (2016): *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, doi:10.1145/2841316.
- [26] Jørgen Villadsen (2020): *Tautology Checkers in Isabelle and Haskell*. In Francesco Calimeri, Simona Perri & Ester Zumpano, editors: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020, CEUR Workshop Proceedings 2710*, CEUR-WS.org, pp. 327–341. Available at <http://ceur-ws.org/Vol-2710/paper21.pdf>.
- [27] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018, EPTCS 290*, pp. 14–29, doi:10.4204/EPTCS.290.2.
- [28] Jørgen Villadsen, Asta Halkjær From & Patrick Blackburn (2022): *Teaching Intuitionistic and Classical Propositional Logic Using Isabelle*. In João Marcos, Walther Neuper & Pedro Quaresma, editors: *Proceedings 10th International Workshop on Theorem Proving Components for Educational Software, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, Electronic Proceedings in Theoretical Computer Science 354*, Open Publishing Association, pp. 71–85, doi:10.4204/EPTCS.354.6.
- [29] Jørgen Villadsen & Frederik Krogsdal Jacobsen (2021): *Using Isabelle in Two Courses on Logic and Automated Reasoning*. In João F. Ferreira, Alexandra Mendes & Claudio Menghi, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 117–132, doi:10.1007/978-3-030-91550-6_9.
- [30] Hadley Wickham, Jim Hester & Jennifer Bryan (2022): *readr: Read Rectangular Text Data*. <https://readr.tidyverse.org>, <https://github.com/tidyverse/readr>.

Appendix: Survey data

To save space, the survey data has been compressed using numeral encodings. The questions are numerically encoded according to the following table, which also shows the Likert-type scale used for their response options (see subsection 4.1 for details).

#	Question	Scale
1	How important have the implementations in Isabelle been for your understanding of the course topics?	Importance
2	How important has the reading material been for your understanding of the course topics?	Importance
3	How important have the lectures been for your understanding of the course topics?	Importance
4	How important do you think it is to experiment with your own examples when learning about a new concept?	Importance
5	When using Isabelle, how often do you evaluate your own concrete examples to understand new concepts? (E.g. using the "value" command.)	Frequency
6	How confident are you that you could implement a system introduced in this course (without any formal proofs) in a programming language of your choice using the Isabelle implementation as a reference?	Confidence
7	How confident are you that you can design your own formal logical system to solve a practical problem?	Confidence
8	How confident are you that you can implement your own formal logical system in a programming language of your choice?	Confidence
9	How confident are you that you can prove your own implementation of a formal logical system to be correct?	Confidence
10	How confident did you feel with functional programming before starting the course?	Confidence
11	How confident in your abilities have you felt when doing the exercises and assignments in this course?	Confidence
12	How confident do you feel with functional programming now?	Confidence

The responses to our questionnaire are numerically encoded in the following table according to the position on the Likert scale of the question, with 1 being the lowest possible level and 5 being the highest possible level.

Question #	Answers (each column contains the answers from a single student)																				
1	5	4	4	5	3	5	3	4	5	5	4	4	3	4	4	5	5	5	3	5	3
2	5	5	3	3	4	5	4	3	5	2	4	2	5	2	2	2	3	2	5	2	4
3	3	2	2	3	2	2	2	3	2	2	1	1	3	1	1	2	1	1	1	3	2
4	5	2	4	4	5	5	4	5	4	5	5	5	4	4	4	3	5	5	5	5	5
5	5	5	4	5	4	4	4	1	4	5	3	1	5	3	5	1	4	1	5	5	4
6	5	2	1	1	5	3	1	5	4	1	2	2	2	2	4	1	1	2	3	4	5
7	5	1	1	1	3	3	1	3	4	2	2	2	3	2	1	3	1	1	3	3	2
8	5	1	1	1	3	3	1	4	4	2	2	1	2	2	1	4	1	2	3	4	3
9	5	1	1	1	2	4	2	4	3	2	3	2	3	4	4	3	1	1	4	2	2
10	1	3	4	3	5	4	1	4	5	1	4	5	4	4	3	4	1	1	1	1	4
11	3	4	4	3	3	4	1	4	4	2	4	2	3	3	2	2	2	4	2	2	4
12	5	3	4	3	5	4	2	4	5	3	3	5	4	4	3	4	1	3	4	2	4