The Design and Use of Tools for Teaching Logic

Citation for published version (APA):

Lodder, J. S. (2020). *The Design and Use of Tools for Teaching Logic*. Open Universiteit.

Document status and date:
Published: 04/09/2020

Document Version:
Publisher's PDF, also known as Version of record

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

https://www.ou.nl/taverne-agreement

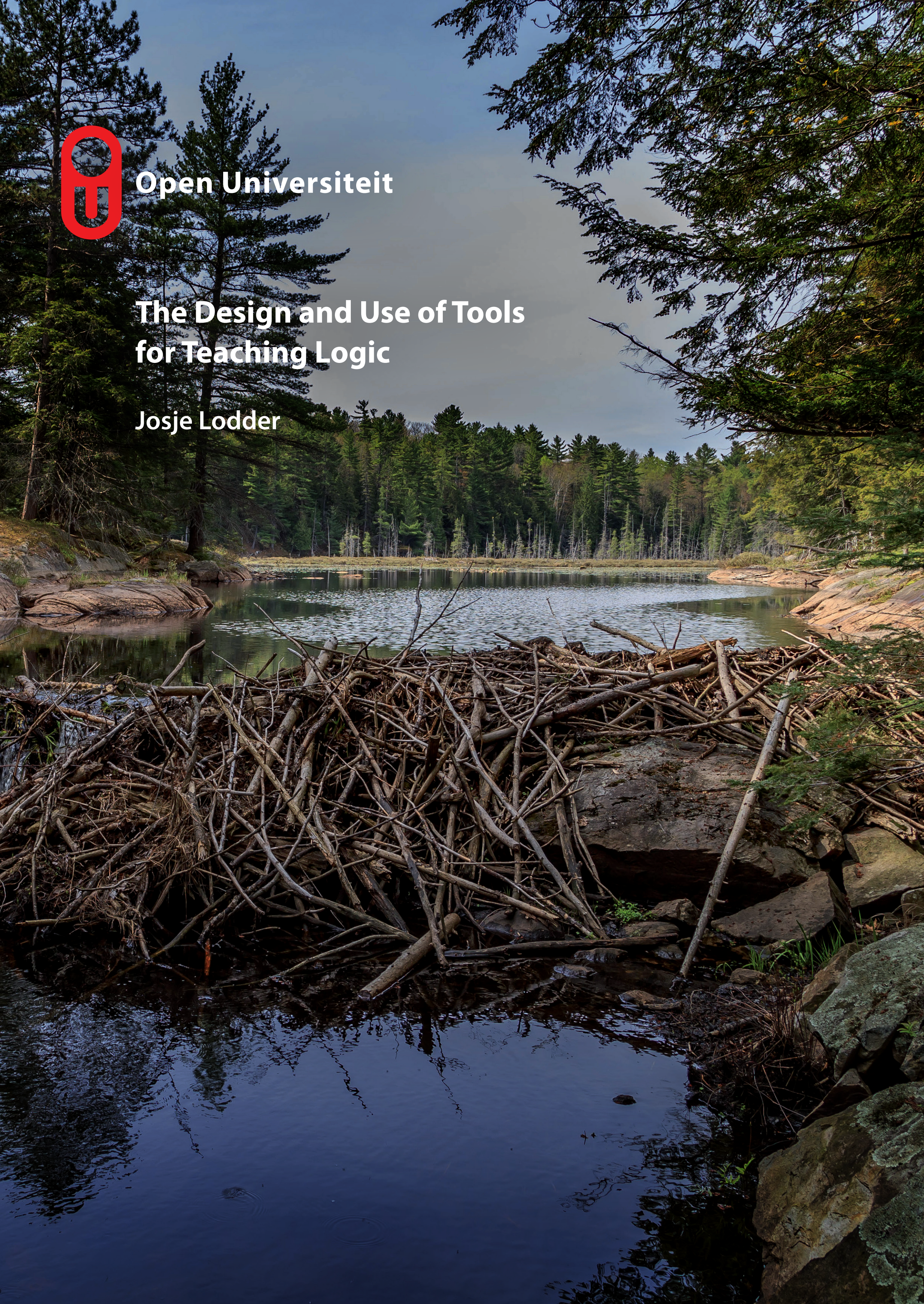
Take down policy
If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 09 Sep. 2021

Open Universiteit
www.ou.nl



Open Universiteit

The Design and Use of Tools for Teaching Logic

Josje Lodder

The Design and Use of Tools for Teaching Logic

Proefschrift

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. dr. Th. J. Bastiaens
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 11 september 2020 te Heerlen
om 13:30 uur precies

door

Jacoba Sophia Lodder

geboren op 10 september 1956 te Den Haag

Promotor

Prof. dr. J. T. (Johan) Jeuring	Open Universiteit
	Universiteit Utrecht

Co-promotor

Dr. B. J. (Bastiaan) Heeren	Open Universiteit

Leden van de beoordelingscommissie

Prof. dr. M. G. (María) Manzano	Universidad de Salamanca
Prof. dr. L.C. (Rineke) Verbrugge	Rijksuniversiteit Groningen
Prof. dr. E. (Erik) Barendsen	Radboud Universiteit Nijmegen
	Open Universiteit
Em. prof. dr. G. (Bert) Zwaneveld	Open Universiteit
Dr. H.P. (Hans) van Ditmarsch	CNRS, Frankrijk

Cover design: Marco Smeets, Open Universiteit
Printed by Canon Business Service, Heerlen.

Inhoudsopgave

1 Introduction

1.1 Logic Tutoring

Consider the next argument:

> *Some employees of the Open University do not like computers.*
>
> *All staff members of the Computer Science department of the Open University are employees of the Open University.*
>
> *Hence some staff members of the Computer Science department of the Open University do not like computers.*

Students have difficulty recognizing that the kind of reasoning given above is incorrect (Øhrstrøm et al., 2013). Logic courses teach students how to formalize such arguments, and either to prove that an argument is correct, or show that it is incorrect by giving a counter example.

Students learn logic in programs such as mathematics, philosophy, computer science, law, etc. For example, the ACM IEEE Computer Science Curricula 2013[1] mentions several topics in logic in its Core.

A typical course in logic contains amongst others the following topics (Burris, 1998; Huth and Ryan, 2004; Goldrei, 2005):

- syntax and semantics of propositional logic (truth tables)
- syntax of predicate language
- 'translation' of propositional and predicate formulae into natural language and vice versa
- a formal notion of semantics of predicate logic
- logical consequences in propositional and predicate logic
- standard equivalences and normal forms (disjunctive and conjunctive normal forms, prenex forms)
- one or more proof systems for propositional and predicate logic (natural deduction, Hilbert-style axiomatic proofs)

[1]`http://www.acm.org/education/CS2013-final-report.pdf`

- metatheorems (completeness etc.)
- induction

Similar topics can be found in online logic courses, such as the Stanford Introduction to Logic.[2] Depending on the target group other topics such as resolution, Hoare calculus, modal logic etc. are included.

Two essential factors underpinning successful learning are ‘learning by doing’ and ‘learning through feedback’ (Race, 2005). Students learning logic practice by solving exercises about the above mentioned topics. The nature of these exercises is rather diverse. A solution to an exercise may consist of a single step, for example the translation of a natural language sentence in logic, but most of the exercises ask for a derivation or a proof. Some exercises have a unique correct answer (for example the truth value of a formula, given a valuation), but most exercises have more than one correct answer or solution.

Textbooks for logic (Benthem et al., 2003; Hurley, 2008; Vrie and Lodder et al., 2009; Burris, 1998; Kelly, 1997) sometimes describe how exercises are solved, and give examples of good solutions. Because there are often many good solutions, it is infeasible to give all of them in a textbook, or to provide them online. A student who has solved an exercise in a way that is different from the solution in the textbook cannot check her solution for correctness by comparing it with the textbook solution. Hence, students need other sources of feedback when working on exercises in logic. Many universities organise exercise classes or office hours to help students with their work. However, it is not always possible to have a human tutor available. A student may be working at home, studying at a distance teaching university, or the teaching budget may be too limited to appoint enough tutors. For multi-step exercises, access to an intelligent tutoring system (ITS) (VanLehn, 2006) that provides feedback at step level might be of help. An ITS provides several services to students and teachers. Important services of ITSs are selecting tasks to be solved by the student, determining the level and progress of the student, diagnosing student actions, and giving feedback and hints to students. An ITS that follows the steps of a student when solving a task can be almost as effective as a human tutor (VanLehn, 2011).

As far as we know the first tutoring system for logic was developed in 1963 by Suppes (1971). His system supports the construction of natural deduction style proofs. A student can enter the name of a rule and the lines on which this rule should be applied. If the rule is applicable, the system performs this step automatically, otherwise the student receives an error message containing information about the mistake, for example that modus ponens is not applicable since the line that should contain an implication contains a conjunction instead.

[2] `http://intrologic.stanford.edu/public/index.php`

Since this first system, many systems have been developed and quite a lot have been abandoned. In 1993, Goldson evaluated eight tutoring systems available at that moment (Goldson et al., 1993), based on three criteria: what languages and logics are supported, is the system easy to use and is it useful for teaching purposes. Van Ditmarsch collected tutoring systems for different types of logics[3] and compared the interface of various tools for natural deduction (Van Ditmarsch, 1998). Since the year 2000, the conference Tools for Teaching Logic offers a platform for research in logic education. At the third conference, Huertas presented a comparative study of 26 different tools (Huertas, 2011). She discussed functional characteristics (basic functionalities and logical content), interaction characteristics (interactivity, feedback and help) and assessment characteristics. These overviews show that the distribution of the tools among the various topics is quite uneven, and the amount of support provided by the tools is very diverse. For example, there are some tools on natural deduction with extensive feedback services, but the few tools on axiomatic proofs or structural induction provide hardly any feedback, and they cannot help a student who does not know how to proceed. In the related work sections in the next chapters we will further discuss other tools for teaching logic.

1.2 Feedback and feed forward

This section introduces the terms feedback and feed forward, discusses how we use them, and contains some pointers to further literature.

Teaching and learning are as old as mankind. According to Morrison and Miller (2017), language plays an essential role in human teaching and learning, and they conjecture that the need to transmit cultural knowledge and skills might have influenced the evolution of language. The added value of using language in education is confirmed by an experiment set up by Morgan et al. (2015), in which students learn how to produce stone tools. The students were divided into groups that received different teaching interventions. In just one of the groups, the teacher was allowed to use language. When looking at the results, this group clearly outperformed the other groups. The use of language in teaching can have different functions such as instruction, explanation, but also feedback. According to Castro and Toro (2004), the capacity to provide feedback was a key factor in cultural evolution, since approval and disapproval make learning much more efficient than learning based on pure imitation.

The research on feedback in education is vast, and the field is still very active. Several authors performed reviews for different purposes. For example, Natriello (1987) developed a conceptual framework for integrating research on evaluation processes in schools and classrooms, Jaehnig and Miller (2007) identified and analysed studies on the effect of different types of feedback, Crooks (1988) studied the results of different evaluation practices on student results, Black and Wiliam (1998)

[3] `http://www.ucalgary.ca/aslcle/logic-courseware`

continued the work of Natriello and Crooks for the period 1988–1998, and Shute (2008) formulated guidelines for feedback.

Different authors use the term 'feedback' in different ways. For example, Boud and Molloy (2013) define feedback by

> *"Feedback is a process whereby learners obtain information about their work in order to appreciate the similarities and differences between the appropriate standards for any given work, and the qualities of the work itself, in order to generate improved work."*

This definition puts the learner in the centre. Evans (2013) distinguishes several aspects in definitions of the term 'feedback', for example product versus process, the function of feedback, and the approach such as constructivist or cognitive. Although we recognize that the role of the student in feedback is essential, in this thesis we will mainly use the term feedback for the product, by which we mean the comments provided by the ITS on the answers of the student. Narciss (2008) gives a classification of different types of feedback, which we will use in Chapter 2 in a review of tools for teaching the rewriting of logical formulae.

Where different definitions of feedback do have a common core, authors use the term 'feed forward' in at least two different meanings. For example, Rodríguez-Gómez and Ibarra-Sáiz (2015) define feed forward as

> *"strategies and comments that provide information about the results of assessment in a way that enables students to take a proactive approach to making progress."*

In their definition feed forward is provided after the completion of a task, and it is meant to be used in a next task. Other authors such as Koedinger and Aleven (2007); Nakevska et al. (2014); Herding (2013) use the term 'feed forward' to denote information that hints or tells the student what to do next. In this thesis we will use this second meaning; we use the term feed forward for hints and next steps provided by an ITS. Effectiveness of feed forward may depend on factors such as timing, content, level and presentation (Herding, 2013; Goldin and Carlson, 2013; Goldin et al., 2012; Perrenet and Groen, 1993). An ITS can provide feed forward without being asked, but often the initiative to request feedback lies with the student. In that case, hint abuse or underuse of feed forward may be a problem (Aleven et al., 2004).

1.3 Research questions

In this thesis we are interested in the design of ITSs for logic that support multiple-step exercises with different possible solutions. We will look at the following topics:

- standard equivalences and normal forms (disjunctive and conjunctive normal forms)
- Hilbert-style axiomatic proofs
- structural induction

In general, the rewriting of a propositional formula to normal form takes several steps, and both the rewriting and the final solution are not unique. Also, most axiomatic proofs consist of more than a single step, and the number of possible correct proofs is infinite, although in practice one only comes across a limited number of different solutions. Inductive proofs contain at least a base case and an inductive case in which the induction hypothesis has to be applied. Hence, this is also a multi-step exercise, with in general different possibilities (for example in the order of the steps) for completing a proof.

Topics such as syntax (writing correct formulae, producing a syntax tree etc.) and semantics (translations of natural language in logic and vice versa, finding models for predicate logic formulae etc.) are not part of this research. Some topics ask for activities that are completely mechanical and that lead to a unique answer, such as finding the truth value of a formula given a valuation. In general, tools to support this kind of exercises are already available.[4] Also, we do not investigate natural deduction and semantic tableaux since there are already several learning environments for these topics (Bornat, 2017; Sieg, 2007; Broda et al., 2006; Minica, 2015)[5], nor metatheorems, since in most courses students do not learn to prove such theorems by themselves. However, we will investigate structural induction, a basic proof technique for metatheorems.

The architecture of intelligent tutoring systems can be described by four components corresponding to domain expertise, pedagogical expertise, a student model and a user interface (Wenger, 1987). The domain expert module describes the domain knowledge necessary for solving a problem in the domain. A domain reasoner for logic contains the rules that may be used, and describes how the rules can be applied to construct a proof. A second task of this module is to check a student solution. The pedagogical module performs decisions about interventions and the sequencing of tasks. The student model contains information about the student knowledge, and the student communicates with the system via the interface. Not all ITSs contain all four components, and the boundaries between the components are not always sharp. Our interest is mainly in the domain expert module, which we denote by the domain reasoner, a term introduced by Goguadze (2010). To build an ITS we investigate how we can represent the knowledge about the subdomain of logic we want to model in a domain reasoner. A next question is how we can use this domain reasoner to provide feedback that points out common mistakes or

[4]see for example `https://www.cs.utexas.edu/~learnlogic/truthtables/` or `https://www.ixl.com/math/geometry/truth-tables`

[5]and online for example `https://creativeandcritical.net/prooftools`

misconceptions, and to help a student who gets stuck with a hint, a next step or an example solution. Whether students indeed learn by using an ITS for logic is a question that can only be answered by having students practice with the ITS. The feedback services of the Ideas framework (Heeren and Jeuring, 2014) serve as a basis for a learning environment for logic. These services have been developed to provide feedback and feed forward for exercises that can be solved stepwise. The services themselves are domain independent, and they can be applied to any domain with a domain reasoner that contains rules and strategies to solve exercises.

Summarizing, in this thesis we study the domains of standard equivalences and normal forms, Hilbert-style axiomatic proofs, and structural induction. The main questions we address are:

R1 How can we describe the expert knowledge of these topics in a domain reasoner?

R2 How can we generate feedback and feed forward?

R3 What is the effect of the use of the designed tools in logic education?

1.4 Content of this thesis

In the next subsections we will summarize the contents of the main chapters in this thesis.

Chapter 2: A domain reasoner for propositional logic

An important topic in courses in propositional logic is rewriting propositional formulae with standard equivalences. This chapter analyses what kind of feedback is offered by the various learning environments for rewriting propositional logic formulae, and discusses how we can provide these kinds of feedback in a learning environment. To give feedback and feed forward, we define solution strategies for several classes of exercises. We offer an extensive description of the knowledge necessary to support solving this kind of propositional logic exercises in a learning environment and introduce our implementation LogEx, an ITS for rewriting formulas in normal form and proving equivalences. Normal form rewritings and equivalence proofs may differ in the direction in which the rewritings are performed. Where a rewriting in normal form starts with the formula that has to be rewritten, an equivalence proof can be performed in two directions, starting with the left-hand side or the right-hand side formula. Also switching direction during the proof is possible. We describe our solution to the problem how to provide feedback and feed forward when a student changes the direction of the proof. Textbooks give standard strategies for rewriting formulas in normal form and equivalence proofs can use these. However, it is often possible to find shorter and more elegant solutions using heuristics. In this chapter we describe some of the implemented heurisics.

The origin of this chapter is:

Lodder, J., Heeren, B., and Jeuring, J. (2016). A domain reasoner for propositional logic. *Journal of Universal Computer Science*, 22(8):1097–1122

Chapter 3: A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae

This chapter describes an experiment with LogEx, an e-learning environment that supports students in learning how to prove the equivalence between two logical formulae, using standard equivalences such as DeMorgan. In the experiment, we compare two groups of students. The first group uses the complete learning environment, including hints, next steps, worked solutions and informative timely feedback. The second group uses a version of the environment without hints or next steps, but with worked solutions, and delayed flag feedback. We use pre and post tests to measure the performance of both groups with respect to error rate and completion of the exercises. We analyze the loggings of the student activities in the learning environment to compare its use by the different groups. Both groups score significantly better on the post test than on the pre test. We did not find significant differences between the groups in the post test, although the group using the full learning environment performed slightly better than the other group. In the examination, which took place five weeks after the experiment, the group of students who used the complete learning environment scored significantly better than a group of students who did not participate in the experiment, even when correcting for different skills in discrete mathematics.

This origin of this chapter is:

Lodder, J., Heeren, B., and Jeuring, J. (2019). A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae. *Journal of Computer Assisted Learning*, 35(5):620–632

Chapter 4: Generation and use of hints and feedback in a Hilbert-style axiomatic proof tutor

This chapter describes LogAx, an interactive tutoring tool that gives hints and feedback to a student who stepwise constructs a Hilbert-style axiomatic proof in propositional logic. LogAx generates proofs to calculate hints and feedback. We use an adaptation of an existing algorithm for natural deduction proofs to generate axiomatic proofs. We compare these generated proofs with expert proofs and student solutions, and conclude that the quality of the generated proofs is comparable to that of expert proofs. LogAx recognizes most steps that students take when constructing a proof. Even if a student diverges from the generated solution, LogAx still provides hints, including next steps or reachable subgoals, and feedback. With a few improvements in the design of the set of buggy rules, LogAx will cover about 80% of the mistakes made by students by buggy rules. The hints help students to complete the exercises.

This chapter is an extended version of:

Lodder, J., Heeren, B., and Jeuring, J. (2017). Generating Hints and Feedback for Hilbert-style Axiomatic Proofs. In Caspersen, M. E., Edwards, S. H., Barnes, T., and Garcia, D. D., editors, *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, March 8-11, 2017*, pages 387–392. ACM

The extension of this paper is partially based on the research from Wendy Neijenhuis for her MSc thesis on 'Using lemmas in an intelligent tutoring system for axiomatic derivation'.

Chapter 5: Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction

Structural induction is a proof technique that is widely used to prove statements about discrete structures. Students find it hard to construct inductive proofs, and when learning to construct such proofs, receiving feedback is important. In this chapter we discuss the design of a tutoring system, LogInd, that helps students with constructing stepwise inductive proofs by providing hints, next steps and feedback. As far as we know, this is the first tutoring system for structural induction with this functionality. We explain how we use a strategy to construct proofs for a restricted class of problems. This strategy can also be used to complete partial student solutions, and hence to provide hints or next steps. We use constraints to provide feedback. A pilot evaluation with a small group of students shows that LogInd indeed can give hints and next steps in almost all cases.

The origin of this chapter is:

Lodder, J., Heeren, B., and Jeuring, J. (2020). Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction. *Electronic Proceedings in Theoretical Computer Science*, 313:17–34

Chapter 6: Epilogue

This last chapter offers some conclusions and directions for future work.

Contribution of the candidate and co-authors in these chapters:

The candidate designed the research, performed the experiments, analysed the results, and wrote the papers, Bastiaan Heeren helped in implementing the software, and both Bastiaan Heeren and Johan Jeuring contributed to the discussions about the research, experiments, and results, and helped writing the papers.

2 A Domain Reasoner for Propositional Logic

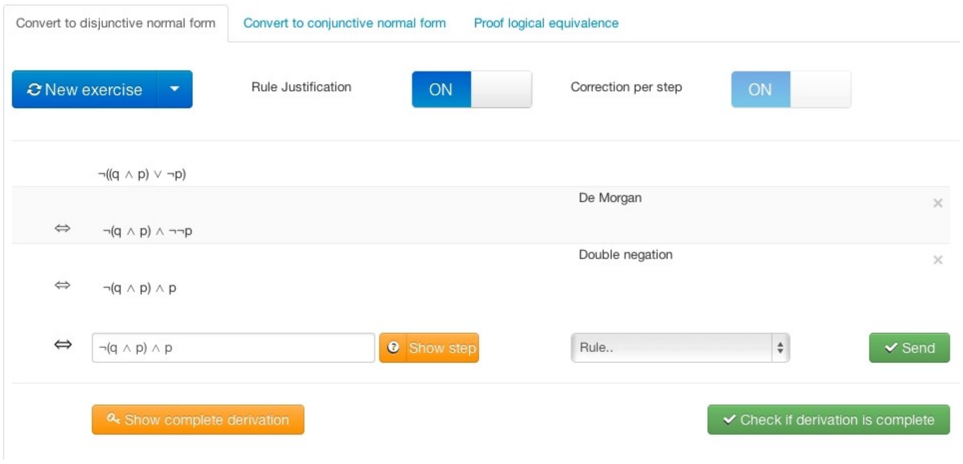
2.1 Introduction

Students learn propositional logic in programs such as mathematics, philosophy, computer science, law, etc. Students learning propositional logic practice by solving exercises about rewriting propositional formulae. Most textbooks for propositional logic (Benthem et al., 2003; Hurley, 2008; Vrie and Lodder et al., 2009; Burris, 1998; Kelly, 1997) contain these kinds of exercises. Such an exercise is typically solved in multiple steps, and may be solved in various correct ways. Textbooks sometimes describe how such exercises are solved, and give examples of good solutions. Because there often are many good solutions, it is infeasible to give all of them in a textbook, or provide them online.

How do students receive feedback when working on exercises in propositional logic? Many universities organise exercise classes or office hours to help students with their work. However, it is not always possible to have a human tutor available. In these cases, access to an intelligent tutoring system (ITS) (VanLehn, 2006) might be of help.

Feedback is an important aspect of an ITS. Usually an ITS offers various kinds of feedback: a diagnosis of a student step, a hint for the next step to take, in various levels of detail, or a completely worked-out solution. A diagnosis of a student step may analyse the syntax of the expression entered by the student, whether or not the step brings a student closer to a solution, or whether or not the step follows a preferred solution strategy, etc. An ITS that follows the steps of a student when solving a task can be almost as effective as a human tutor (VanLehn, 2011).

What kind of feedback do ITSs for propositional logic give? There are many tutoring systems for logic available (Huertas, 2011). In this paper we look at systems that deal with standard equivalences, in which a student has to learn to rewrite formulae, either to a normal form or to prove an equivalence. We analyse what kind of feedback is offered by the various learning environments for rewriting propositional logic formulae, and what kind of feedback is missing, and we discuss how we can provide these kinds of feedback in a learning environment. To give feedback we define solution strategies (procedures describing how basic steps may be combined to find a solution) for several classes of exercises, and we discuss the role of our strategy language in defining these solution strategies.


Convert to disjunctive normal form
Convert to conjunctive normal form
Proof logical equivalence
New exercise
Show step
Rule..
Send
Show complete derivation
Check if derivation is complete

Rule Justification ON

Correction per step ON

¬((q ∧ p) ∨ ¬p)

⇔ ¬(q ∧ p) ∧ ¬¬p — De Morgan

⇔ ¬(q ∧ p) ∧ p — Double negation

⇔ ¬(q ∧ p) ∧ p

Figuur 2.1: Screenshot of our learning environment for logic

Some interesting aspects of solving exercises in propositional logic are:

- Exercises such as proving equivalences can be solved from left to right (or top to bottom), or vice versa. How do we support solving exercises in which a student can take steps at different positions?
- Proving the equivalence of two formulae requires heuristics. These heuristics support effective reasoning and the flexible application of solution strategies in these proofs. How do we formulate heuristics in our solution strategies for solving these kinds of exercises? How 'good' are our solutions compared to expert solutions?
- Reuse and adaptivity play an important role in this domain: different teachers allow different rules, rewriting to normal form is reused, in combination with heuristics, in proving equivalences, etc. How can we support reusing and adapting solution strategies for logic exercises?

This paper describes the knowledge necessary to support solving propositional logic exercises in a learning environment, including solutions to the above aspects of solving propositional logic exercises.

Most existing systems for propositional logic do not have a student model; this paper calls such systems learning environments (LE). This paper focusses on the components necessary for providing feedback and feed forward in a propositional logic LE. However, we have also developed an LE on top of these components (Lodder et al., 2006), see Figure 2.1.[1]

This paper is organised as follows. Section 2.2 gives an example of an interaction of a student with a (hypothetical) LE. Section 2.3 describes the characteristics of

[1] `http://ideas.cs.uu.nl/logex/`

LEs for propositional logic, which Section 2.4 uses to compare existing LEs. We identify a number of aspects that have not been solved satisfactorily, and describe our approach to tutoring propositional logic in Section 2.5. Until Section 2.5 we describe a theoretical framework and look at related work. From section 2.5 on we present our own approach to tutoring propositional logic using so-called feedback services, and the implementation of our approach in the LogEx environment. Section 2.6 shows how this approach supports solving logic exercises for rewriting logic expressions to disjunctive or conjunctive normal form and for proving the equivalence of two logical formulae. We conclude with briefly describing the results of several small experiments we performed with LogEx.

2.2 Example interactions in an LE for propositional logic

This section gives some examples of interactions of a student with a logic tutor with advanced feedback facilities. Suppose a student has to solve the exercise of rewriting the formula

$$\neg((q \to p) \wedge p) \wedge q$$

into disjunctive normal form (DNF). The student might go through the following steps:

$$(\neg(q \to p) \wedge \neg p \wedge q \tag{2.1}$$

If a student submits this expression the LE reports that a parenthesis is missing in this formula. After correction the formula becomes:

$$(\neg(q \to p) \wedge \neg p) \wedge q \tag{2.2}$$

The LE reports that this formula is equivalent to the previous formula, but it cannot determine which rule has been applied: either the student performs multiple steps, or applies an incorrect step. In this case the student has very likely made a mistake in applying the DeMorgan rule. Correcting this, the student submits:

$$(\neg(q \to p) \vee \neg p) \wedge q$$

Now the LE recognises the rule applied (DeMorgan), and adds the formula to the derivation. Suppose the student does not know how to proceed here, and asks for a hint. The LE responds with: use Implication elimination. The student asks the LE to perform this step, which results in:

$$\neg((\neg q \vee p) \vee \neg p) \wedge q$$

The student continues with:

$$(\neg\neg q \vee \neg p \vee \neg p) \wedge q \tag{2.3}$$

The LE reports that this step is not correct, and mentions that when applying DeMorgan's rule, a disjunction is transformed into a conjunction. Note that in the second step of this hypothetical interactive session, the student made the same mistake, but since the formulae were accidentally semantically the same, the LE did not search for common mistakes there. The student corrects the mistake:

$$((\neg\neg q \wedge \neg p) \vee \neg p) \wedge q$$

and the LE appends this step to the derivation, together with the name of the rule applied (DeMorgan). The next step of the student,

$$\neg p \wedge q$$

is also appended to the derivation, together with the name of the rule applied (Absorption). At this point, the student may recognise that the formula is in DNF, and ask the LE to check whether or not the exercise is completed.

As a second example we look at an exercise in which a student has to prove that two formulae are equivalent:

$$(\neg q \wedge p) \rightarrow p \Leftrightarrow (\neg q \leftrightarrow q) \rightarrow p$$

The LE places the right-hand side formula below the left-hand side formula, and the student has to fill in the steps in between. It is possible to enter steps top-down or bottom-up, or to mix the two directions. The student chooses to enter a bottom-up step and to rewrite

$$(\neg q \leftrightarrow q) \rightarrow p$$

into:

$$\neg(\neg q \leftrightarrow q) \vee p$$

If she does not know how to proceed, she can ask for a hint. The LE suggests to rewrite this last formula; a first hint for these kinds of exercises will always refer to the direction of the proof. Now she can choose to perform this rewriting or she can ask for a second hint. This hint will suggest to use equivalence elimination.

She can continue to finish the exercise, but she can also ask the LE to provide a complete solution.

2.3 Characteristics of tutoring systems

This section introduces a number of characteristics of tutoring systems, which we will use for the comparison of existing LEs for logic in Section 2.4. This is not a complete description of the characteristics of LEs, but large enough to cover the most important components, such as the inner and outer loop of tutoring systems (VanLehn, 2006), and to compare existing tools. The outer loop of an ITS presents different tasks to a student, in some order, depending on a student model, or by letting a student select a task. The inner loop of an ITS monitors the interactions between a student and a system when a student is solving a particular task. Important aspects of the inner loop are the analyses performed and the feedback provided. We distinguish feedback consisting of reactions of the system on steps performed by the students, and hints, next steps and complete solutions provided by the system. Although Narciss (and others) call this last category also feedback, others use the term feed forward (Hattie and Timperley, 2007), which we also will use in this paper. For the interactions in the inner loop, some aspects are specific for LEs for logic.

2.3.1 Tasks

The starting point of any LE is the tasks it offers. The kind of tasks we consider in this paper are calculating normal forms (NF; in the text we introduce the abbreviations used in the overview in Figure 2.2) and proving an equivalence (EQ).

An LE may contain a fixed set of exercises (FI), but it may also randomly generate exercises (RA). Some LEs offer the possibility to enter user-defined exercises (US).

2.3.2 Interactions in the inner loop

In the inner loop of an LE, a student works on a particular task. In most LEs for rewriting logical formulae a student can submit intermediate steps. Some systems allow a student to rewrite a formula without providing the name of a rewrite rule (FO), in other systems she chooses a rule and the system rewrites the formula using that rule (RU). Some LEs require a student to provide both the name of the rule to apply, and the result of rewriting with that rule (RaF).

The interactions in the inner loop are facilitated by the user-interface. A user interface for an LE for logic needs to satisfy all kinds of requirements; too many to list in this paper. For our comparison, we only look at offering a student the possibility to work in two directions when constructing a proof (2D).

2.3.3 Feedback

How does an LE give feedback on a step of a student? To distinguish the various types of feedback, we give a list of possible mistakes. The numbers refer to examples of these mistakes in Section 2.2.

- A syntactical mistake (2.1)
- A mistake in applying a rule. We distinguish two ways to solve an exercise in an LE depending on whether or not a student has to select the rule she wants to apply. If she indicates the rule she wants to apply, she can make the following mistakes: perform an incorrect step by applying the rule incorrectly or perform a correct step that does not correspond to the indicated rule. If a student does not select the rule she wants to apply, the categories of possible mistakes are somewhat different. A student can rewrite a formula into a semantically equivalent formula, but the LE has no rule that results in this formula. This might be caused by the student applying two or more rules in a single step, but also by applying an erroneous rule, which accidentally leads to an equivalent formula (2.2). A second possibility is the rewriting of a formula into a semantically different formula (2.3).
- A strategic mistake. A student may submit a syntactically and semantically correct formula, but this step does not bring her closer to a solution. We call this a strategic mistake.

We distinguish three categories of mistakes: syntactic errors, errors in applying a rule, and strategic errors. Narciss characterises classes of feedback depending on how information is presented to a student (Narciss, 2008). When a student has made an error, we can provide the following kinds of feedback: Knowledge of result/response (KR, correct or incorrect), knowledge of the correct results (KCR, description of the correct response), knowledge about mistakes (KM, location of mistakes and explanations about the errors), and knowledge about how to proceed (KH).

2.3.4 Feed forward

To help a student with making progress when solving a task, LEs use feed forward: they may give a hint about which next step to take (HI, in various levels of detail), they may give the next step explicitly (NE), or they may give a general description of the components that can be used to solve an exercise (GE). If steps can be taken both bottom-up and top-down, is feed forward also given in both directions (FF2), or just in one of the two directions (FF1)?

2.3.5 Solutions

Some LEs offer worked-out examples (WO), or solutions to all exercises available in the tool (SOL).

2.3.6 Adaptability

Finally, we look at flexibility and adaptability. Can a teacher or a student change the set of rules or connectives (YES, NO)?

2.4 A comparison of tools for teaching logic

This section describes some LEs for logic using the characteristics from the previous section. We build upon a previous overview of tools for teaching logic by Huertas (2011). Some of the tools described by Huertas no longer exist, and other, new tools have been developed. We do not give a complete overview of the tools that currently exist, but restrict ourselves to tools that support one or more of the exercise types of LogEx: rewriting a formula in normal form and proving an equivalence using standard equivalences as rewrite rules. Quite a few tools for logic support learning natural deduction, which is out of scope for our comparison. We summarise our findings in Figure 2.2.

2.4.1 Rewriting a formula in normal form

Using Organon[2] Dostálová and Lang (2011, 2007), a student practices rewriting propositional formulae into DNF or CNF (conjunctive normal form). It automatically generates exercises, based on a set of schemas. A student cannot indicate a rule she wants to apply when taking a step. If a rewritten formula is semantically equivalent Organon accepts it, even if the student probably made a mistake, as in (2.2). When a student enters a syntactically erroneous or non-equivalent formula, Organon gives a KR error message. In training mode, a student can ask for a next step. The steps performed by Organon are at a rather high level: it removes several implications in a single step, or combines DeMorgan with double negation. A student can ask for a demo, in which case Organon constructs a DNF stepwise.

FMA contains exercises on rewriting propositional formulae to complete normal form: a DNF or CNF where each conjunct respectively disjunct contains all the occurring variables, possibly negated (Prank, 2014). A student highlights the subformula she wants to change. In input mode, she enters the changed subformula. The tool checks the syntax, and provides syntax error messages if necessary. In rule mode, a student chooses a rule, and FMA applies this rule to a subformula, or it gives an error message if it cannot apply it. In 2013, an analyser was added to FMA. The analyser analyses a complete solution, and provides error messages on steps where a student solution diverges from a solution obtained from a predefined strategy. For example, the analyser might give the feedback: "Distributivity used too early".

[2] `http://organon.kfi.zcu.cz/organon/`

	outer loop		interactions		feedback			feed forward		
tool	type	exercises	input	direction	syntax	rule	str	hint	solution	adapt.
Organon	NF	RA	FO	n.a.	KR	KR	-	NE	WO	NO
FMA	NF	RA, FI	FO, RU	n.a	KR	KR	KCR	-	-	NO
Logicweb	NF*	FI, US	RU	n.a.	KR	n.a.	KR	-	-	NO
SetSails	EQ	FI, US	RaF	2D	KCR	KCR	-	GE***	-	YES
Logic Cafe	EQ, CO	FI, US	RaF	2D	KR	KCR	-	GE, FF1	WO	NO
FOL equivalence	EQ**	FI	RaF	?	?	KM?	-	?	-	NO

Figuur 2.2: Comparison of logic tools and their characteristics

Type	NF: normal form; EQ: equivalence proofs; *: normal forms as part of a resolution proof; **: equivalence proof in first order logic
Exercises	US: user defined exercises; RA: randomly generalised exercises; FI: fixed set
Input	FO: input a formula; RU: input a rule name; RaF: input a rule name and a formula
Direction	2D: student can work forwards and backwards; n.a.: not applicable, because tool does not offer these exercises
Syntax	n.a.: not applicable; KR: correct/incorrect; KCR: correction of (some) syntax errors
Rule	n.a.: not applicable; KR: correct/incorrect; KCR: explanation i.e. a rule-example
Str	n.a.: not applicable; KR: step does/does not follow a desired strategy; KCR: explanation why a step does not follow a desired strategy
Hint	NE: LE provides next step; GE: list of possible useful rules, subgoal, etc.; ***: not always available; FF1: feed forward only in one direction (top down)
Solution	WO: worked out demos
Adapt.	YES: users may adapt the rule set; NO: users cannot adapt the rule set

Logicweb[3] is a tool for practicing resolution (and semantic trees), and strictly spoken not a tool for rewriting a formula into normal form. However, to solve an exercise, a student starts with rewriting the given formulae in clausal form (conjunctive normal form), using standard equivalences. Logicweb is an example of a tool where rewriting is performed automatically. At each step, the student selects a formula and the tool offers a list of (some of the) applicable rules. The student selects a rule, and the tool applies it. Thus a student can focus on the strategy to solve an exercise. The only mistake a student can make is choosing a rule that does not bring the student closer to a solution. Rules can only be applied in one direction, hence the only possible 'wrong' rule is distribution of and over or, since that rule can bring a student further from a clausal form. If a student chooses to distribute and over or, the tool can tell the student that this is not the correct rule to apply at this point in the exercise. The tool contains a fixed set of exercises, but user-defined exercises are also possible. In the latter case the tool reports syntactic errors.

2.4.2 Proving equivalences

SetSails[4] (Zimmermann and Herding, 2010; Herding et al., 2010) offers two kinds of exercises: prove that two set-algebra expressions denote the same set, or prove that two propositional logic formulae are equivalent. We only look at the last kind of exercises. SetSails contains a (small) set of predefined exercises, but a user can also enter an exercise.

SetSails provides immediate feedback on the syntax of a formula and automatically adds parentheses if a formula is ambiguous. In each step a student chooses a rule, and the system suggests possible applications of this rule, from which the student picks one. However, some of these alternatives are deliberately wrong: in some cases another rule is applied, or the suggested formula contains a common mistake. Choosing an alternative is thus a kind of multiple choice exercise. A student can also enter a formula, in case it is missing in the list of suggested formulae. Further feedback, such as corrections on the applied rules and hints, is given when a student asks the system to check a proof, which can be done at each step. The system recognises if a new formula is equivalent to the old one, but cannot be obtained by rewriting with a particular rule, and also recognises when the rule name does not correspond to the rule used. Although the alternative rewritings offered by the LE seem to be generated by some buggy rules, these are not mentioned when a student chooses a wrong alternative. The hints mention the rules possibly needed, but not how to apply them, and the list of the rules needed is not complete. The system does not provide next steps or complete solutions. After entering an exercise, a user chooses rules from a predefined set or adds new rules that can be used in a

[3]`http://ima.udg.edu/~humet/logicweb`
[4]`http://sail-m.de/`

derivation. This makes it possible to adapt the rule set, or to use previous proofs as lemmas in new proofs. However, the tool does not guarantee that an exercise can be solved with the set of rules provided by a user. A user might have forgotten to include some essential rules from the rule set. A student can work both forwards and backwards, but the tool does not give advice about these directions.

Logic Cafe[5] contains exercises covering most of the material of an introductory logic course. The part on natural deduction contains some exercises in which a student has to rewrite a formula by using standard equivalences. If a student makes a mistake in a step, it is not accepted. In some cases Logic Cafe gives global feedback about a reason, for example that a justification should start with the number of the line on which the rule is applied, or that a justification contains an incorrect rule name. When a student asks for a hint, she gets a list of rules she has to apply. This kind of feed forward is only available for predefined exercises. A student can enter her own exercise. The LE contains some small animations that illustrate the construction of a proof, and some example derivations in which the LE tells a student exactly what to do at each step.

In the FOL equivalence system (Grivokostopoulou et al., 2013), a student practices with proving the equivalence between formulae in first order logic. We describe the tool here because it uses a standard set of rewriting rules for the propositional part of the proof. A student selects an exercise from a predefined set of exercises. To enter a step she first selects a rule, and then enters the formula obtained by applying the rule. The system checks this step, and gives a series of messages in case of a mistake. The first message signals a mistake. Successive messages are more specific and give information about the mistake and how to correct it. As far as we could determine, the system does not give a hint or a next step if a student does not select a rule, and does not provide complete solutions. It is not clear whether a student can work forwards, backwards, or both.

2.4.3 A comparison

We compare the above tools by means of the aspects described at the beginning of this section.

The kind and content of the feedback varies a lot, partly depending on the way a student works in the tool. Feedback on the rule-level consists of mentioning that a rule name is incorrect, or that a mistake has been made. SetSails gives a general form of the correct rule to be applied. FOL equivalence is the only tool that gives error-specific feedback. None of the other tools report common mistakes (KM). Logicweb gives feedback on the strategic level when a student uses a wrong distribution rule, and FMA indicates where a student solution diverges from a solution obtained from a predefined strategy.

[5]`http://thelogiccafe.net/PLI/`

Feed forward varies a lot between the different tools too. There is some correlation between the type of exercise and the kind of feed forward. For the 'easy' exercises (rewriting into normal form), tools do provide feed forward, such as complete solutions to exercises as given by Organon. For the other tools, feed forward is more restricted, and mainly consists of general observations about the rules you might need to solve the problem.

SetSails and Logic Cafe offer the possibility to prove equivalences while working in two directions. However, these tools do not offer hints on whether to perform a forward or a backward step, and it is not possible to receive a next backward step.

In SetSails a user can define her own set of rules. However, it does not adapt the feed forward to this user set.

In conclusion, there already are a number of useful LEs for propositional logic, but there remains a wish-list of features that are not, or only partially, supported in these LEs. The main feature missing in almost all tools is feed forward: only the LEs for practicing normal forms offer next steps or complete solutions in any situation. Tools on proving equivalences do not provide feed forward, or provide feed forward only in a limited number of situations. This might be caused by the fact that the decision procedures for solving these kinds of exercises are not very efficient or smart. A good LE provides feed forward not only for a standard way to solve an exercise, but also for alternative ways. It also supports a student that uses both forward and backward steps in her proof.

The feedback provided in LEs for propositional logic is also rather limited. A good LE should, for example, have the possibility to point out common mistakes (KM).

We hypothesise that the number of tools for propositional logic is relatively high because different teachers use different logical systems with different rule sets. An LE that is easily adaptable, with respect to notation, rule sets, and possibly strategies for solving exercises, might fulfil the needs of more teachers.

2.5 Feedback services

The architecture of an intelligent tutoring system (ITS) is described by means of four components (Nwana, 1990): the expert knowledge module, the student model module, the tutoring module, and the user interface module. The expert knowledge module is responsible for 'reasoning about the problem', i.e., for managing the domain knowledge and calculating feedback and feed forward. Typically, this component also includes a collection of exercises, and knowledge about the class of exercises that can be solved. Following Goguadze, we use the term *domain reasoner* for this component (Goguadze, 2011). We discuss how to construct a domain reasoner for propositional logic that has all characteristics introduced in Section 2.3.

A domain reasoner provides feedback services to an LE. We use a client-server style in which an LE uses stateless feedback services of the domain reasoner by sending JSON or XML requests over HTTP (Heeren and Jeuring, 2014). We identify three categories of feedback services: services for the outer loop, services for the inner loop, and services that provide meta-information about the domain reasoner or about a specific domain, such as the list of rules used in a domain. The feedback services are domain independent, and are used for many domains, including rewriting to DNF or CNF and proving two formulae equivalent.

2.5.1 Services for the outer loop

The feedback services supporting the outer loop are:

- give a list of predefined *examples* of a certain difficulty
- *generate* a new (random) exercise of a specified difficulty
- *create* a new user-defined exercise

The domain reasonar has to specify the difficulty of the exercise or example. We have defined a random formula generator that is used for the DNF and CNF exercises, but we do not generate random pairs for equivalence proofs (or consequences). Since in this paper we do not investigate the effect of the difficulty of an exercise, we use a rather pragmatic way to define this difficulty, namely by looking at the length of a solution and the possible complexity caused by occurrences of the equivalence connective.

2.5.2 Services for the inner loop

There are two fundamental feedback services for the inner loop. The *diagnose* service generates feedback. It analyses a student step and detects various types of mistakes, such as syntactical mistakes, common misconceptions, strategic errors, etc. The *allfirsts* service calculates feed forward, in the form of a list of all possible next steps based on a (possibly non-deterministic) strategy.

To provide feedback services for a class of exercises in a particular domain, we need to specify Heeren and Jeuring (2014):

- The *rules* (laws) for rewriting and common misconceptions (buggy rules). In Section 2.5.5 we present rules for propositional logic.
- A *rewrite strategy* that specifies how an exercise can be solved stepwise by applying rules. Section 2.6 defines strategies for the logic domain.
- Two relations on terms: semantic *equivalence* of logical propositions compares truth tables of formulae, whereas syntactic *similarity* compares the structure of two formulae modulo associativity of conjunction and disjunction. These relations are used for diagnosing intermediate solutions.
- Two predicates on terms. The predicate *suitable* identifies which terms can be solved by the strategy of the exercise class. The predicate *finished* checks if a term is in a solved form (accepted as a final solution): for instance, we

check that a proposition is in some normal form, or that an equivalence proof is completed.

Explicitly representing rules and rewrite strategies improves adaptability and reuse of these components. We come back to the issue of adaptability in Section 2.6.2.

2.5.3 Alternative approaches

There are different ways to specify feedback or feed forward for logic exercises. Defining feedback separately for every exercise is very laborious, especially since solutions are often not unique. In this approach it is hard to also provide feedback or hints when a student deviates from the intended solution paths. One way to overcome this is to use a database with example solutions (Aleven et al., 2009); an implementation of this idea for a logic tutor is described by Stamper. In this tutor, Deep Thought, complete solutions and intermediate steps are automatically derived using data mining techniques based on Markov decision processes. These solutions and steps are then hard coded. In this way, Deep Thought (Stamper et al., 2011b) can provide a hint in 80% of the cases. Another advantage of using example solutions over using solution strategies, is that it is not always clear how to define such a strategy.

The use of example solutions also has some disadvantages. In our experience with Deep Thought, if a solution diverges from a 'standard' solution, there are often no hints available. Furthermore, the system can only solve exercises that are similar to the exercises in the database.

2.5.4 The use of services in the LogEx learning environment

We have developed a domain reasoner for logic, which is used in the LogEx learning environment[6]. In this section we describe how LogEx deals with the characteristics given in Figure 2.2. LogEx presents exercises on rewriting a formula into normal form and on proving equivalences. We use all three kinds of exercise creation: users can enter their own exercises, LogEx generates random exercises for normal form exercises, and LogEx contains a fixed set of exercises for proving equivalence.

A student enters formulae. When proving equivalences a student also has to provide a rule name. In the exercises about rewriting to normal form this is optional. Equivalence exercises can be solved by taking a step bottom-up or top-down.

Most of the feedback on syntax is of the KR type: only if parentheses are missing LogEx gives KCR feedback. LogEx provides KM feedback on the level of rules. It not only notes that a mistake is made, but also points out common mistakes, and mentions mistakes in the use of a rule name. LogEx does not support strategic feedback. LogEx accepts any correct application of a rule, even if the step is not recognised by the corresponding strategy. In such a case the domain reasoner

[6] `http://ideas.cs.uu.nl/logex/`

CommOr:	$\phi \lor \psi \Leftrightarrow \psi \lor \phi$	ComplOr:	$\phi \lor \neg\phi \Leftrightarrow T$
CommAnd:	$\phi \land \psi \Leftrightarrow \psi \land \phi$	ComplAnd:	$\phi \land \neg\phi \Leftrightarrow F$
DistrOr:	$\phi \lor (\psi \land \chi) \Leftrightarrow (\phi \lor \psi) \land (\phi \lor \chi)$	DoubleNeg:	$\neg\neg\phi \Leftrightarrow \phi$
DistrAnd:	$\phi \land (\psi \lor \chi) \Leftrightarrow (\phi \land \psi) \lor (\phi \land \chi)$	NotTrue:	$\neg T \Leftrightarrow F$
		NotFalse:	$\neg F \Leftrightarrow T$
AbsorpOr:	$\phi \lor (\phi \land \psi) \Leftrightarrow \phi$		
AbsorpAnd:	$\phi \land (\phi \lor \psi) \Leftrightarrow \phi$	TrueOr:	$\phi \lor T \Leftrightarrow T$
IdempOr:	$\phi \lor \phi \Leftrightarrow \phi$	FalseOr:	$\phi \lor F \Leftrightarrow \phi$
IdempAnd:	$\phi \land \phi \Leftrightarrow \phi$	TrueAnd:	$\phi \land T \Leftrightarrow \phi$
		FalseAnd:	$\phi \land F \Leftrightarrow F$
DefEquiv:	$\phi \leftrightarrow \psi \Leftrightarrow (\phi \land \psi) \lor (\neg\phi \land \neg\psi)$		
DefImpl:	$\phi \rightarrow \psi \Leftrightarrow \neg\phi \lor \psi$		
DeMorganOr:	$\neg(\phi \lor \psi) \Leftrightarrow \neg\phi \land \neg\psi$		
DeMorganAnd:	$\neg(\phi \land \psi) \Leftrightarrow \neg\phi \lor \neg\psi$		

Figuur 2.3: Rules for propositional logic

restarts the strategy recogniser from the point the student has reached. Thus, LogEx can give hints even if a student diverges from the strategy.

LogEx gives feed forward in the form of hints on different levels: which formula has to be rewritten (in case of an equivalence proof), which rule should be applied, and a complete next step. Feed forward is given for both forward and backward proving, and even recommends a direction. LogEx also provides complete solutions.

LogEx does not offer the possibility to adapt the rule set. In Section 2.6.2 we will sketch an approach to supporting adaptation.

2.5.5 Rules

All LEs for propositional logic use a particular set of logical rules to prove that two formulae are equivalent, or to derive a normal form. There are small differences between the sets used. The rule set we use is taken from the discrete math course of the Open University of the Netherlands Vrie and Lodder et al. (2009) (Fig. 2.3). Variants can be found in other textbooks. For example, Burris defines equivalence in terms of implication Burris (1998), and Huth leaves out complement and true-false rules Huth and Ryan (2004).

Sometimes derivations get long when strictly adhering to a particular rule set. For this reason we implicitly allow associativity in our solution strategies, so that associativity does not need to be mentioned when it is applied together with another rule. This makes formulae easier to read, and reduces the possibility of syntax errors. Commutativity has to be applied explicitly; but we offer all commutative variants of the complement rules, the false and true rules, and absorption (Fig. 2.3). For example, rewriting $(q \land p) \lor p$ into p is accepted as an application of AbsorpOr.

Also a variant of the distribution rule is accepted: students may rewrite $(p \wedge q) \vee r$ in $(p \vee r) \wedge (q \vee r)$ using DistrOr, and the same holds for DistrAnd.

In our services we use generalised variants of the above rules. For example, generalised distribution distributes a subterm over a conjunct or disjunct of n different subterms, and we recognise a rewrite of $\neg(p \vee q \vee r \vee s)$ into $\neg p \wedge \neg(q \vee r) \wedge \neg s$ as an application of a generalised DeMorgan rule. These generalised rules are more or less implied by allowing implicit associativity.

Buggy rules describe common mistakes. An example of a buggy rule is given in the introduction of this paper, where a student makes a mistake in applying DeMorgan and rewrites $\neg(p \vee q) \vee (\neg\neg p \wedge \neg q) \vee \neg q$ into $(\neg p \vee \neg q) \vee (\neg\neg p \wedge \neg q) \vee \neg q$. This step is explained by the buggy rule $\neg(\phi \vee \psi) \not\Leftrightarrow \neg\phi \vee \neg\psi$; a common mistake in applying DeMorgan. In case of a mistake, our diagnose service tries to recognise if the step made by the student matches a buggy rule. The set of (almost 100) buggy rules we use is based on the experience of teachers, and includes rules obtained from analysing the log files of the diagnose service.

2.5.6 A strategy language

Although some textbooks give strict procedures for converting a formula into normal form (Huth and Ryan, 2004), most books only give a general description (Vrie and Lodder et al., 2009; Burris, 1998), such as: first remove equivalences and implications, then push negations inside the formula using DeMorgan and double negation, and finally distribute and over or (DNF), or or over and (CNF). In general, textbooks do not describe procedures for proving equivalences, and these procedures do not seem to belong to the learning goals. We hypothesise that the text books present these exercises and examples to make a student practice with the use of standard equivalences (Dalen, 2004; Ben-Ari, 2012). Since we want to provide both feedback and feed forward, we need solution strategies for our exercises.

We use *rewriting strategies* to describe procedures for solving exercises in propositional logic, to generate complete solutions and hints, and to give feedback. To describe these rewriting strategies we use the strategy language developed by Heeren et al. (Heeren et al., 2010). This language is used to describe strategies in a broad range of domains. The meaning of the word 'strategy' here slightly deviates from its usual meaning. A rewriting strategy is any combination of steps, which could be used to solve a procedural problem, but which can also be a more or less random combination of steps. We recapitulate the main components of this language, and extend it with a new operator. The logical rules (standard equivalences) that a student can apply when rewriting a formula in normal form or proving the equivalence of two formulae are described by means of rewriting rules. These rules are the basic steps of a rewriting strategy, and in the (inductive) definition of the language, they are considered rewriting strategies by themselves. We use combinators to combine two rewriting strategies, so a rewriting strategy is a logical rule r, or, if s and t are rewriting strategies then:

- $s <\!\star\!> t$ is the rewriting strategy that consists of s followed by t
- $s <\!|\!> t$ is the rewriting strategy that offers a choice between s and t
- $s >\!|\!> t$ is the rewriting strategy that offers a choice, but prefers s
- $s \triangleright t$ is a left-biased choice: t is only used if s is not applicable
- *repeat* s repeats the rewriting strategy s as long as it is applicable

We offer several choice operators. The preference operator is new, and has been added because we want to give hints about the preferred next step, but allow a student to take a step that is not the preferred step. For example, consider the formula $(p \lor s) \land (q \lor r) \land (u \lor v)$. To bring this formula into DNF we apply distribution. We can apply distribution top-down (to the first conjunct in $(p \lor s) \land ((q \lor r) \land (u \lor v)))$ or bottom-up (to the second conjunct). A diagnosis should accept both steps, but a hint should advise to apply distribution bottom-up, because this leads to a shorter derivation. We implement this using the preference operator.

2.6 Strategies for propositional logic exercises

This section gives rewriting strategies for rewriting a logic formula to normal form and for proving the equivalence of two logical formulae. Furthermore, we show how a rewriting strategy can be adapted in various ways.

2.6.1 A strategy for rewriting a formula to DNF

There are several strategies for rewriting a formula to DNF. A first strategy allows students to apply any rule from a given set of rules to a formula, until it is in DNF. Thus a student can take any step and find her own solution, but worked-out solutions produced by this strategy may be unnecessarily long, and the hints it provides will not be very useful. A second strategy requires a student to follow a completely mechanic procedure, such as: first remove implications and equivalences, then bring all negations in front of atomic formulae by applying the DeMorgan rules and removing double negations, and conclude with the distribution of conjunctions over disjunctions. This strategy teaches a student a method that always succeeds in solving an exercise, but it does not help to get strategic insight. This strategy also does not always produce a shortest solution. The problem of finding a shortest derivation transforming a formula into DNF is decidable, and we could define a third strategy that only accepts a shortest derivation of a formula in DNF. There are several disadvantages to this approach. First, it requires a separate solution strategy for every exercise. If a teacher can input an exercise, this implies that we need to dynamically generate, store, and use a strategy in the back-end. This might be computationally very expensive. Another disadvantage is that although such a strategy produces a shortest derivation, it might confuse a student, since the strategy might be too specialised for a particular case. For example, to rewrite a formula into DNF, it is in general a good idea to remove implications and apply DeMorgan before applying distribution. However, in the formula $\neg(q \vee (p \rightarrow q)) \wedge (p \rightarrow (p \rightarrow q))$ a derivation that postpones rewriting the implication $p \rightarrow q$ and starts with applying DeMorgan and distribution takes fewer steps than a derivation that starts with removing the three implications. If a strategy gives the hint to apply DeMorgan, a student might not understand why this hint is given. We think that a strategy does not always have to produce a shortest derivation. This implies that there might be situations in which a student can construct a solution with fewer steps than the strategy. As long as an LE accepts such a solution, this need not be a problem.

We choose to develop a variant of the second strategy: a strategy based on a mechanical procedure that allows a student to deviate from the procedure, and which uses heuristics to implement strategic insights. This strategy offers a student a basis to start from when she works on an exercise, but also stimulates finding strategically useful steps that lead to shorter derivations. These steps include steps that at first sight seem to complicate formulae, but offer possibilities for simplification later on (Schoenfeld, 1987). Our strategy, based on the strategy described in the Open University textbook (Vrie and Lodder et al., 2009), prescribes an order in which substrategies are applied. For example, simplifying a formula has highest priority while distributing and over or has lowest priority. However, when a formula is a disjunct, both disjuncts can be rewritten separately, in any order. For example, a student might solve the exercise $(\neg\neg p \wedge (q \vee r)) \vee (p \rightarrow \neg\neg q)$ by first rewriting the first disjunct, reducing it to DNF in two steps:

$$\begin{array}{ll}(\neg\neg p \wedge (q \vee r)) \vee (p \rightarrow \neg\neg q) & \Leftrightarrow \\ (p \wedge (q \vee r)) \vee (p \rightarrow \neg\neg q) & \Leftrightarrow \\ (p \wedge q) \vee (p \wedge r) \vee (p \rightarrow \neg\neg q) & \end{array}$$

If a strategy requires to remove double negations before applying distribution, this last step is not allowed, because the right-hand disjunct should be rewritten first. On the other hand, applying distribution before removing double negations leads to duplicating the subformula $\neg\neg p$. For such a situation we introduce the combinator *somewhereOr* s: check whether a formula is a disjunction and apply s to one of the disjuncts, otherwise apply s to the complete formula.

If a formula is a disjunction, we can rewrite it to DNF by rewriting the disjuncts separately. However, sometimes it is possible to apply a simplification rule on multiple disjuncts. For example,

$$(\neg\neg p \wedge (q \vee r)) \vee (p \rightarrow \neg\neg q) \vee \neg(p \rightarrow \neg\neg q)$$

is best rewritten by using the complement rule on the last two disjuncts. For such cases we introduce a set of disjunction simplification rules that we try to apply at top-level: FALSEOR, TRUEOR, IDEMPOR, ABSORPOR, and COMPLOR. The substrategy *orRulesS*, the definition of which is omitted, applies one of these rules, if possible.

When applying the *orRulesS* substrategy is no longer possible, the rewriting strategy for DNF continues with rewriting a formula into negation normal form (NNF). A formula in negation normal form does not contain implications or equivalences, and all negations occur in front of an atom. To obtain an NNF we introduce three substrategies:

– *simplifyStep*: simplify by applying *orRulesS*, or the dual strategy *andRulesS*, or one of the three rules DoubleNeg, NotFalse, or NotTrue
– *eliminateImplEquivS*: remove implications by applying DefImpl or equivalences by applying DefEquiv
– *deMorganS*: use DeMorganOr or DeMorganAnd to move negations down

The rewriting strategy *nnfStep* combines these three substrategies:

$$\textit{nnfStep} = \textit{simplifystep} \triangleright (\textit{eliminateImplEquivS} >\!|\!> \textit{deMorganS})$$

The *nnfStep* strategy performs at most one step. We use the *repeat* combinator to perform all steps necessary to obtain an NNF. The resulting rewriting strategy always tries to simplify a formula first. After simplifying it removes implications or equivalences, simplifying in between, and then applies DeMorgan.

After obtaining an NNF we distribute conjunctions over disjunctions to obtain a DNF. This is achieved by the rewriting strategy *distrAndS*, which applies DistrAnd or GeneralDistrAnd, preferring the second rule.

We now have all the ingredients of a rewriting strategy for rewriting to DNF:

$$\begin{aligned}&\textit{dnfStrategy} = \textit{repeat}\\&\quad(\textit{orRulesS} <\!|\!> \textit{somewhereOr}\ (\textit{nnfStep} \triangleright \textit{distrAndS}))\end{aligned}$$

Using *repeat* at the beginning of the rewriting strategy ensures that at each step we apply the complete rewriting strategy again, and hence simplify whenever possible.

In some cases *dnfStrategy* does not further simplify a formula. It simplifies $p \vee (q \wedge \neg q \wedge r)$, but not $p \vee (q \wedge r \wedge \neg q)$, because q and $\neg q$ are not adjacent formulae in the conjunct. We introduce a rewriting strategy *groupLiterals*, the definition of which is omitted, that checks if rearranging conjuncts makes it possible to apply a simplification. If this is the case, conjuncts are rearranged such that equal or negated conjuncts appear next to each other. We also use this substrategy in our rewriting strategy for proving the equivalence between two formulae later in this section.

A second rewriting strategy that we add to our *dnfStrategy* is a specialization of the distribution of disjunction over conjunction. In general, we do not allow distribution of disjunction in our rewriting strategy. However, if the distribution can be followed by a complement rule, it simplifies the formula. For example, applying distribution to $p \vee (\neg p \wedge q)$ leads to $(p \vee \neg p) \wedge (p \vee q)$, which can be simplified to $p \vee q$. For the same reason, if a formula is of the form $\phi \wedge (\neg\phi \vee \psi)$, distributing and over or before a possible application of DeMorgan shortens the derivation. We define a substrategy *distrNot* that is only used if after an application of a distribution rule a complement rule is applicable.

A third rewriting strategy, *deMorganNot*, checks whether an application of De-Morgan leads to the possibility to simplify.

The improved definition of *nnfStep* looks as follows:

$$\begin{aligned} \mathit{nnfStep} = {} & \mathit{simplifyStep} \\ & \triangleright (\mathit{groupLiterals} >\!|\!> \mathit{distrNot} >\!|\!> \mathit{deMorganNot}) \\ & \triangleright (\mathit{eliminateImplEquivS} >\!|\!> \mathit{deMorganS}) \end{aligned}$$

The rewriting strategy ends if there are no steps left that can be applied. Possibly the rewriting strategy reaches a normal form before it ends. The finished service is used to check whether a formula is indeed in normal form. We allow a student to simplify a formula even if a normal form is reached.

2.6.2 Adapting a strategy

We hypothesise that one of the reasons for the many variants of LEs for logic is that every teacher uses her own strategy or rule set. Our framework supports adapting rewriting strategies or rule sets. Section 2.6.1 shows how to define variants of the DNF strategy.

Another way in which a teacher can adapt feedback services is by changing the rule set. Our DNF strategy is structured in a way that makes it easy to adapt the rewriting strategy for users who apply more, fewer, or different rules. Our basic strategy contains five substrategies that can be considered as sets of rules: *orRulesS*, *simplifyStep*, *eliminateImplEquivS*, *deMorganS* and *distrAndS*. Note that the first four substrategies turn a formula into NNF, and the last substrategy turns a formula in NNF into DNF. To modify a rewriting strategy, a teacher can change the content of any of these five sets. To guarantee that a modified strategy still returns a normal form, the modification has to satisfy certain criteria, described in (Lodder et al., 2015a).

2.6.3 A rewriting strategy for proving two formulae equivalent

This subsection discusses a rewriting strategy for proving two formulae equivalent. This strategy builds upon the strategy for rewriting a formula into DNF. In particular, we discuss the heuristics used in this rewriting strategy.

The basic idea behind our strategy for proving two formulae equivalent is simple: rewrite both formulae into DNF and prove that the two resulting formulae are equivalent by extending the normal forms Lodder and Heeren (2011). However, without including heuristics in this strategy, students do not get the necessary strategic insight for this kind of problems, and derivations may become rather long.

The first heuristic we use in our rewriting strategy is a general principle that divides a problem in smaller subproblems. In our case this means that if we for example want to prove $\phi \Leftrightarrow \psi$ and ϕ and ψ are both conjunctions: $\phi = \phi_1 \wedge \phi_2$, $\psi = \psi_1 \wedge \psi_2$, we first check using truth-tables whether or not $\phi_1 \Leftrightarrow \psi_1$ and $\phi_2 \Leftrightarrow \psi_2$ hold, or $\phi_1 \Leftrightarrow \psi_2$ and $\phi_2 \Leftrightarrow \psi_1$. If so, the rewriting strategy splits the proof in two subproofs, applying commutativity if necessary. The same steps are performed if ϕ and ψ are both disjunctions, negations, implications or equivalences. For example, a proof of

$$(p \wedge p) \rightarrow (q \wedge (r \vee s)) \Leftrightarrow p \rightarrow ((s \vee r) \wedge q)$$

takes only three steps: two applications of commutativity and one of idempotency. The rewriting strategy does not rewrite the implication, nor distributes and over or. After ϕ and ψ have been rewritten into simplified DNF, the heuristic rearranges conjuncts and disjuncts to try proving equivalence. Since normal forms are not unique, we rewrite these normal forms into complete normal forms in some cases. In a complete normal form, each conjunct contains all the occurring variables, possibly negated. Complete normal forms are unique up to commutativity. Since rewriting into complete normal forms may take quite a number of steps, and the resulting formulae may get very long, we introduce two additional heuristics.

We use inverse distribution to factor out common literals in the disjuncts of ϕ and ψ. When we rewrite a formula into complete normal form, we do not use distribution anymore, and hence prevent a loop. We now have to prove the equivalence of two simpler formulae, which might not make the proof shorter, but at least the formulae are smaller.

A complete normal form is seldom necessary. We considerably shorten proofs by using splitting rules of the first heuristic during normalization, together with applications of the absorption rule. The subformula we choose to extend to normal form influences the length of the proof too. For example, we do not choose subformulae that occur on both sides.

To evaluate our rewriting strategy we asked 4 human experts (theoretical computer scientists and logicians) to solve a set of 6 'independent' exercises[7]. The expert solutions to two exercises were almost equal to the solutions of LogEx, up to a few differences in the order of the applied rules. One expert found a quicker solution to one exercise. In another exercise LogEx found a short cut which was overlooked by the experts. The experts found shorter solutions to the other three exercises. LogEx constructs a proof via a DNF while in some cases a proof via CNF is shorter, and using inverse distribution more often than in our strategy also helps. From the 187 steps taken by the experts in their solutions to the exercises 169 (= 90%) were recognised as part of the strategy by LogEx.

[7] `https://en.wikibooks.org/wiki/Logic_for_Computer_Scientists/Propositional_Logic/Equivalence_and_Normal_Forms#Problems`

	pre-test	post-test
Completion	0,54	1,0
Mistakes	0,84	0,7

Figuur 2.4: Results pre- and post-test December 2015

2.7 Experimental results

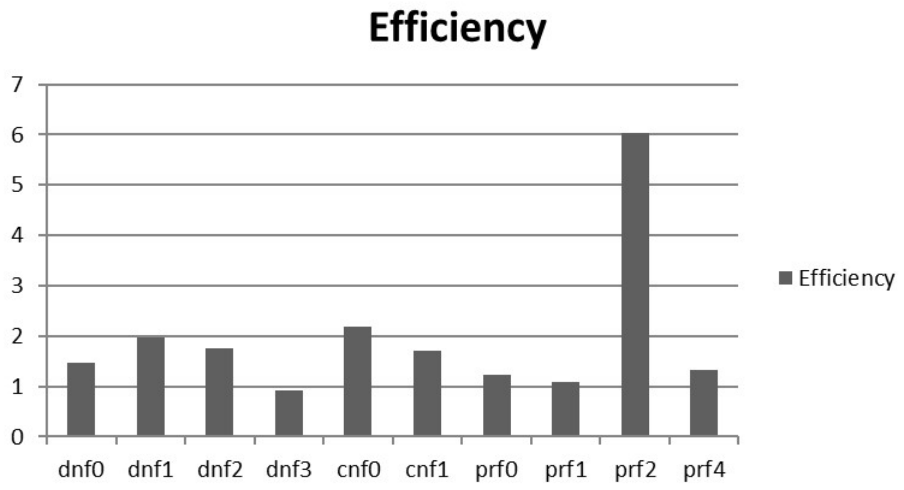
Since the first version of LogEx we performed several small-scale experiments with students (Lodder et al., 2008). In this section we describe the results of three experiments carried out in December 2014, December 2015 and February 2016.

In December 2014 we organised a pilot study with the LogEx learning environment (Lodder et al., 2015b). In this pilot study we used pre- and post-tests together with an analysis of the log files to answer the question whether using LogEx helps students to reach the following learning goals: after practicing with the LE, a student can

- recognise applicable rules
- apply rules correctly
- rewrite a formula in normal form
- prove the equivalence of two formulae using standard equivalences
- demonstrate strategic insight in how to rewrite a formula in normal form or prove an equivalence in an efficient way.

The number of the students participating in this experiment (5) was too low to draw firm conclusions. However, the results of the study indicate that indeed LogEx helps students to reach the learning goals, except for the last goal. We hypothesise that the reason that students do not improve in efficiency is because LogEx does not provide strategic feedback.

In December 2015 we repeated the pilot evaluation with a group of 8 students. Again we used pre- and post-tests and analysed the loggings. We graded the tests in two ways: we measured the completion (percentage of completed exercises) and the number of mistakes per completed exercise. The results are given in Figure 2.4. Although we constructed pre- and post-tests such that the difficulty of both tests is comparable, we have no objective criterion to measure difference in difficulty. Still, the results on the pre- and post-tests indicate that indeed students learn to apply rules correctly, and to rewrite a formula in normal form or prove the equivalence between two formulae. The analysis of the loggings also showed that while working with LogEx, students become more skilled in applying the appropriate rules. In contrast to the results of the 2014 test, here we find that students do learn to work more efficiently. We measure efficiency by dividing the number of steps of a completed exercise by the number of steps of the example solution generated by LogEx.We only took exercises that were completed by at least half of the students



Figuur 2.5: Logging analysis for December 2015 (measuring efficiency)

into account. Figure 2.5 shows the results. The exercises are presented in the order in which they were completed: students started with the first exercise on rewriting a formula to DNF, and ended with the fifth exercise on proving an equivalence. Note that the first exercise on deriving a CNF (*cnf0*) was solved much less efficiently than the preceding DNF exercises: students had to find out how to adjust their solution strategy. Exercise *prf2* has a short solution. If a student does not find this solution, the alternative is long, which explains the high value for this exercise.

In February 2016 we performed a small experiment with students from the Open University of the Netherlands. Our main research question for this experiment was:

- to what extent do feedback and feed forward contribute to the learning of the students?

The Open University is a university for distance education with students living all over the Netherlands. Hence the experiment was performed in the context of distance learning. After a short instruction via an on-line learning environment, students worked on a 20-minute pen and paper pre-test. After sending in the answers, they received access to LogEx, in which they practiced with DNF, CNF and proof exercises. The experiment was concluded with a 20 minute pen and paper post-test. 15 students participated, but only 12 handed in both pre- and post-test. In our analysis we restrict ourselves to these 12 students. They were divided into two groups of 6 students. One group (group A) could use the full functionality of LogEx, the second group (group B) received no hints and next steps. Students in the second group got only feedback after finishing their exercise. Students in both

	Group A	Group B
Completion pre-test	0,44	0,47
Completion post-test	0,60	0, 53
Mistakes pre-test	1,0	1,2
Mistakes post-test	1,1	1,5

Figuur 2.6: Results pre- and post-test, Open University 2016

groups could ask for a worked out solution, for example to compare it with their own solution. Both pre- and post-test consisted of three exercises. As in the December 2015 experiment, we graded the tests in two ways: we measured the completion and the number of mistakes per completed exercise. We have no objective criterion for determining the difficulty of both tests, hence we cannot measure overall learning effects, but we can compare the results of both groups. As shown in Figure 2.6, both groups made the same amount of mistakes in the pre-test. In the post-test both made more mistakes (maybe due to the difficulty of the post-test or tiredness), but group A made fewer mistakes than group B. Completion of the pre-test was higher in group B than in group A, in the post-test these results were reversed. The results indicate that indeed the presence of immediate feedback and feed forward do increase learning. We also analysed the loggings. This analysis shows that students in group A spent more time working in LogEx than students in group B (on average 57 min. versus 41 min.) The average number of exercises that students worked on in LogEx is for both groups more or less the same: (8,3 versus 8,8), but in group B the deviation was greater: one student worked on only four exercises, another only five. If we omit the results of these two students from the pre- and post-test, the results on completion for group B are somewhat better than those of group A. This experiment suggests that the main reason for the difference in performance between the two groups is a motivational one: students practice more when they get feedback and feed forward, and hence their results are better. This explanation is confirmed by the remark of a student in group B who complained that she could not correct her mistakes and hence “did not learn anything”.

2.8 Conclusions

We have used a framework for describing rules and strategies to develop strategies that support students solving exercises in propositional logic. The framework provides services for analysing student steps, and for giving feed forward such as hints, next steps, examples, and complete solutions. Our approach guarantees that this feedback and feed forward is available at any time when solving an exercise, also if a student diverges from a preferred solution, or enters her own exercise.

We have shown how we can adapt our strategy with different rule sets. Since feedback and feed forward are provided by services separate from a user-interface or learning environment, it is easy to adapt the feedback and feed forward for different languages or logical symbols.

We have performed small-scale experiments with a learning environment built on top of the services described in this paper, and the results are promising. We will perform more experiments in the academic year 2016-2017, in which we will investigate whether our learning environment supports students in developing their skills and understanding of propositional logic.

Acknowledgements

We thank our students for experimenting with the tools and giving feedback. Maarten Hemker and Peter Dol developed the first version of a user interface for our learning environment for proving equivalences, and René Dohmen and Renaud Vande Langerijt extended this for exercises about rewriting to a normal form, and made some further improvements. Marco Huijben en Wouter Tromp developed a useful tool for analysing the loggings of LogEx. Expert solutions were provided by Nikè van Vugt, Hans van Ditmarsch, Fer-Jan de Vries and José Martín Castro-Manzano. We thank our anonymous referees for their constructive comments.

3 A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae

3.1 Introduction

Students learning propositional logic practice by solving different kinds of exercises. Many of these exercises are solved stepwise. To support a student solving such an exercise, an intelligent tutoring system can be very effective (VanLehn, 2011). These systems offer several kinds of assistance, for example step by step feedback, instructions to repair common errors, hints or next steps, or even complete solutions. The timing of this assistance varies: directly after the performance of a step or only after the completion of an exercise. Based on a review of the literature, Koedinger and Aleven (2007) state that offering assistance can make learning more efficient, but misuse of help can cause shallow learning. On the other hand, withholding information forces students to construct their own solution, which may benefit attention, but might waste time and result in confusion. Koedinger and Aleven introduce the term 'assistance dilemma', and review several experiments that compare different strategies for giving and withholding feedback. The conditions immediate versus delayed yes/no feedback were studied in an experiment with a Lisp tutor (R. Anderson et al., 1995) and an Excel tutor (Mathan and Koedinger, 2005). The first study concludes that immediate feedback causes students to learn faster and better than with feedback after completion of the exercise, but in the experiment with an Excel tutor, where learning to detect and repair mistakes was one of the goals, allowing initial errors resulted in better performance not only on a post test, but also on long term retention and transfer. An experiment with the Geometry Proof Tutor (Koedinger and Aleven, 2007) comparing explanatory feedback with yes/no feedback resulted in a significantly lower post-error rate in the explanatory feedback condition. The question whether a hint containing conceptual information is more effective than providing a next step is partially answered by a study that compares explanatory error messages with correcting next steps, where the former strategy turns out to be more effective. These experiments support the approach of balancing giving and withholding information taken in cognitive tutors. However, Koedinger and Aleven claim that the question of how to decide which information should be given at what moment is a fundamental open problem.

Studies on the assistance dilemma often address a particular subproblem, such as whether or not supplying worked examples results in more efficient learning. The outcomes of studies related to worked examples vary. While a comparison of untutored learning versus worked examples shows that worked examples are superior (Sweller and Cooper, 1985), the results of comparing tutored learning with worked examples are less clear, and may depend on the level of a student, exercise difficulty, or content (procedural versus conceptual) (Shrestha et al., 2009; Razzaq and Heffernan, 2009; Kim et al., 2009). Strategies where (untutored) problems are alternated with worked examples are superior when a worked example is followed by a problem instead of a problem followed by a worked example (van Gog et al., 2011). Offering a worked solution can be seen as a special case of providing a worked example. Compared to a situation where exercises are scaffolded by giving students hints, good students perform better when receiving a worked solution, but for average students this is the other way around (Razzaq et al., 2007). As far as we know, the question whether adding the possibility to ask for hints and next steps supports learning in a situation where a student can ask for a worked solution has not yet been studied yet.

Several models try to explain the effects of different assistance strategies. Chi (2009) introduces a framework to differentiate the terms 'active, constructive, and interactive' in terms of observable activities and underlying cognitive processes. She classifies physical activities as active, the production of output beyond the presented information as constructive, and performing a dialogue taking the partner's contributions into account as interactive. The involved cognitive processes are attending processes, creating processes and creating processes that incorporate a partner's contributions, respectively. She uses this classification to hypothesize that constructive processes have better learning results than active processes, and interactive processes have better results than constructive processes. Cognitive load theory is also used to explain differences between assistance strategies. According to Salden et al. (2010), worked examples reduce extraneous cognitive load and save time. The Interactive Tutoring Feedback Model (Narciss, 2013) introduces a framework that distinguishes an internal learner's feedback loop and an external feedback loop. The model suggests that learning not only depends on external factors such as content and timing of feedback, but also on learner characteristics. Narciss et al. (2014) study the influence of learner characteristics in an experiment with sixth and seventh graders working on fractions. One of the outcomes is that male students profit less from feedback than females.

A second subquestion of the assistance dilemma concerns the timing and amount of feedback when a student makes an error. Based on a review of the literature, Shute (2008) lists several guidelines to enhance formative feedback, but she does not give definitive answers. According to these guidelines, immediate feedback should be used for retention of procedural knowledge, and delayed feedback for transfer of learning. The question remains which approach is best for a particular domain of study.

In this paper we describe an experiment with LogEx[1], a learning environment (LE) that supports students in rewriting propositional logical formulae using standard equivalences. The learning goals addressed by LogEx are: after practicing with LogEx a student can

- correctly apply rewriting rules for propositional logic
- prove the equivalence of two formulae using standard equivalences
- demonstrate strategic insight in how to efficiently prove an equivalence

Here, an efficient proof is a solution that uses a minimal number of steps.

The main research question we investigate in this paper is: do students reach the above learning goals by practicing with LogEx? We also want to contribute to the assistance dilemma by investigating whether or not hints and immediate feedback have an effect on student learning. Do students who receive hints and feedback while practicing perform better than students who practice with a version of LogEx with just delayed feedback and worked solutions? We hypothesize that students who receive immediate feedback and who can use hints make fewer errors and can complete more exercises.

This paper is organized as follows. The next section reviews several evaluation studies with other LEs for rewriting logical formulae or proving logical consequences. We continue with describing LogEx in more detail, together with a short review of previous studies performed with LogEx. The experiment is described in Section 3.4. Section 3.5 presents and discusses the results of the assessment tests and loggings. Section 3.6 summarizes our conclusions and proposes future research.

3.2 Evaluation results from other LEs

This section discusses related work in educational experiments with logic tutors.

In a previous paper (Lodder et al., 2016) we reviewed six e-learning environments comparable with LogEx. Only one of these environments has been used in an experiment with students. In FOL (Grivokostopoulou et al., 2013), students rewrite first-order logical formulae using standard equivalences. Feedback is presented in stages: first a student chooses a rule that can be applied, and only after the system approves, the student can continue with the rewriting step. The designers of FOL compared a group of students who practice with the e-learning environment for one week 20 minutes a day with a control group of students who solve homework using pen and paper, discussed by the teacher afterwards. The results show a statistically significant better performance on a post test by the group who practiced with FOL.

[1]http://ideas.cs.uu.nl/logex/

We have found a number of evaluation studies using LEs for teaching logic focusing on different kinds of exercises.

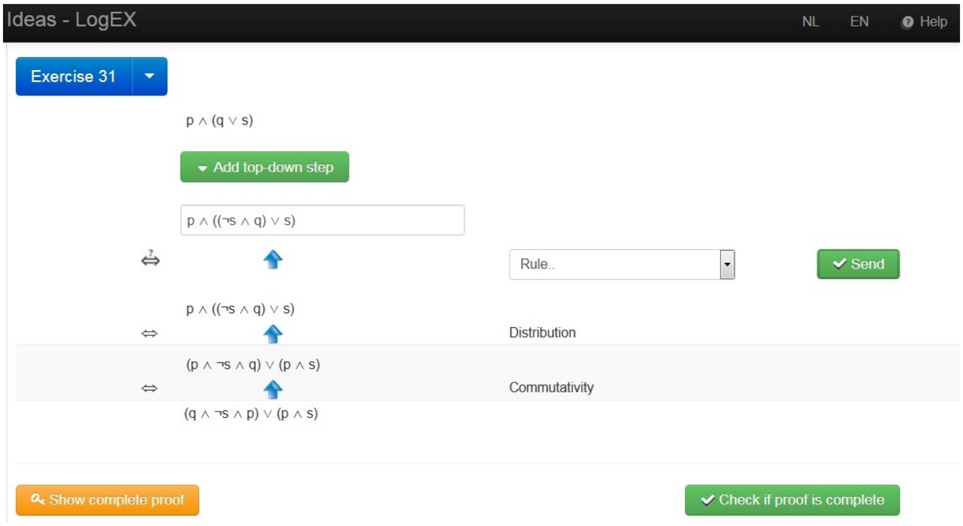
Logic Tutor (Yacef, 2005) supports learning how to prove a consequence using rewriting rules (such as DeMorgan) in combination with inference rules. It presents proofs in a linear form, and only allows rewriting in one direction. It provides feedback, for instance about a missing reference to a previous proof-line, at each step, but offers no hints or next steps. Student interactions are logged and can be analyzed by teachers, for example to improve their teaching. Several experiments with the Logic Tutor were performed in 2000–2003. Answers to exam questions show improvement from year to year, partly because of the use of the tutor by students, but also because of teachers analyzing the loggings of the tool.

Deep Thought (Mostafavi and Barnes, 2017; Stamper et al., 2011b) offers exercises comparable to Logic Tutor, but presents proofs as trees, and allows students to construct a proof by adding forward and backward steps. An evaluation study of Deep Thought addressed the question of whether the use of data-driven methods in problem selection and feedback in the development of Deep Thought influences student drop out and the time needed to complete the exercises in the tutor (Mostafavi and Barnes, 2017). A comparison of four versions of Deep Thought showed that in each new version student drop out and time to complete the exercises in the tutor decreased significantly. An experiment where students either solved a set of three or four problems, or watched the worked solution of one or two of these problems and solved another two, showed that worked examples reduced hint dependency for high proficiency students. Students who received two worked solutions constructed shorter solutions, but also made more mistakes. On the other hand, low proficiency students in the worked example condition made more mistakes and produced longer solutions than low proficiency students who did not receive worked examples (Liu et al., 2016). An earlier paper showed that students who could ask for hints performed significantly better than students who did not receive hints (Stamper et al., 2011b).

Miwa et al. (2014) describe an intelligent tutor to help a student with solving natural deduction problems. It contains a complete problem solver, which provides various kinds of support, for example, which rule can be applied to which formula, or which set of rules is applicable. An experiment with the LE showed that students who used the LE performed significantly better on easy post test exercises than a control group that received traditional classroom instruction. There was no significant difference in performance on the more difficult exercises.

3.3 LogEx

LogEx is a learning environment in which a student practices rewriting propositional logical formulae. LogEx contains three kinds of exercises: rewriting a formula in DNF, in CNF, and proving the equivalence of two formulae. A student enters



p ∧ (q ∨ s)

p ∧ ((¬s ∧ q) ∨ s)

p ∧ ((¬s ∧ q) ∨ s) ⇔ Distribution

(p ∧ ¬s ∧ q) ∨ (p ∧ s) ⇔ Commutativity

(q ∧ ¬s ∧ p) ∨ (p ∧ s)

Figuur 3.1: Screenshot of LogEx

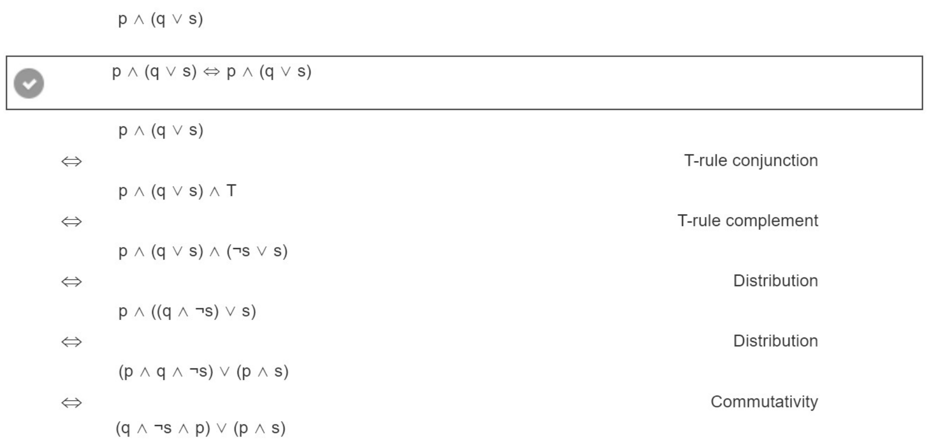
her solution stepwise. To illustrate the functionality of the LE, we give an example of how a student might solve an exercise in LogEx.

Suppose the student has to prove that:

$$p \wedge (q \vee s) \Leftrightarrow (q \wedge \neg s \wedge p) \vee (p \wedge s)$$

In LogEx, the left-hand side of this equivalence is shown at the top of the screen and the right-hand side at the bottom. A student might recognize that after swapping p and q in the bottom line it is possible to take the variable p out of the conjunctions by applying distribution in reverse. LogEx allows to rewrite the bottom formula, and after applying commutativity and distribution, the partial proof is of the form given in Fig. 3.1. The student can continue by rewriting the formula in the edit field, using shortcuts or a small keyboard to enter the logical connectives, and motivating the step by choosing the name of the rule applied from a drop-down list. The student can also change the direction in which she is working at any moment. For example, she could proceed by rewriting the line $p \wedge (q \vee s)$ at the top of the proof into $(q \vee s) \wedge p$.

In the complete version of LogEx, a student receives feedback after each step. Feedback concerns syntax errors, such as missing parentheses, or rule feedback. After a student enters a formula, LogEx tries to recognize the rule that is used. If it detects a rule, it compares this rule with the rule specified by the student, and gives an error message giving the correct rule name if the wrong rule name is specified. LogEx uses a set of common mistakes, also called buggy rules, to try to give informative feedback. For example, if a student rewrites $\neg(p \vee q) \vee (\neg\neg p \wedge$



$p \wedge (q \vee s)$

✓ $p \wedge (q \vee s) \Leftrightarrow p \wedge (q \vee s)$

$p \wedge (q \vee s)$

$\Leftrightarrow$ T-rule conjunction

$p \wedge (q \vee s) \wedge T$

$\Leftrightarrow$ T-rule complement

$p \wedge (q \vee s) \wedge (\neg s \vee s)$

$\Leftrightarrow$ Distribution

$p \wedge ((q \wedge \neg s) \vee s)$

$\Leftrightarrow$ Distribution

$(p \wedge q \wedge \neg s) \vee (p \wedge s)$

$\Leftrightarrow$ Commutativity

$(q \wedge \neg s \wedge p) \vee (p \wedge s)$

Figuur 3.2: The complete solution of the exercise in Figure 3.1, generated by LogEx

$\neg q) \vee \neg q$ into $(\neg p \vee \neg q) \vee (\neg\neg p \wedge \neg q) \vee \neg q$, then LogEx reports that this step is incorrect, and mentions that when applying DeMorgan's rule, a disjunction is transformed into a conjunction. If no rule or buggy rule is detected, LogEx checks whether or not the new and old formulae are semantically equivalent. If they are not equivalent, LogEx mentions that an error is made, otherwise the student receives a message that she either combined two or more steps in one, or made a mistake. In the version of LogEx discussed in this paper, a student can only proceed after correcting a mistake.

In LogEx, a student can ask for

- a hint, for example, in the situation of Fig. 3.1 LogEx first hints to rewrite the boxed formula, continuing in the same direction of the proof, and then to apply Distribution
- a next step, for example, LogEx rewrites $p \wedge ((q \wedge \neg s) \vee s)$ into $p \wedge ((q \vee s) \wedge (\neg s \vee s))$
- or a complete worked solution as shown in Fig. 3.2

at any moment. The LE uses solution strategies to calculate this feed forward. This strategy can be restarted after each rewriting, so that hints and next steps can also be given when a student diverges from the solution of the problem that is calculated by the LE. A student can choose between exercises of different difficulty levels, or enter her own exercise. Feedback and feed forward are available for all exercises, including user defined problems. LogEx integrates improved versions of earlier tools to rewrite formulas in disjunctive normal form (Lodder et al., 2006, 2008) and to prove equivalences (Lodder and Heeren, 2011).

3.3.1 Pilot studies

We have evaluated various aspects of LogEx in several pilot studies (Lodder et al., 2015b, 2016, 2008). We have used these pilot studies to evaluate the usability of LogEx, and to prepare for a large scale experiment (Shute and Regian, 1993). In our first experiments, we compared the complete version of LogEx, in which hints and next steps are available and a user gets feedback directly after performing a step, with a version without hints or next steps and a user receives postponed feedback. The number of participating students was too low to draw firm conclusions, but the loggings of the use of LogEx in these experiments indicated that (Lodder et al., 2008):

- the possibility to ask for a next step is essential for weaker students, the students who performed less well on the pretest. Students who used a version of LogEx without the availability of next steps could not complete more complicated exercises.
- the availability of next steps teaches students to use rules they overlook (for example, false-true rules to simplify an expression).
- the requirement to perform one step at a time forces students to recognize mistakes they would overlook otherwise. An example of such a mistake is applying distributivity on equal connectives, which results in an equivalent formula, but is not a correct application of distributivity.
- since learning an efficient strategy is implicit in LogEx, students who do not use the hint and next step button can proceed with inefficient strategies without receiving feedback on this aspect.

In a second experiment (Lodder et al., 2015b), analysis of the loggings showed that during the experiment students gradually need less time to complete an exercise, and feedback helps students to recognize and correct their mistakes.

3.4 Method

In September 2017 we performed an experiment with LogEx at a university of applied sciences. The participants were second year computer science students taking a course in discrete mathematics, which has propositional logic as one of its topics. Students have to learn to simplify formulae using standard equivalences and to prove the equivalence of formulae.

3.4.1 Pilot

To prepare for the experiment, we performed a pilot with 13 part-time students in May 2017. This experiment took place directly after class-based instruction on equivalences. The pilot consisted of:

- a short introduction about the purpose of the experiment and instruction on how to use LogEx.
- a 20-minute pre test consisting of three exercises comparable to the LogEx exercises.
- working with LogEx for 50 minutes.
- a 20-minute post test consisting of three exercises comparable to the pre test.

Students were divided into two groups. One group used the complete version of LogEx, the other group could not use hints or next steps and only received check marks for correct steps after completing an exercise. The latter group could also ask for a worked solution, and compare it with their own solution. All students could use a formula sheet so that they did not have to memorize the logic rules.

The main outcome of this experiment was that students scored very low on the pre test. On average students completed only half of the first exercise, 5% of the second, and nothing of the third. This implies that the pre test cannot be used to differentiate between student levels.

Since these results are not very encouraging for students, and not very useful for teachers and researchers, we changed our experiment in two ways. First, we planned the experiment the week after class-based instruction of standard equivalences. This way, students could review the topic before the experiment and already practice a bit. Second, we replaced the first exercise of the pre test with an exercise that was slightly easier.

3.4.2 Experiment

Three classes with a total of 74 students participated in the experiment. The participants were males between 19 and 31 years. We compared two conditions: the use of the complete version of LogEx with elaborated feedback (Narciss, 2008), versus the version without hints and next steps, and with delayed checkmark feedback, see Table 3.1.

To validate the pre test and post test, we divided both groups into two subgroups, for which we used the two variants of the pre test and post test given in Table 3.2. Both tests consist of three exercises. We used exercise 1 of the pre test to measure the difference in rewriting skills between the groups before the start of the experiment, and hence offered this exercise to both groups. The first exercise in the post

functionality	full LogEx	restricted LogEx
hints	✓	×
next step	✓	×
complete solution	✓	✓
immediate feedback	✓	×
informed feedback	✓	×
delayed checkmark feedback	×	✓

Tabel 3.1: Functionalities of the full and restricted versions of LogEx

test is a slightly more complicated variant of the first exercise in the pre test. Exercises 2 and 3 of pre test version 1 are the same as exercises 2 and 3 in the post test in version 2, and vice versa. We used these exercises to measure learning gains, as described for example by Bartsch et al. (2008). We use the following abbreviations for the subgroups of students:

- F1: students using the full version of LogEx and test 1
- F2: students using the full version of LogEx and test 2
- R1: students using the restricted version of LogEx and test 1
- R2: students using the restricted version of LogEx and test 2
- F: F1 + F2, all students using the full version of LogEx
- R: R1 + R2, all students using the restricted version of LogEx
- 1: F1 + R1, all students taking test 1
- 2: F2 + R2, all students taking test 2

Table 3.5 shows the number of students in each group.

The organization of the experiment was comparable to the pilot: a short introduction, a 20-minute pre test, followed by 50 minutes practicing with LogEx, and, after a short break, a 20-minute post test. Students were allowed to use a formula sheet during pre test, practicing and post test. We logged the use of LogEx. The list of twelve exercises used in LogEx can be found in the appendix. All raw data is available via data.mendeley.com[2].

	Test 1
pre test	
1.	$q \rightarrow \neg(p \vee q) \Leftrightarrow \neg q$
2.	$(p \vee q \vee r) \wedge (r \vee \neg p) \Leftrightarrow (q \wedge \neg p) \vee r$
3.	$((\neg p \vee q) \wedge p) \vee (\neg(\neg p \vee q) \wedge \neg p) \Leftrightarrow q \wedge p$
post test	
1.	$((\neg p \vee q) \wedge \neg p) \rightarrow p \Leftrightarrow p$
2.	$(p \wedge q) \rightarrow (q \wedge r) \Leftrightarrow q \rightarrow (p \rightarrow r)$
3.	$((p \wedge q) \vee (\neg p \wedge \neg q)) \rightarrow p \Leftrightarrow p \vee q$

	Test 2
pre test	
1.	$q \rightarrow \neg(p \vee q) \Leftrightarrow \neg q$
2.	$(p \wedge q) \rightarrow (q \wedge r) \Leftrightarrow q \rightarrow (p \rightarrow r)$
3.	$((p \wedge q) \vee (\neg p \wedge \neg q)) \rightarrow p \Leftrightarrow p \vee q$
post test	
1.	$((\neg p \vee q) \wedge \neg p) \rightarrow p \Leftrightarrow p$
2.	$(p \vee q \vee r) \wedge (r \vee \neg p) \Leftrightarrow (q \wedge \neg p) \vee r$
3.	$((\neg p \vee q) \wedge p) \vee (\neg(\neg p \vee q) \wedge \neg p) \Leftrightarrow q \wedge p$

Tabel 3.2: Pre test and post test

3.5 Results and discussion

3.5.1 Results of pre test and post test

The first exercise in the pre test, which was the same for all students, was used to test whether the prior knowledge of all groups was comparable. We scored the exercise in two ways. The first score is the completion rate of the exercise: the number of completed steps divided by the total number of a completed version of the student's solution. The second score is the relative number of incorrect lines (the number of incorrect steps divided by the total number of steps). The first score is used to measure the learning goal 'being able to prove equivalence', the second to measure the learning goal 'applying the rules correctly'. Some students make a mistake in the first or second line, but continue without mistakes, which may result in a shorter solution. Since we do not know whether students are able to finish the exercise had they not made the mistake, we grade these cases as follows: the grade consists of the number of completed steps divided by the total number of steps in the standard solution. The descriptive statistics of both measures are shown in Table 3.3. The statistics indicate that differences between the four groups F1, F2,

[2]doi:10.17632/4wdj3b2t5g.1

Completion	group 1		group 2		total	
	mean	std dev	mean	std dev	mean	std dev
Full LogEx	0.86	0.27	0.81	0.29	0.84	0.27
Restricted LogEx	0.83	0.30	0.82	0.18	0.83	0.26
Total	0.84	0.29	0.82	0.22	0.84	0.27

Relative number of incorrect lines	group 1		group 2		total	
	mean	std dev	mean	std dev	mean	std dev
Full LogEx	0.12	0.21	0.15	0.34	0.13	0.25
Restricted LogEx	0.20	0.33	0.12	0.22	0.17	0.29
Total	0.17	0.28	0.13	0.26	0.15	0.27

Tabel 3.3: Descriptive statistics of completion and relative number of incorrect lines in the first exercise of the pre test

R1, and R2 are small, although group R seems to make some more mistakes.

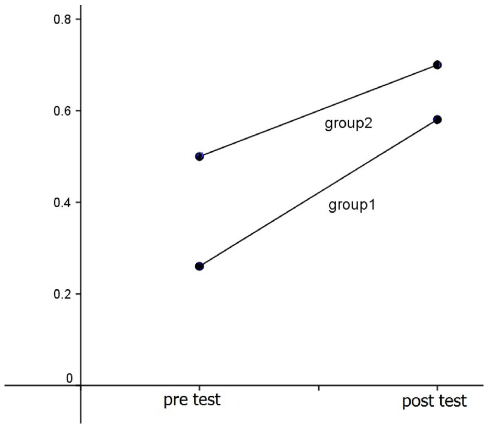
	completion	relative number of incorrect lines
chi square	1.7	1.1
df	3	3
p	0.65	0.79

Tabel 3.4: Results of the Kruskal Wallis test on differences between group F1, F2, R1 and R2 in performance in completion of, and relative number of incorrect lines in, pre test exercise 1

We use non-parametric tests to compare the distribution of the variables completion rate and relative number of incorrect lines in the four different groups F1, F2, R1, and R2, since the Kolmogorov-Smirnov test on normality of the variables completion rate and relative numbers of incorrect lines fails. The results can be found in Table 3.4. The outcome of the Kruskal Wallis test indicates that there is no difference in the distribution of these variables between the different groups. A comparison of group F versus group R, and of group 1 versus group 2, also shows no significant difference. We use a Mann Whitney U test with threshold p = 0.05 (Nachar, 2008; Hayes, 1988), and find significance levels of 0.58 for completion and 0.50 for relative numbers of incorrect lines when we compare group F with group R, and significance levels of 0.26 for completion and 0.48 for relative number of incorrect lines when we compare group 1 with group 2. We conclude that prior knowledge was evenly distributed between the four groups F1, F2, R1, and R2, and that the difference in the number of mistakes between the groups F and R is not

Group	Test 1		Test 2		Total
Full LogEx F	F1	21	F2	9	30
Restricted LogEx R	R1	29	R2	15	44
Total		50		24	74

Tabel 3.5: Number of students in different groups

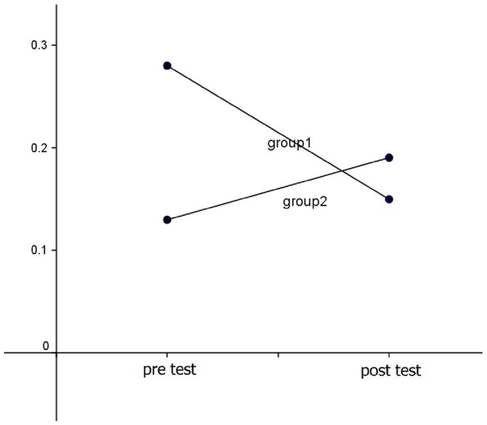


Figuur 3.3: Pre and post test completion rates on exercise 2 and 3

significant. Since we only look at differences between pre and post test, a small variation between the groups does not influence our conclusions.

Our first research question is: do students learn by using LogEx, or, more precisely, do students learn to apply rules correctly, prove the equivalence of two formulae using standard equivalences, and solve these exercises efficiently. We use the second and third exercise of the pre and post test to answer the first and second subquestion. To correct for a possible difference in the level of difficulty between the exercises in the pre and post test, we divided the students into four groups F1, F2, R1, and R2 as described in Table 3.5.

First we looked at the overall knowledge gain, independent of the version of LogEx, which means that we take groups F1 and R1 (group 1) and F2 and R2 (group 2) together. For group 1 and 2 we compared completion of exercises 2 and 3 in the pre test with completion in the post test. Fig. 3.3 shows the results. The graph indicates that pre test 1 (= post test 2) might be more difficult than pre test 2 (= post test 1), but both groups complete more of the exercises in the post test than in the pre test. This is confirmed by tests on effect size: Cohen's d for group 1 equals 0.72 (confidence interval [0.59, 0.82]), and for group 2 equals 0.42 ([0.20, 0.58]), so despite the possibly more difficult post test in group 2 the effect



Figuur 3.4: Relative numbers of incorrect lines in pre and post test exercise 2 and 3

can be classified as medium–high. The results for the relative number of incorrect lines were less conclusive: students in group 1 made relatively fewer mistakes in the post test than in the pre test (Cohen's d $= -0.54[-0.60, -0.44]$), but in the second group this was the other way around (Cohen's d $= 0.25[0.15, 0.36]$), see Fig. 3.4. Note that in this case a negative number means relatively fewer mistakes. Although the second group made relatively more mistakes in the post test, the decrease in mistakes in group 1 was larger than the increase in group 2.

To interpret the numbers on effect size, we compare them with Hattie's list of effects ranks. He mentions "Computer aided instruction" with an effect size of 0.37, and an effect size of 0.6 is reached, for example, by teaching strategies or problem-solving teaching (Hattie, 2012). Compared to other interventions, the effect of practicing with LogEx on completion is indeed substantial. We conclude that working with LogEx helps students to learn how to prove equivalence between formulas. Although our measure for errors already takes into account the total number of steps in a solution, the inconclusive results on errors might be explained by the fact that since students complete more of the exercises, they will also have to rewrite more complicated formulae. Since students could use a formula sheet, errors are not caused by incorrectly remembered rules. There are several other sources of errors, such as sloppiness, misunderstanding, overgeneralization, or just creative rule interpretation to finish a proof. We looked more closely at the errors made, but concluded that without asking students, it is hard to categorize the errors. For example, a student who forgets to change a disjunction into a conjunction while applying DeMorgan, may misunderstand the rule but may also be sloppy. In the same way, distributing a conjunction over a conjunction may be caused by sloppiness, but also by overgeneralization. We think that a large part of the mistakes are slips, and that practicing with LogEx for 50 minutes is too short to

Group 1	norm. compl. gain			relative error gain			error gain		
	n	median	mean	n	median	mean	n	median	mean
Full LogEx	21	0.15	0.21	15	-0.21	-0.26	21	0	0.33
Restr. LogEx	29	0.11	0.13	16	-0.04	-0.07	29	0	0.38

Group 2	norm. compl. gain			relative error gain			error gain		
	n	median	mean	n	median	mean	n	median	mean
Full LogEx	9	0.07	0.17	8	0	0.03	9	0	0.33
Restr. LogEx	14	0.04	0.10	14	-0.02	0.03	15	1	0.33

Tabel 3.6: Descriptive statistics of normalized gain in completion exercise 2 and 3, relative error gain and error gain

address this. Fatigue might also have influenced these results, as may have the fact that the students knew they were not going to be graded based on their results.

To answer the question whether giving feedback and hints has an effect on student learning, we compare the results of the group using the full version of LogEx with the restricted version. We compare the normalized knowledge gain between group F1 and R1 and between F2 and R2. Normalized knowledge gain is defined by: normalized gain = $(post - pre)$ / $(100 - pre)$ where *post* en *pre* can reach values between 0 and 100 (Hake, 1998).

We use the completion rates of exercises 2 and 3 as results of the pre and post tests. Since the maximum score for the completion rate of the exercises is 2, we use the following variant of normalized gain:

$$\frac{post2 + post3 - pre2 - pre3}{2 - pre2 - pre3}.$$

We also compared the relative error gain:

$$relative\ error\ gain = \frac{errorpost2 + errorpost3}{\#post2 + \#post3} - \frac{errorpre2 + errorpre3}{\#pre2 + \#pre3}$$

where *errorpre2* is the number of lines containing one or more errors in pre test exercise 2, and *#pre2* is the number of lines in the student submission of exercise 2 in the pre test. The definition of the other variables is similar. Since quite a number of students did not fill out any line in exercise 2 or 3 in the pre test, we also compared the absolute number of errors made in pre and post test. The results can be found in Table 3.6.

We used a Mann Whitney U test to examine whether the users of the full version of LogEx performed significantly better than the users of the restricted version. According to the test, this result is not statistically significant, see Table 3.7.

There are several reasons why the differences between group F and group R are small. Although group R could not use hints or next steps, they could ask for

Group 1	normalized gain	relative error gain	absolute error gain
Mann-Whitney U	257.5	86	278
Z	-0.93	-1.35	-0.55
p	0.18	0.092	0.298

Group 2	normalized gain	relative error gain	absolute error gain
Mann-Whitney U	54	53	64
Z	-0.57	-0.21	-0.22
p	0.29	0.43	0.44

Tabel 3.7: Results of the Mann Whitney U test on differences between users of the full version of LogEx versus the restricted version, for group 1 and group 2

a complete solution, and use this as a worked example. Learning with worked examples can be very effective (Sweller et al., 2011), and in paragraph 3.5.3 we will show that group R indeed used the complete solution to get a hint. Where most of the studies described by Koedinger and Aleven (2007) showed better results for immediate and informed feedback, in our experiments the effect on the number of errors is not significantly different between the two groups. A possible reason might be that our students could use a formula sheet, which makes informed feedback partly superfluous. Since male students profit less from feedback than female students Narciss et al. (2014), our 100% male population might be another explanation for the non significant effects. Students worked individually on the pre and post test, but could help each other while working with LogEx, and we actually observed this. As argued by Chi (2009), helping each other might make more difference than the presence or absence of feedback. Another reason could be that the experiment was too short to yield significantly different results between both versions of LogEx. The opposite results for high and low proficient students in the study by Liu et al. (2016) suggest that a separate analysis for these groups might yield significant results. However, the number of students in our experiment was too low to perform such an analysis.

We also wanted to find out whether students learn to solve exercises efficiently, by which we mean that students construct short solutions. We measure efficiency by dividing the total number of steps a student takes to solve an exercise by the number of steps of a worked solution generated by LogEx. Hence, a low score means an efficient solution. When a student finds a shorter solution than LogEx this score is less than 1, which actually happened in a few cases (for three exercises, with respectively two, five and one student). Efficiency is only measured when a student finishes an exercise correctly. In the pre test and post test the number of correct solutions for exercise 2 and 3 was too low to draw conclusions. Students who finished the first exercise in the pre test found an efficient solution (efficiency = 1 in group F and 1.2 in group R). The solutions of the slightly more difficult exercise 1 in the post test were less efficient (1.6 for both groups). We conclude that the pre and post test do not provide enough information to decide whether students develop strategic insight.

3.5.2 Exam results

To measure the medium-term effect of the use of LogEx, we analyzed the results of two exam questions. The exam took place five weeks after the experiment and contained two questions on rewriting propositional formulae, besides other questions in discrete mathematics. In the first question students had to simplify a propositional formula using rewrite rules, in the second question they had to prove an equivalence. One hundred and eleven students took the exam, 43 of which did not participate in the experiment. Of the remaining 68 students, 30 practiced with the full version of LogEx and 38 with the restricted version. Most of the students who did not participate in the experiment were taking a resit. The scores of this group are much lower than those of the other students. In the following we denote these students by group N. The maximum score for the exam was 100 points, 6 of which could be earned by correct answers to the questions on rewriting logical formulae. The results of the students on the questions on rewriting logical formulae can be found in Table 3.8.

Group	n	logic exercise		total	
		mean	std dev	mean	std dev
Group N	43	2.5	2.5	46.5	15.8
Group F	30	4.1	2.1	50.8	17.2
Group R	38	3.5	2.4	55.8	19.5

Tabel 3.8: Exam results for the exercises on rewriting logical formulae and the total exam score

On average, the students using the full version of LogEx performed better on the rewriting logical formulae questions than the users of the restricted version. They performed slightly worse on the overall results of the exam. The difference in performance when working with LogEx was not statistically significant, see Table 3.9.

Since we did not have results of a pre test of the students who did not participate in the experiment, we cannot compare their results with the students who did participate. However, since we have their exam results, we can use these as a measure of the general level and compare the difference of this general level with the results on the rewriting items. Therefore, we normalize the results by dividing

	logic exercises
Mann-Whitney U	502.5
Z	-0.88
p	0.2

Tabel 3.9: Results of the Mann Whitney U test on differences in the results on the logic exercises in the exam between users of the full version of LogEx versus the restricted version

the total score by 10 and multiplying the score for the logic questions by 10/6, and subsequently we look at the difference between these normalized scores. For example, a student with 60 points in total (normalized 6) and 5 points for the logic questions (normalized 8.3) scores 2.3 better on the logic question than expected. This logic score versus total score is normally distributed, and hence we can use a one way ANOVA test and post hoc tests to compare the differences. Again, group F performs better than group R, and group R performs better than the students who did not participate. Table 3.10 shows the descriptives.

Group	n	mean	std dev	95% confidence interval
Group N	43	-0.54	3.53	[-1.63 , 0.54]
Group F	30	1.75	2.92	[0.66, 2.84]
Group R	38	0.29	3.37	[-0.81, 1,41]
Total	111	0.36	3.42	[-0.27, 1.01]

Tabel 3.10: Descriptive statistics of the difference between the results of the logic exercises and overall exam performance

The effect of practicing with LogEx on the difference is significant, F(2, 108) = 4.23, p = 0.017. Post hoc comparisons using the Tukey HSD test indicate that group F performs significantly better than group N. The other comparisons do not show a significant difference, see Table 3.11.

Groups	mean difference	std error	sig
N versus F	-2.30	0.79	0.012
N versus R	-0.84	0.74	0.49
F versus R	1.46	0.81	0.18

Tabel 3.11: Post hoc comparison using Tukey HSD test of the difference of normalized scores from the logic exercises and the exam results for the three groups

This is an interesting result. It seems to indicate that in general students have more problems with the logic questions than with the other questions of the exam, but that after practicing with LogEx this is the other way around.

3.5.3 Results of the loggings

We analyzed the loggings of LogEx to answer the question whether students learn while working with LogEx, and to detect possible differences between the groups using the full and restricted version. The logging data consists of all the steps students take, all the hints, next steps, or worked solutions they ask for, together with time stamps. We analyze the loggings in various ways. We determine:

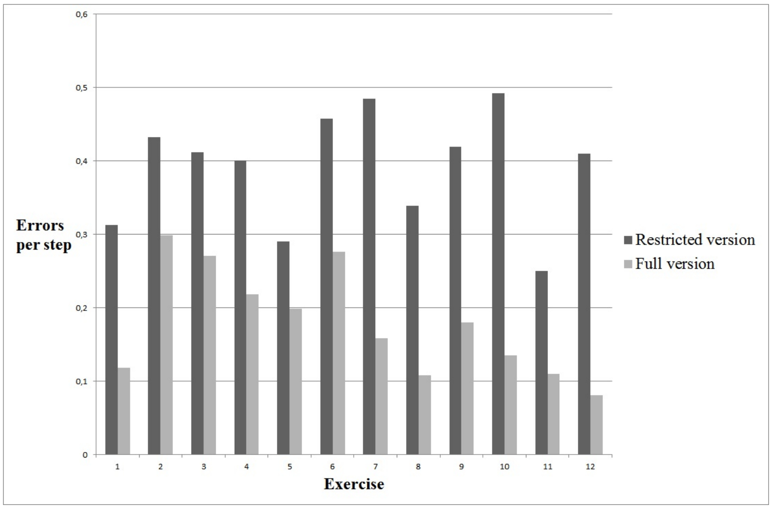
- the number of mistakes students make over time, and whether or not this number decreases.
- what kind of mistakes students make.
- how many of the exercises students complete.
- the time students need to take a step, and whether or not this decreases the longer they work in LogEx.
- how long student solutions are compared to the solutions generated by LogEx.
- at what point students in group F use hints and next steps.

In the rest of this section we describe each of these aspects in detail.

Students may show progress by making fewer mistakes after practicing with LogEx for some time. However, this progress may not be present in the data, since the first exercises in LogEx are rather simple while the last exercises are more complicated and present a student with longer formulae. Fig. 3.5 shows the number of erroneous steps per total number of steps for each exercise.

Both groups make many more mistakes in the second exercise than in the first. Group F gradually makes fewer mistakes except for exercises 6, 9 and 10 (see the appendix for the list of exercises). The last two exercises require more complicated steps, which leads to more mistakes. In exercise 6, students tend to perform more than one step at a time, which is not allowed. Group R does not make fewer mistakes while working with LogEx.

Further inspection of the loggings shows that students in group R perform multiple steps at a time also in the other exercises, and they do this much more often than the students in group F, probably because a student in group F cannot proceed with an exercise after performing several steps simultaneously. The difference in the number of mistakes per step between the two groups is mainly due to these multiple steps error, but when we correct for these errors, the number of errors made by students in group R still does not decrease while working with LogEx.



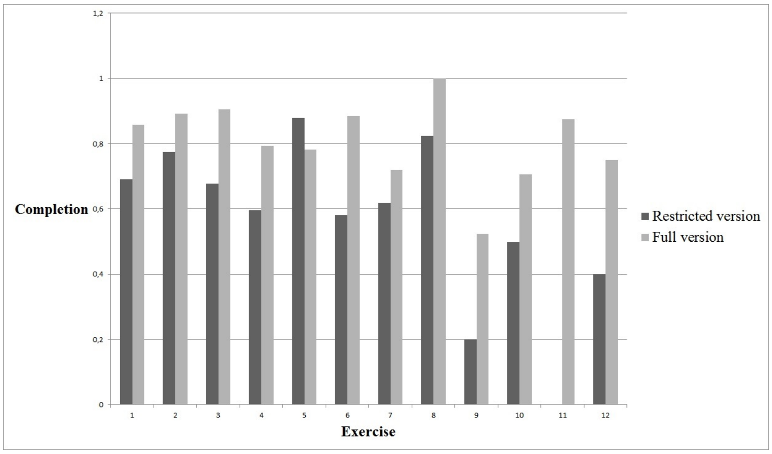
Figuur 3.5: Errors per step for each exercise

Since students in group F could not continue an exercise before correcting an error, these students might have been more careful when taking steps after some practice with LogEx.

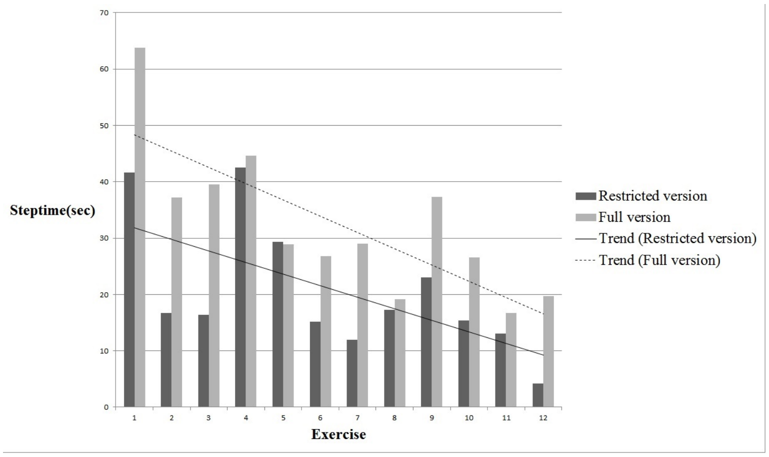
We also examined the completion rate of exercises in our loggings. Here we measure the percentage of students that complete an exercise from the number of students that started the exercise and took at least one step. The results are shown in Fig. 3.6. In general, students from group F complete more of the exercises than students from group R, and this difference is larger in the more difficult exercises. This is in line with our findings in the pilot studies: students need hints and next steps to complete an exercise.

Another way to examine whether students learn to solve exercises while working with LogEx is by measuring the time it takes to perform a step. Obviously, practice makes perfect, and we expect that with practice, the time to perform a step decreases. In the first exercises, students familiarize themselves with LogEx, but we expect that while working with LogEx, they decide faster which rule to apply. Fig. 3.7 shows the average step time per exercise. This varies per exercise (with outliers for the more complicated exercises), but the trend line suggests that students in both groups gradually solve exercises faster. Fig. 3.7 suggests that after the first exercise, students have learned to use LogEx, and they complete the rather easy second and third exercise much faster. They need more time to solve the more complicated fourth exercise, after which the step time gradually decreases. This is in line with our pilot experiments.

We compared the efficiency of working with LogEx for both groups. Fig. 3.8

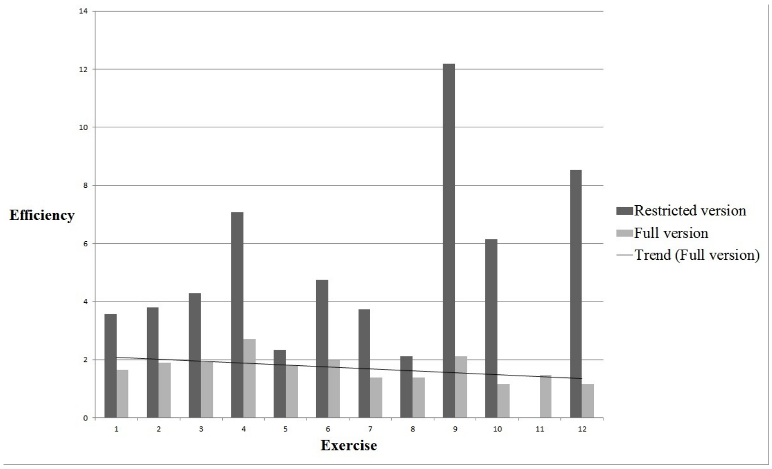


Figuur 3.6: Completion per exercise



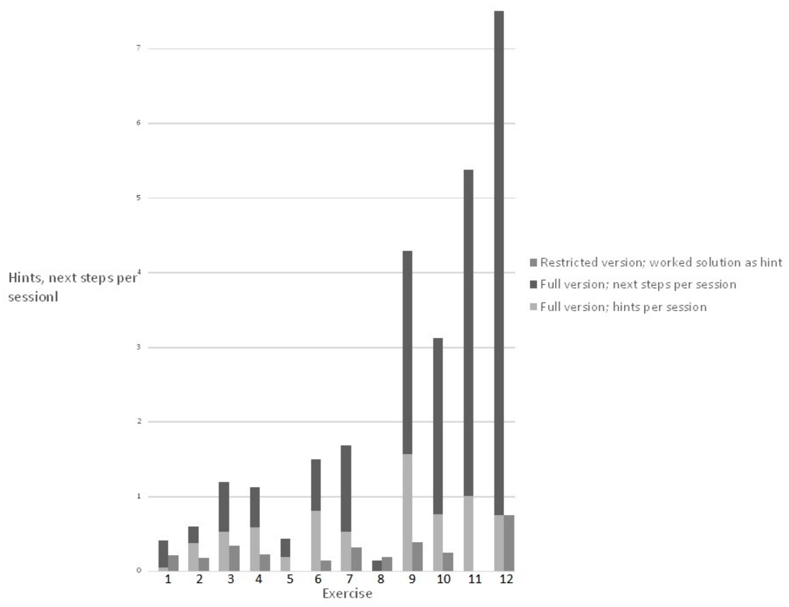
Figuur 3.7: Average step time per exercise

shows the efficiency per exercise. The linear regression trend line indicates that over time group F learns to solve the exercises slightly more efficient. Since the use of hints or worked solutions can influence the efficiency we also show the use of hints and next steps for group F and worked solutions for group R in Fig. 3.9 and Fig. 3.10. These figures suggest that the apparent progress in efficiency is in fact a direct result of the increased use of hints or worked solutions. These results are consistent with our findings in the pilot studies. We hypothesize that practicing with LogEx for 50 minutes is too short to learn an efficient solving procedure,

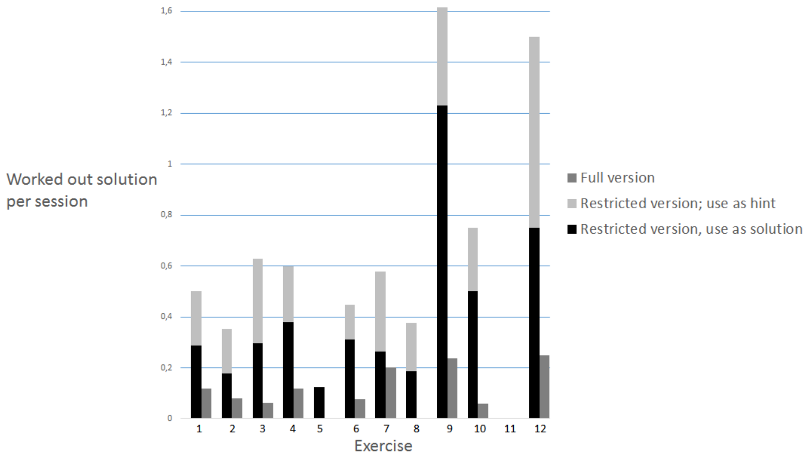


Figuur 3.8: Efficiency measured by the number of performed steps as a fraction of the number of steps in a worked solution per exercise

in particular since LogEx does not provide explicit strategic information. We expect that more and longer practice will help with the construction of efficient solution strategies. This is in line with findings from other studies, see Section 3.2. In the experiment with FOL, students practiced for one week, and the evaluation studies of Logic Tutor and Deep Thought took several weeks. All these studies found significant results. The experiment with the natural deduction LE took only one session of 80 minutes, and only showed a significant difference on the easy exercises (Miwa et al., 2014). Students in group R could ask for a complete solution at any moment. By counting the number of times that a worked out solution was directly followed by a student step we found that students use this solution as a hint on how to proceed in about one third of the cases, see Fig. 3.10. Figure 3.9 shows that although students in group R often use the complete solution to obtain a hint, they still ask much less help than the students in group F. Worked solutions also contain more information than a hint or a next step. Not only does a student receive all steps, but possibly also a clue about the usefulness of a next step. Further research is necessary to determine whether a student can indeed extract this kind of information from a solution. Further inspection of the loggings shows that some students in group F use the possibility to ask for a next step to obtain a worked solution. More than half of the next steps belong to sequences of next steps in which part of a worked solution is constructed. Another possible explanation for why students in group F ask for more help, is that they cannot proceed after a wrong step and ask the system for help in these situations. Interviews with students could show whether this is indeed the case, but the loggings already give an indication: a hint is asked twice as much after an incorrect step than after a correct step.



Figuur 3.9: Use of hints and next steps in group F compared to the hint use of worked examples in group R



Figuur 3.10: Use of worked examples in group R and F, group R divided in hint use and solution use

3.6 Conclusion and future work

We have performed an experiment in which we study the effect on student learning of using LogEx, a learning environment for proving the equivalence between two logical formulae using standard equivalences. Furthermore we compare different ways to give support in LogEx. The experiment indicates that LogEx can be a helpful LE for students who practice rewriting logical formulae. We conclude that students do learn to prove the equivalence of two formulae. The number of mistakes they make while working with LogEx decreases, but they do not exhibit improved strategic insight. The exam results (4.1 points out of 6 on average for students using the full LE and 3.5 out of 6 for students using the restricted version) provide additional support for the conclusion that students reach the first two learning goals. Further research is needed to find out whether practicing with LogEx for a longer time improves strategic insight. Another way to improve strategic insight could be to provide strategic feedback when a student solution is longer than necessary. The loggings show that especially students who practice with the full LE hardly use the possibility to ask for a worked solution. When students in this group finish an exercise successfully, they do not compare their solution with a possibly shorter example solution. Although in general students learn more when they have to ask for help themselves (VanLehn, 2006), in this case it might be necessary to let the system give help without being asked. Yet another way to improve LogEx could be by providing explicit strategic hints. LogEx recognizes when a student solution diverges from one of the possible paths determined by LogEx. In a next version, we might give a warning in such a case. In this way LogEx would exploit the fact that it generates proofs from a strategy, in contrast with data-driven or example-based tutors such as Deep Thought (Stamper et al., 2011a; Mostafavi and Barnes, 2017).

The extra features of the full version of LogEx: providing hints, next steps, and informative feedback after each step, do not have a significant effect on the exam results of students. Students using the full version perform slightly better, and on the exam this group performed significantly better than a control group of students who did not practice with the tool. In a next experiment we could measure the effects of informative timely feedback versus delayed feedback, and the effects of providing hints and next steps versus worked solutions separately. Since in both conditions in our experiment students could ask for a worked solution, they could use this solution as a hint. Therefore, the distinction between the two groups was less clear, with possibly negative effects on the significance of our results. The number of students in our experiment was too small to analyze whether there was a difference in effects on weak students or good students. This is also a question we would like to address in a follow-up study.

3.A Appendix

List of the exercises used in the experiment:

1. $\neg(p \wedge q) \vee s \vee \neg r \Leftrightarrow (p \wedge q) \rightarrow (r \rightarrow s)$
2. $p \wedge q \Leftrightarrow \neg(p \rightarrow \neg q)$
3. $(p \wedge q) \rightarrow p \Leftrightarrow \top$
4. $\neg(p \vee (\neg p \wedge q)) \Leftrightarrow \neg(p \vee q)$
5. $\neg(p \wedge (q \vee r)) \Leftrightarrow \neg p \vee (\neg q \wedge \neg r)$
6. $(p \rightarrow q) \vee (q \rightarrow p) \Leftrightarrow \top$
7. $\neg((p \rightarrow q) \rightarrow (p \wedge q)) \Leftrightarrow (p \rightarrow q) \wedge (\neg p \vee \neg q)$
8. $\neg(\neg p \wedge \neg(q \vee r)) \Leftrightarrow p \vee q \vee r$
9. $p \wedge (q \vee s) \Leftrightarrow (q \wedge \neg s \wedge p) \vee (p \wedge s)$
10. $(p \rightarrow q) \wedge (r \rightarrow q) \Leftrightarrow (p \vee r) \rightarrow q$
11. $(p \rightarrow \neg q) \rightarrow q \Leftrightarrow (s \vee (s \rightarrow (q \vee p))) \wedge q$
12. $p \rightarrow (q \rightarrow r) \Leftrightarrow (p \rightarrow q) \rightarrow (p \rightarrow r)$

4 Generation and Use of Hints and Feedback in a Hilbert-style Axiomatic Proof Tutor

4.1 Introduction

The ACM 2013 computer science curriculum lists the ability to construct formal proofs as one of the learning outcomes of a basic logic course (Association for Computing Machinery (ACM) and IEEE Computer Society Joint Task Force on Computing Curricula, 2013). The three main formal deductive systems are Hilbert systems, sequent calculus, and natural deduction. Natural deduction is probably the most popular system, but classical textbooks on mathematical logic usually also discuss Hilbert systems (Kelly, 1997; Mendelson, 2015; Enderton, 2001). Hilbert systems belong to the necessary foundation to the introduction of logics (temporary, Hoare, unity, fixpoint, and description logic) used in teaching of various fields of computer science (Varga and Várterész, 2006), and are treated in several textbooks on logic for computer science (Ben-Ari, 2012; Nievergelt, 2002; Arun-Kumar, 2002; Benthem et al., 2003). Hilbert systems are also taught in mathematics and logic programs (Leary and Kristiansen, 2015; Goldrei, 2005).

Students have problems with constructing formal proofs. An analysis of the high number of drop-outs in logic classes during a period of eight years shows that many students give up when formal proofs are introduced (Galafassi et al., 2015; Galafassi, 2012). Our own experience also shows that students have difficulties with formal proofs. We analyzed the homework handed in by 65 students who participated in the course "Logic and Computer Science" during the academic years 2014-2015 and 2015-2016. From these students, 22 had to redo their homework exercise on axiomatic proofs. This is significantly higher than, for example, the number of students in the same group who had to redo the exercise on semantic tableaux: 5 out of 65.

A student practices axiomatic proofs by solving exercises. Since it is not always possible to have a human tutor available, an intelligent tutoring system (ITS) might be of help. There are several ITSs supporting exercises on natural deduction systems (Sieg, 2007; Perkins, 2007; Broda et al., 2006). In these ITSs, students construct proofs and get hints and feedback. We found two e-learning tools that can be used by a student to practice the construction of axiomatic proofs: Metamath Solitaire (Megill, 2007) and Gateway to logic (Gottschall, 2012). Both tools are proof-editors: a student chooses an applicable rule and the system applies this rule automatically. These systems provide no help on how to construct a proof.

In this paper we describe LogAx, a new tool that helps students in constructing Hilbert-style axiomatic proofs. LogAx provides feedback, hints at different levels, next steps, and complete solutions. LogAx is part of a suite of tools assisting students in studying logic, such as a tool to practice rewriting formulae in disjunctive or conjunctive normal form, and to prove an equivalence using standard equivalences (Lodder et al., 2016, 2019).

The main contributions of this paper are:

- an algorithm for generating axiomatic proofs and dynamically extending partial proofs
- an extension of this algorithm to incorporate lemmas
- generating hints and feedback based on this algorithm
- the results of small-scale experiments with LogAx.

To determine the quality of LogAx, we compare the proofs generated by the tool with expert proofs and student solutions. We use the set of homework exercises mentioned above to collect common mistakes, which we have added as buggy rules (rules to provide informative feedback) to LogAx.

This paper is organized as follows. Section 4.2 describes Hilbert's axiom system and the way it is introduced in textbooks and Section 4.3 explains the interface of our e-learning tool LogAx. Section 4.4 introduces the algorithm to generate proofs automatically. Section 4.5 explains how we linearize these generated proofs and Section 4.6 how we add the possibility to use lemmas. Section 4.7 explains how we use the generated proofs for providing hints. This section also describes how we collect a set of buggy rules. Section 4.8 and Section 4.9 discuss the results of several evaluations of our work. We relate our work to existing approaches of generating solutions and hints in Section 5.3. Section 4.11 concludes and presents ideas for future work.

4.2 Teaching Hilbert-style axiomatic proofs

We start with a short description of Hilbert-style axiomatic proofs and the way they are introduced in different textbooks. Axiomatic proof systems come in several variants. The most common axiom systems are

$$\begin{array}{ll} \phi \to (\psi \to \phi) & \text{Axiom a} \\ (\phi \to (\psi \to \chi)) \to ((\phi \to \psi) \to (\phi \to \chi)) & \text{Axiom b} \\ (\neg\phi \to \neg\psi) \to (\psi \to \phi) & \text{Axiom c} \end{array}$$

used for example in Ben-Ari (2012); Nievergelt (2002); Benthem et al. (2003); Goldrei (2005); Kelly (1997), and the system consisting of Axiom a and b, but Axiom c' instead of Axiom c:

$$(\neg\phi \to \neg\psi) \to ((\neg\phi \to \psi) \to \phi) \qquad \text{Axiom c'}$$

used for example in Hirst and Hirst (2015); Arun-Kumar (2002); Wasilewska (2018); Mendelson (2015). These axioms are schemas that can be instantiated by replacing the metavariables ϕ, ψ and χ by concrete formulae. A proof consists of a list of statements of the form $\Sigma \vdash \phi$, where Σ is a set of formulae (assumptions) and ϕ is the formula that is derived from Σ. In a 'pure' axiomatic proof, each line is either an instantiation of an axiom, an assumption, or an application of the Modus Ponens (MP) rule:

$$\text{if } \Sigma \vdash \phi \text{ and } \Delta \vdash \phi \to \psi \text{ then } \Sigma \cup \Delta \vdash \psi$$

From these axioms and MP, the deduction theorem can be derived:

$$\text{if } \Sigma, \phi \vdash \psi \text{ then } \Sigma \vdash \phi \to \psi$$

The Open University of the Netherlands teaches axiomatic proofs in a bachelor course "Logic and computer science" and in a premaster program that prepares for admission to a master in computer science. The learning objective related to axiomatic proofs is:

– students are able to construct simple axiomatic proofs.

The course lectures start with recognizing instances of the axioms, and proceed with simple proofs, providing strategies such as:

– can you derive the last line of the proof by an application of the deduction theorem or Modus Ponens?
– how can you use the assumptions?

The textbooks we studied (Ben-Ari (2012); Nievergelt (2002); Benthem et al. (2003); Goldrei (2005); Hirst and Hirst (2015); Arun-Kumar (2002); Wasilewska (2018); Mendelson (2015)) do not give explicit learning goals, except for Kelly (1997), which starts each chapter with chapter aims. The aims of the chapter on axiomatic proofs are amongst others: "When you have completed your study of this chapter you should

– have a clear understanding of the structure of formal axiomatic systems
– be able to construct formal proofs of theorems".

From the other textbooks we can deduce learning goals from the examples and exercises. The textbooks all start the chapter on axiomatic proofs with introducing the axioms, followed by some examples and exercises in which a student has to construct simple proofs or provide the motivation to given proof lines. Some of these proofs use earlier results such as lemmas or derived rules. Some books start with the first two axioms (Wasilewska, 2018; Nievergelt, 2002) and introduce the negation axiom (Axiom c or c') after the deduction theorem, others introduce the deduction theorem after the three axioms. After the introduction of the deduction theorem exercises using it are presented. The exercises in these textbooks suggest that constructing proofs is a learning goal. The single exception is Wasilewska (2018): here most exercises only ask to motivate steps in an already constructed proof.

Hardly any textbook provides substantial information about how to construct a proof, except from providing examples and showing the use of the deduction theorem. Wasilewska (2018) explicitly states that constructing a proof may start with searching for two statements such that the conclusion is an application of Modus Ponens on these statements, and Kelly (1997) explains how to use the deduction theorem and gives a heuristic to derive $\Sigma \vdash \psi \to \phi$ from $\Sigma \vdash \phi$. Constructing proofs requires knowledge of the syntax of propositional logic, and competencies in rewriting logical formulae. Therefore, most textbooks deal with rewriting formulas using standard equivalences (Goldrei, 2005; Ben-Ari, 2012; Wasilewska, 2018; Arun-Kumar, 2002) or semantic tableaux (Kelly, 1997; Benthem et al., 2003; Ben-Ari, 2012), before the introduction of axiomatic proofs.

4.3 An e-learning tool for Hilbert-style axiomatic proofs

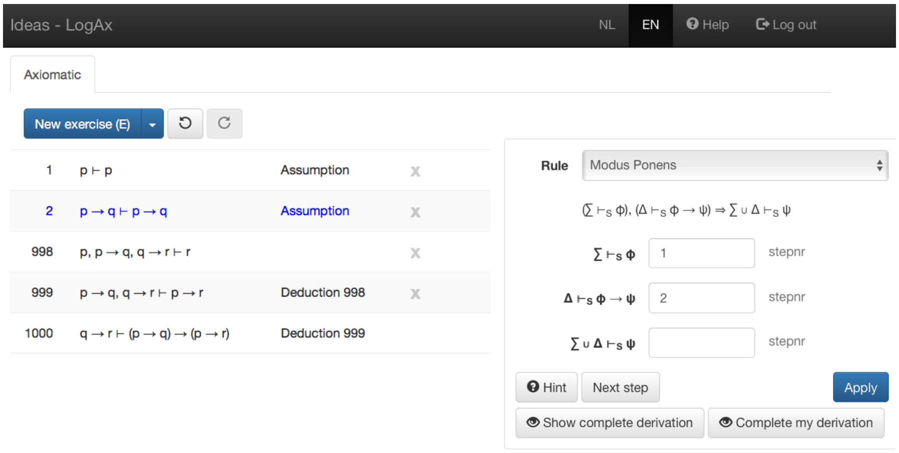
The e-learning tool that we developed, LogAx, uses the set of axioms a, b and c described in Section 4.2 and Modus Ponens and the deduction theorem.

A proof in this system can be constructed in two directions. To take a step in a proof, a student can ask two questions:

- How can I reach the conclusion?
- How can I use the assumptions?

An answer to the first question might be: use the deduction theorem to reach the conclusion. This answer creates a new goal to be reached, and adds a backward step to the proof. An answer to the second question might be: introduce an instance of an axiom that can be used together with an assumption in an application of Modus Ponens. This adds one or more forward steps. Fig. 4.1 shows an example of a partial proof, constructed in our tool LogAx. A full proof that completes this partial proof is:

1.	$p \vdash p$	Assumption
2.	$p \to q \vdash p \to q$	Assumption
3.	$p, p \to q \vdash q$	Modus Ponens, 1, 2
4.	$q \to r \vdash q \to r$	Assumption
5.	$p, p \to q, q \to r \vdash r$	Modus Ponens, 3, 4
6.	$p \to q, q \to r \vdash p \to r$	Deduction 5
7.	$q \to r \vdash (p \to q) \to (p \to r)$	Deduction 6



1	$p \vdash p$	Assumption	x
2	$p \to q \vdash p \to q$	Assumption	x
998	$p, p \to q, q \to r \vdash r$		x
999	$p \to q, q \to r \vdash p \to r$	Deduction 998	x
1000	$q \to r \vdash (p \to q) \to (p \to r)$	Deduction 999	

Rule: Modus Ponens

$(\Sigma \vdash_S \phi), (\Delta \vdash_S \phi \to \psi) \Rightarrow \Sigma \cup \Delta \vdash_S \psi$

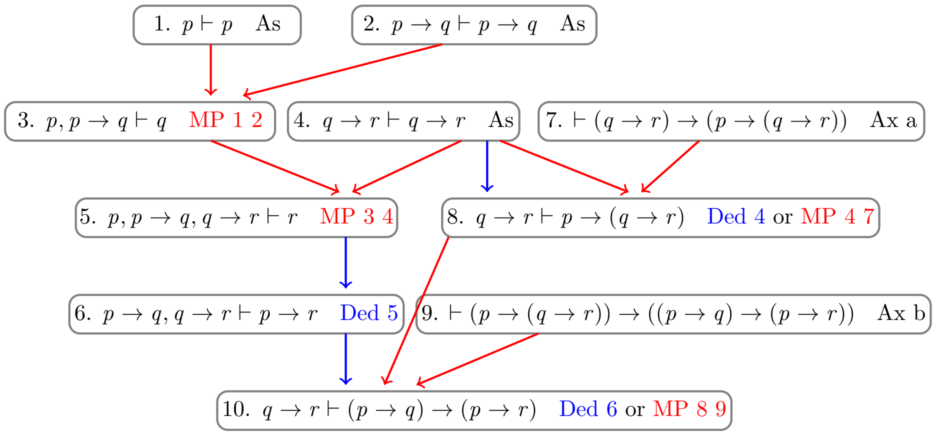
$\Sigma \vdash_S \phi$	1	stepnr
$\Delta \vdash_S \phi \to \psi$	2	stepnr
$\Sigma \cup \Delta \vdash_S \psi$		stepnr

Figuur 4.1: A partial proof of $q \to r \vdash (p \to q) \to (p \to r)$ performed in LogAx. On the right is the dialog box, in which a student can choose rules and fill in step numbers and help buttons below this dialog box. On the left is the proof as presented by LogAx.

Fig. 4.1 illustrates most of the functionality of our e-learning tool LogAx. A student starts with choosing a new exercise from the list, or formulating her own exercise. She continues working in the dialog box to add new proof lines. Here she can first choose which rule to apply: an assumption, axiom, an application of Modus Ponens or deduction theorem, or a new goal. In case of an assumption she enters a formula, and in case of an axiom, LogAx asks for parameters to add the instantiation of the axiom to the proof. Fig. 4.1 shows adding a Modus Ponens: a student has to fill in at least two of the three line numbers. LogAx performs a step automatically and adds a forward or backward step to the proof. In the same way, a student provides a line number to perform a backward application of the deduction theorem. If the deduction theorem is applied in a forward step, the student also provides a formula ϕ. The new goal option can be used to formulate a subgoal to be reached.

If a student makes a mistake, e.g. she writes a syntactical error in a formula, or tries to perform an impossible application of Modus Ponens, the tool provides immediate feedback. At any moment she can ask for a hint, next step, or a complete proof. The high number labelling the target statement (1000) is chosen deliberately, because at the start of the proof it is not yet clear how long the proof will be. After finishing the proof a student can ask the tool to renumber the complete proof.

As described above, LogAx contains a dialog box to add new proof lines. The idea behind this approach is that a student can concentrate on proof construction. The design choice to allow a student to choose a rule and let the software


1. $p \vdash p$ As
2. $p \to q \vdash p \to q$ As
3. $p, p \to q \vdash q$ MP 1 2
4. $q \to r \vdash q \to r$ As
7. $\vdash (q \to r) \to (p \to (q \to r))$ Ax a
5. $p, p \to q, q \to r \vdash r$ MP 3 4
8. $q \to r \vdash p \to (q \to r)$ Ded 4 or MP 4 7
6. $p \to q, q \to r \vdash p \to r$ Ded 5
9. $\vdash (p \to (q \to r)) \to ((p \to q) \to (p \to r))$ Ax b
10. $q \to r \vdash (p \to q) \to (p \to r)$ Ded 6 or MP 8 9

Figuur 4.2: A DAM for the proof of $q \to r \vdash (p \to q) \to (p \to r)$

perform the rule has successfully been applied in several e-learning tools for logic and mathematics (Mostafavi and Barnes, 2016; Beeson, 1998; Robson et al., 2012). For instance, Robson et al. (2012) state that their interface “allows students to concentrate on strategies while the software carries out procedures”. The use of the dialog box also implies that students can make fewer mistakes. Since students should know in principle how to write correct syntactical formulae, we hope that by using the dialog box, students spend less time on correcting parentheses in long formulae, such as for example instances of Axiom b. The only possible oversights students still can make are syntax mistakes in smaller formulae that need to be entered when adding for example an instance of an axiom to the proof. The evaluation in Section 4.9 shows that students indeed make very few syntactical mistakes.

4.4 An algorithm for generating proof graphs

An ITS for axiomatic proofs provides hints and feedback. There are at least two ways to construct hints and feedback for a proof. First, they can be obtained from a complete proof. Such a proof can either be supplied by a teacher or an expert, or deduced from a set of student solutions. An example of an ITS for natural deduction proofs that uses student solutions has been developed by Mostafavi and Barnes (2017). A drawback of this approach is that the tool only recognizes solutions that are more or less equal to the stored proofs. The tool cannot provide hints when a student solution diverges from these stored proofs. Also, this only works for a fixed set of exercises. If a teacher wants to add a new exercise, she also has to provide solutions, and the tool cannot give hints for exercises that are defined by a student herself. The second way to provide the tool with solutions, which we use,

$$\neg_{\,elim}\ \frac{\neg\neg\phi}{\phi} \qquad \neg_{\,intro}\ \frac{\begin{matrix}\cancel{\psi} & \cancel{\psi}\\ \vdots & \vdots\\ \phi & \neg\phi\end{matrix}}{\neg\psi} \qquad \to_{\,elim}\ \frac{\phi\to\psi \quad \phi}{\psi} \qquad \to_{\,intro}\ \frac{\begin{matrix}\cancel{\phi}\\ \vdots\\ \psi\end{matrix}}{\phi\to\psi}$$

Figuur 4.3: Rules for natural deduction

is to create proofs automatically. At first sight this might only solve the second problem: automatically providing hints for new exercises. Section 4.5 explains how our approach makes it possible to provide hints also in case a student diverges from a model solution.

We develop an algorithm that automatically generates proofs. This algorithm should generate the kind of proofs we expect from our students. Existing algorithms, such as the Kalmár constructive completeness proof (Kalmár, 1935), or the algorithms used in automatic theorem proving (Harrison, 2009), are unsuitable for this purpose. Natural deduction tools such as ProofLab (Sieg, 2007) and Pandora (Broda et al., 2006) also use algorithms to calculate solutions, and these algorithms can provide useful hints and feedback. We adapt an existing algorithm for natural deduction to create axiomatic proofs. Before we describe the algorithm, we first explain how we represent proofs.

Fig. 4.1 shows a partial example proof of $q \to r \vdash (p \to q) \to (p \to r)$. There are alternative ways to start this proof. A student may choose between various orders, for example swap line number 1 and line number 2. Using one or more axiom instances we may obtain entirely different proofs. Since we want to recognize different proofs, we represent proofs as labeled directed acyclic multi graphs (DAM), where the vertices are statements $\Sigma \vdash \phi$ and the edges connect dependent statements. We annotate vertices with the applied rule: Assumption, Axiom, Modus Ponens or Deduction. Note that a statement can be the result of different applications of rules. An example of such a DAM is shown in Fig. 4.2. Vertices are numbered for readability. A blue arrow means that the lower statement follows from the higher by application of the deduction theorem. A pair of red arrows represents an application of Modus Ponens. This DAM contains three essentially different proofs: one that uses Axiom a and b, one that applies the deduction theorem and Axiom a, and one that uses no axioms and applies the deduction theorem twice. This last proof is a continuation of the proof provided in Fig. 4.1.

The basis for our algorithm for axiomatic proofs is Bolotov's algorithm for natural deduction proofs (Bolotov et al., 2005). The rules used in this system are presented in Fig. 4.3, restricted to the connectives $\neg$ and $\to$ since these are the only connectives used in the Hilbert axiomatic system. Here we use the same notation as presented in Benthem et al. (2003). A natural deduction proof is here presented as a tree-like structure. The elimination rules ($\neg_{\,elim}$ and $\to_{\,elim}$) say that you can extend a proof of $\neg\neg\phi$ with ϕ and combine subproofs of $\phi \to \psi$ and ϕ into a proof of ψ. The introduction rules discard assumptions: subproofs of ϕ and $\neg\phi$ can be combined in a proof of $\neg\psi$ by an application of rule $\neg_{intro}$ while discarding ψ. The last rule, $\to_{intro}$, says that if you have a proof of ψ, you can add $\phi \to \psi$ and discard ϕ. The natural deduction rules for implication translate directly to rules in the Hilbert system: $\to_{elim}$ corresponds to Modus Ponens and $\to_{intro}$ to the deduction theorem. The rules for negation do not have direct counterparts in the axiomatic system. Therefore, the first adaptation that we have to make to Bolotov's algorithm is the use of axiomatic subproofs that mimic the natural deduction rules for negation. The $\neg_{elim}$ rule is translated to a single subproof, and we use seven different subproofs to translate the $\neg_{intro}$ rule, mainly to cover the possible different dependencies from ϕ and $\neg\phi$ on ψ.

The Bolotov algorithm is goal-driven, and uses a stack of goals. We build a DAM using steps that are divided into five groups. The first group contains a single step to initialize the algorithm. The steps in the second group check whether or not a goal is reached. The steps in the third group extend the DAM. The steps in group 4 handle the goals and may add new formulae to the DAM. In this group, a goal F can be added. The symbol F is not part of the language, but we use F as shorthand for "prove a contradiction". Finally, group 5 completes the algorithm, where we omit certain details for the steps that are needed to prevent the algorithm from looping.

1. We start the algorithm by adding the target statement (e.g. $q \to r \vdash (p \to q) \to (p \to r)$) to our stack of goals, and the assumptions of this goal ($q \to r \vdash q \to r$) to the DAM.

Until the stack of goals is empty, repeat:

2. a) If the top of the stack of goals (the top goal from now on) belongs to the DAM, we remove this goal from the stack of goals.

   Motivation: the goal is reached.

   b) If the top goal is $\Delta \vdash F$ and the DAM contains the statements $\Delta' \vdash \phi$ and $\Delta'' \vdash \neg\phi$ such that $\Delta' \cup \Delta'' \subseteq \Delta$, we add a set of axioms to the DAM that can be used to prove the goal below the top from these two statements. We remove the goal $\Delta \vdash F$ from the stack.

   Motivation: we can use the contradiction to prove the goal below the top. Apart from the instances of the axioms, this proof will use applications of Modus Ponens. Hence, the goal below the top will be removed in a later step.

3. a) If the DAM contains a formula $\Delta \vdash \neg\neg\phi$, we add an instance of Axiom a ($\vdash \neg\neg\phi \to (\neg\neg\neg\neg\,\phi \to \neg\neg\phi)$) and two instances of Axiom c to the DAM. The next step uses these axioms to deduce $\Delta \vdash \phi$.

   Motivation: use the doubly negated formula.

b) We close the DAM under applications of Modus Ponens.

Motivation: here we perform a broad search, and any derivable statement will be added to the DAM.

c) If the DAM contains a formula $\Delta \vdash \psi$ and the top goal is $\Delta \setminus \phi \vdash \phi \to \psi$, we add $\Delta \setminus \phi \vdash \phi \to \psi$ to the DAM.

Motivation: use the deduction theorem.

4. a) If the top goal is $\Delta \vdash \phi \to \psi$, we add $\phi \vdash \phi$ to the DAM and the goal $\Delta, \phi \vdash \psi$ to our stack of goals.

   Motivation: prove $\Delta \vdash \phi \to \psi$ with the deduction theorem.

   b) If the goal is $\Delta \vdash \neg\phi$ we add $\phi \vdash \phi$ to the DAM and the goal $\Delta, \phi \vdash F$ to our stack of goals.

   Motivation: prove $\Delta \vdash \neg\phi$ by contradiction.

   c) If the goal is $\Delta \vdash p$, where p is an atomic formula, we add $\neg p \vdash \neg p$ to the DAM and the goal $\Delta, \neg p \vdash F$ to our stack of goals.

   Motivation: we cannot prove $\Delta \vdash p$ directly, and hence we prove it by contradiction.

5. a) If the top goal is $\Delta \vdash F$ and $\Delta \vdash \phi \to \psi$ belongs to the DAM, we add $\Delta \vdash \phi$ to our stack of goals.

   Motivation: we cannot prove a contradiction with the steps performed thus far. Hence, we exploit the statements we already have. Since our goal is to prove $\Delta \vdash F$, any formula is provable from Δ.

   b) If the top goal is $\Delta \vdash F$ and $\Delta \vdash \neg\phi$ belongs to the DAM we add $\Delta \vdash \phi$ to our stack of goals.

   Motivation: use derived statements.

This algorithm constructs a basic DAM. Bolotov shows that his algorithm is sound and complete. Our adaptations, as for instance the replacement of a negation introduction rule by a set of instances of axioms, preserve soundness and completeness. We omit a detailed description of our adaptations and a proof of the correctness.

The above algorithm only uses axioms in a proof of a contradiction, or in the use of double negations. This means that without extra adaptations, Axiom b will never be used in a generated proof. Since we want the constructed proofs to resemble the proofs constructed by experts or students, and since LogAx should teach our students to recognize the possibility to use axioms, we use extra heuristic rules to add more instances of axioms to the DAM. With these heuristics we can produce the example DAM in Fig. 4.2. The heuristics to produce the right branch consisting of the nodes 4, 7, 8, 9 and 10 of the DAM are:

- If the top goal equals $\Delta \vdash (\phi \to \psi) \to (\phi \to \chi)$ and $\Delta' \vdash \psi \to \chi$ already belongs to the DAM and $\Delta' \subseteq \Delta$, then add an instance $(\phi \to (\psi \to \chi)) \to ((\phi \to \psi) \to (\phi \to \chi))$ of Axiom b to the DAM.
- If the top goal equals $\Delta \vdash \phi$ and $\Delta' \vdash (\psi \to \chi) \to \phi$ and $\Delta'' \vdash \chi$ belong to the DAM with $\Delta' \subseteq \Delta$ and $\Delta'' \subseteq \Delta$, then add an instance $\chi \to (\psi \to \chi)$ of Axiom a to the DAM.

4.5 Distilling proofs for students

In the previous section we described the algorithm used to construct the DAM. Such a DAM may contain different solutions. For example, Fig. 4.2 shows three essentially different solutions for the proof of $q \to r \vdash (p \to q) \to (p \to r)$. Since we use this DAM to generate proofs for the purpose of giving hints to students or providing sample solutions, we have to find a way to isolate single proofs. Moreover, the proofs in the DAM are structured directed acyclic graphs, whereas an axiomatic proof is a linear structure. Hence, we need a procedure to extract linear proofs from a DAM. We will not only use this procedure to provide complete solutions, but also to generate next steps and hints, which means that the procedure should meet the following requirements:

- R1: generate a complete linear proof at once or stepwise
- R2: complete a partial proof, even if this proof diverges from the generated linear proof or contains a user-defined goal
- R3: add steps to a proof in an order that corresponds to the way students or experts add steps.

Requirement R1 is a direct consequence of our goal to use the DAM to provide sample solutions, hints and next steps. Since a student solution may differ from the sample solution constructed from the DAM, we need requirement R2 to ensure that LogAx can always provide a hint or a next step, using the procedure to complete a partial proof. There are two ways in which the order of the steps while constructing a proof may vary. To illustrate the first way, we look at the example proof in Section 4.3. We could construct this proof in a forward way, from top to bottom, starting with line number 1 and finishing with line number 7. However, most textbooks advise to apply the deduction theorem backwards. Hence we prefer a solution that starts with line number 6 and 7. A second way in which the order of the steps may vary is the order of the lines in the completed proof. Take for example the proof in Section 4.3 again. This proof might also start with line number 4. The reason behind the order chosen is that in general, assumptions are introduced for use in an application of Modus Ponens, and in general students and experts introduce assumptions and axioms only when they can be used directly. If a student asks for a hint, we want LogAx to provide the step that would be advised by an expert or a fellow student, hence R3 requires an order of the steps corresponding to the way students or experts add steps. In the rest of this section we will first explain how we extract linear proofs and motivate why this way of extracting proofs matches the requirements later in this section.

The correctness of the algorithm defined in Section 4.4 ensures that the DAM contains a complete proof. Extracting a single proof can be seen as searching for a subtree. Linearization of this subtree requires topological sorting. Since generating a stepwise solution is one of the requirements, we perform these two tasks, extracting and linearization, simultaneously.

The procedure for proof extraction consists of four different kinds of steps, which are repeated until the linearized proof is complete. In each step a new line is added to the proof under construction, or an unmotivated line is motivated. In the following list, the different steps are ordered according to the preference in which a certain step is chosen:

- a close step: add a motivation to an unmotivated proof line
- a backward step: add a backward application of the deduction theorem to a proof
- a forward step: add a forward application of deduction or modus ponens to a proof
- an introduction step: add an assumption or axiom to a proof.

While performing these steps, the procedure keeps track of the partial linearized proof, which consists of the grounded part (the already proven proof lines), and the ungrounded part. The latter part consists of lines that are already in the linearized proof, but are either unmotivated, or their motivation depends on unmotivated proof lines. The unmotivated lines are part of a list of goals, which may also contain a set of other subgoals to be reached.

If possible, the procedure performs a close step, since in general this step completes the proof. The next preferred step is a backward step since a backward application of the deduction theorem replaces the goal to be reached by a simpler goal. When applications of the deduction theorem are impossible, the procedure tries to use already proven lines in an application of a forward step, and only if all other three kinds of steps are impossible, the procedure introduces an assumption or an axiom. Here we have to take care of the logical structure of the proof. We illustrate this by means of the example in Fig. 4.2. Suppose that the partial proof consists of nodes 6 and 10 in Fig. 4.2, which means that deduction was applied to node 10. Continuing with node number 7 would add a superfluous line to the proof. To prevent this, the procedure trims the DAM into a subDAM using the first goal of the list of subgoals as a root. In our example, node number 6 becomes the new root, and the leaves in this subDAM are the nodes 1, 2 or 4. Suppose the procedure continues with node number 1. That leaves two possibilities for the next step, namely node number 2 or 4. Because of the last requirement R3, node number 2 is preferred, since in general students or experts choose an assumption that can be used directly over an assumption that can only be used later. The procedure realizes this preference by adding subgoals to the list of subgoals after performing an introduction step. These subgoals consist of the nodes between the introduced leaf and the node that corresponds to a subgoal already in the list of subgoals. In our example, the line numbers 3 and 5 are added as subgoals, which forces the procedure to look for a next leaf in the subtree rooted by line number 3 in the next step. As a consequence, line number 2 is indeed added in this step.

We claim that this procedure meets the three requirements given above. Requirement R1, generating complete proofs, is guaranteed by the construction of the DAM. To show that requirement R2 is met, we distinguish two situations. As long as the steps in the student solution correspond to the steps generated by the procedure described in this section, this proof can be completed directly. If the student solution diverges from the generated solution, LogAx will use the student solution as a starting point to build a new DAM. Motivated lines will be marked as grounded lines and unmotivated lines will be part of the list of goals. This ensures that the procedure indeed extends the partial proof into a complete proof.

From student solutions to exercises and the loggings of LogAx we know that students can perform the steps of a proof in many different orders. However, there are some heuristics in the construction of a proof, such as trying to use assumptions, or simplifying the goal by applying the deduction theorem. The preference on the order of the steps in our procedure ensures that the procedure follows these heuristics (requirement R3). This implies that steps can be added in two directions, forward and backward, and that a user can switch direction at any moment. Moreover, the order of the steps should be such that we can always motivate the next line: why do we perform a certain step at a certain moment. To achieve this, we use a dynamic programming approach, where subproblems are defined by the list of subgoals. The restriction to subDAMs as described above, ensures that we complete the subproblem defined by the first node of this list before we start a new subproblem. All steps can thus be motivated by a subgoal.

4.6 Lemmas

Reusing proven results is common practice in mathematics and logic. For example, the proof of the fundamental theorem of arithmetic (every number larger than 1 can be written in a unique way as a product of primes) uses the lemma that a prime divisor of a product $a \cdot b$ is also a divisor of a or b. In logic the use of proven results is widespread too. Here, proven results are sometimes presented as derived rules, such as for instance the rule Modes Tollens ($\neg\phi$ can be derived from $\phi \to \psi$ and $\neg\psi$) in Huth and Ryan's textbook (Huth and Ryan, 2004). Axiomatic proofs often build on each other: for example, a proof of $\vdash \neg\neg p \to p$ can be used as a lemma in another proof.

Lemmas appear in several ITSs that deal with constructing proofs. They serve various purposes, such as a starting set to generate geometry problems (Alvin et al., 2014), or just as a predefined set that can be used by the student to solve a problem (Matsuda and VanLehn, 2005). Perhaps more interesting is the possibility to allow the addition of lemmas by the user. In both the Jape natural deduction proof assistant (Bornat, 2017) and the proof assistant described by Aguilera et al. (2000), a student can save proven results and use these results as lemmas in a new proof. The proof assistant Gateway to Logic for axiomatic proofs offers a user the possibility to state and use lemmas too (Gottschall, 2012).

Lemmas in axiomatic proofs have various shapes. For example, we distinguish tautologies ($\vdash \phi$) and valid sequents ($\phi_1, ...\phi_n \vdash \psi$), but also schemas (for example $\neg\neg\phi \vdash \phi$) and instantiations of schemas ($\neg\neg p \vdash p$). We include the use of lemmas in LogAx. The main purpose of including lemmas is to support adding relatively easy exercises. Without lemmas, many axiomatic proofs are too lengthy and complicated to be used in education. With the possibility to use lemmas, a new class of relatively easy exercises becomes available. The second goal is to give users the possibility to use their own lemmas. LogAx can provide a student with an exercise together with a lemma that may be used in the proof, and in a user-defined exercise the user can use her own lemmas. We impose two restrictions: predefined exercises use only instantiations of lemmas, and the interface only accepts user-defined lemmas that are instantiations of a tautology. The latter is not a real restriction since a valid sequent $\phi_1, ...\phi_n \vdash \psi$ can always be rewritten as a tautology $\vdash (\phi_1 \rightarrow (... \rightarrow (\phi_n \rightarrow \psi)))$.

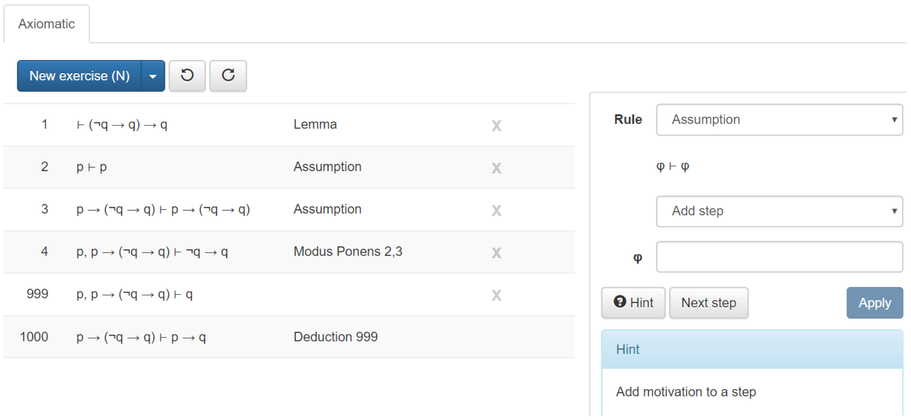
We adapt the algorithm to create a DAM to support the use of lemmas. In predefined exercises, lemmas are added to the DAM at the start of the algorithm, comparable to the addition of assumptions. The construction of the DAM and the extraction of a linear proof work in the same way as described in Section 4.4 and 4.5. Students who solve these predefined exercises receive the lemma as a first line of the proof. To facilitate user-defined lemmas, a lemma rule is added to the set of rules. In a user-defined exercise a student can introduce a lemma at any stage during the proof. The algorithm constructs a DAM based on the partial proof including the lemma and uses this DAM to provide hints and feedback.

Fig. 4.4 shows an example of an exercise with a lemma. Since a motivation of line 999 as an application of Modus Ponens to the lemma in line 1 and line 4 completes the proof, the hint tells the student to add a motivation.

4.7 Hints and feedback

4.7.1 Hints

One of the reasons for the effectiveness of human tutors is that they provide feedback at the level of solution steps, and help a student to overcome impasses using hints (Merrill et al., 1992). Hence, for ITSs to be as effective as human tutors,



1	$\vdash (\neg q \to q) \to q$	Lemma	x
2	$p \vdash p$	Assumption	x
3	$p \to (\neg q \to q) \vdash p \to (\neg q \to q)$	Assumption	x
4	$p, p \to (\neg q \to q) \vdash \neg q \to q$	Modus Ponens 2,3	x
999	$p, p \to (\neg q \to q) \vdash q$		x
1000	$p \to (\neg q \to q) \vdash p \to q$	Deduction 999	

Rule: Assumption

$\varphi \vdash \varphi$

Add step

φ

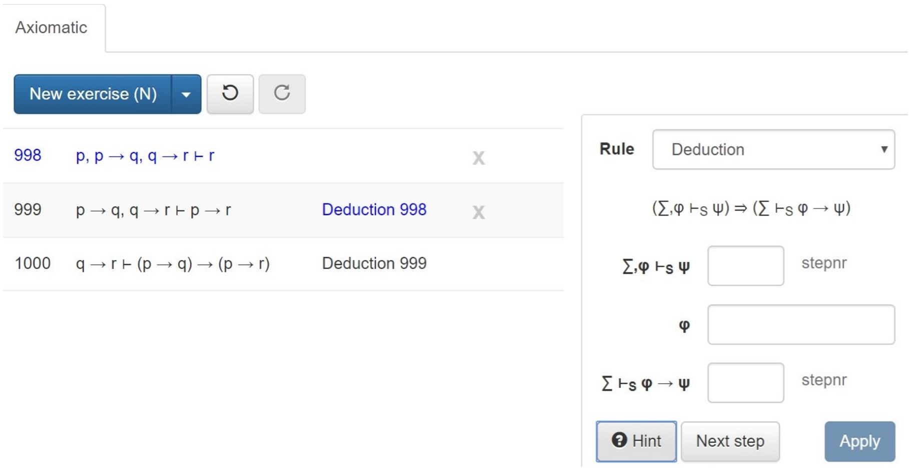
Hint

Add motivation to a step

Figuur 4.4: A partial proof with a lemma, performed in LogAx

they should give stepwise feedback and some form of help. In a first version of LogAx we implemented a hint sequence consisting of three hint types: the direction of a next step (forward or backward), the axiom or rule to apply, and a bottom out hint that shows how to perform a next step. Although these hints can help students to complete a proof, they might not always help a student to understand why a certain step is useful. A study with the Geometry Tutor (McKendree, 1990) shows that students who receive informative feedback combined with information about a subgoal are more effective in correcting mistakes than students who only receive informative feedback. Several logic tutors offer hints containing subgoals. An early attempt is the P-logic tutor (Lukins et al., 2002), in which students learn to construct proofs using standard equivalences and inference rules. Since this tutor cannot construct proofs, it uses heuristics to construct possible useful subgoals, such as an atomic formula from which the truth can be deduced. A drawback of this approach is that the tutor might suggest an unnecessary subgoal. The Deep Thought Logic tutor (Eagle et al., 2012; Barnes and Stamper, 2008) uses datamining to construct proofs and subgoal hints from student solutions. In a comparison of the performance of students receiving next step hints with students receiving hints about a subgoal to be reached in two or three steps, the latter group outperformed the former one in the more difficult exercises both with respect to the time needed to take a step as well as accuracy (Cody et al., 2018).

In LogAx we keep track of a list of subgoals while constructing a proof. We use this list to provide hints about a subgoal. We do not give a subgoal as hint if a student can still apply deduction backwards, when a subgoal coincides with an unmotivated line in the proof, or when a subgoal coincides with the next step. In the other cases, we give a hint concerning a subgoal instead of a hint about the direction of the proof. For instance, the hint for the unfinished proof in Fig. 4.5



998	p, p → q, q → r ⊢ r		x
999	p → q, q → r ⊢ p → r	Deduction 998	x
1000	q → r ⊢ (p → q) → (p → r)	Deduction 999	

Rule: Deduction

(Σ,φ ⊢S ψ) ⇒ (Σ ⊢S φ → ψ)

Σ,φ ⊢S ψ — stepnr

φ

Σ ⊢S φ → ψ — stepnr

Figuur 4.5: The start of a proof for $q \to r \vdash (p \to q) \to (p \to r)$

will be: try to prove $p, p \to q \vdash q$. An example where our algorithm deliberately not gives a subgoal as hint can be found in the proof in Fig. 4.1. Here, he first hint will be: perform a forward step, since in this case the subgoal $p, q \to q \vdash q$, Modus Ponens, is equal to the next step.

4.7.2 Feedback

In this subsection we first analyze student errors, and describe than how we use this analysis to create feedback. Students make mistakes in axiomatic proofs. From the homework of 40 students participating in our course we collected a set of mistakes, and classified these mistakes in three categories:

- oversights (19),
- conceptual errors (11), and
- ‘creative’ rule adaptations (9).

Mistakes such as missing parentheses belong to the first category. This category mainly consists of missing parentheses in Axiom b. A typical example of a mistake in the second category is the following application of Modus Ponens:

1.	$\neg p, \neg q \vdash \neg q$	Assumption
2.	$\neg q \vdash \neg p \to \neg q$	Deduction 2
3.	$\vdash (\neg p \to \neg q) \to (q \to p)$	Axiom c
4.	$\neg p \to \neg q \vdash q \to p$	Modus Ponens 2, 3

Here, the student has the (wrong) idea, that after an application of Modus Ponens on $\Delta \vdash \phi$ and $\Sigma \vdash \phi \to \psi$, the formula ϕ becomes the assumption of the conclusion.

Creative rule adaptations may take various forms. An example of such a rule adaptation is:

1.	$\neg p \rightarrow (q \rightarrow \neg r) \vdash \neg p \rightarrow (q \rightarrow \neg r)$	Assumption
2.	$q \vdash q$	Assumption
3.	$\neg p \rightarrow (q \rightarrow \neg r), q \vdash \neg p \rightarrow \neg r$	Modus Ponens 1, 2

In this example the ultimate goal is to prove that $\neg p \rightarrow (q \rightarrow \neg r), q \vdash r \rightarrow p$. The student tries to reach this via the subgoal $\neg p \rightarrow (q \rightarrow \neg r), q \vdash \neg p \rightarrow \neg r$, but she misses the possibility to reach this subgoal with an instantiation of Axiom b and Axiom a. Instead, she creates her own variant of Modus Ponens in line 3 of the proof.

Further analysis of the homework exercises suggests that students typically make these mistakes when they do not know how to proceed. This is in line with the repair theory, which describes the actions of students when they reach an impasse (Brown and VanLehn, 1980).

The example of a conceptual error given above is impossible to construct in LogAx, since LogAx fills in the assumptions automatically. In the evaluation in Section 5.8 we analyze whether students recognize these kinds of conceptual mistakes after practicing with LogAx. However, it is still possible that a student tries to apply a rule incorrectly. For example, a student might apply Modus Ponens on $\Delta \vdash \phi$ and $\Sigma \vdash \phi' \rightarrow \psi$ where ϕ and ϕ' are equivalent but not equal. We used homework solutions to define a set of buggy rules for mistakes that can be made in LogAx. Most of these rules relate to Modus Ponens (8 buggy forward applications, 3 buggy backward applications and 2 closure rules), the other to Deduction (2 buggy backward applications and 4 buggy closure rules). Using these rules, LogAx can give informative feedback. Shute's guidelines (Shute, 2008) state that feedback should be elaborate, specific, clear, and as simple as possible. Our feedback not only points out a mistake, but if possible also mentions exactly which formula, subformula or set of formulae do not match with the rule chosen. For example, if a student wants to complete a proof

1.	$\neg q, \neg p \vdash \neg q$	Assumption
2.	$\neg q \vdash \neg p \rightarrow \neg q$	Deduction 1
3.	$\vdash (\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$	Axiom c
4.	$\neg p \rightarrow \neg q \vdash q \rightarrow p$	

by applying Modus Ponens to lines 2, 3 and 4, she gets a message that line 4 cannot be the result of an application of Modus Ponens on lines 2 and 3, since the assumption of line 2 does not belong to the set of assumptions in line 4.

Tabel 4.1: Exercises without lemmas in textbooks and lecture notes. The last column compares thet LogAx proof with the expert proof

	Exercise	Textbook	Equal
1.	$q, p \to (q \to r) \vdash p \to r$	Kelly, LenI	yes
2.	$p \to q, q \to r \vdash p \to r$	Kelly, LenI	yes
3.	$p \to q \vdash p \to (r \to q)$	Kelly	yes
4.	$p \to q \vdash (r \to p) \to (r \to q)$	LenI	yes
5.	$\vdash \neg p \to (\neg\neg p \to p)$	LenI	no
6.	$\vdash ((p \to q) \to (p \to r)) \to (p \to (q \to r))$	LenI	no
7.	$\vdash (p \to q) \to ((q \to r) \to (p \to r))$	Ben-Ari	yes
8.	$\vdash (p \to (q \to r)) \to (q \to (p \to r))$	Ben-Ari	yes
9.	$\vdash \neg p \to (p \to q)$	Ben-Ari, Goldrei	yes
10.	$\vdash \neg\neg p \to p$	Ben-Ari	yes

4.8 Evaluation of the generated proofs

We first evaluate the proofs generated by LogAx by comparing them with expert proofs. In the next subsection we evaluate the recognition of student solutions by LogAx. The next section describes a small-scale experiment with students using LogAx.

4.8.1 Comparison of the generated proofs with expert proofs

We evaluate the proofs generated by LogAx in two ways. First, we compare the generated proofs with expert proofs. Since example proofs and worked solutions in textbooks often use earlier proofs as a lemma, the number of proofs that we can compare with the LogAx proofs without lemmas is small. We found 10 examples we could use for a comparison in the textbooks of Ben-Ari, Kelly, Goldrei and in the lecture notes of a course on logic (LenI) (Lodder et al., 2018). The list of exercises can be found in Table 4.1. Eight of the ten expert proofs are equal to the LogAx proofs. In exercise 6, LogAx uses the deduction theorem instead of Axiom a. The LogAx proof of $\vdash \neg p \to (\neg\neg p \to p)$ (exercise 5) is longer than the expert proof, since LogAx proves $\vdash \neg\neg p \to p$ directly without making use of an assumption $\neg p$. We conclude that for most of the examples and exercises in textbooks, LogAx generates a proof that is equal to the expert proof.

To evaluate the version of LogAx with lemmas, we have nine expert proofs available. The course on logic contains five examples of exercises using lemmas, together with example proofs of these exercises, see Table 4.2. In the first three exercises only instantiated lemmas are given. The proofs generated by LogAx for these exercises are equal to the solutions in the course notes. The fourth and fifth exercise ask for a proof in predicate logic, but we can compare the propositional part of these proofs. Both exercises present lemmas as schemas, and instantiations of these schemas are used in the solution. We add these instantiations as lemmas in LogAx. The solution to the fifth exercise is equal to the proof generated by LogAx, except for the order of the proof lines. In the fourth exercise, LogAx

Tabel 4.2: Exercises in a logic course

Textbook	Exercise	Lemma	Equal
LenI	$\vdash (p \to q) \to (\neg q \to \neg p)$	$\vdash (p \to q) \to (\neg\neg p \to \neg\neg q)$	yes
LenI	$\neg\neg p \vdash \neg p \to \neg\neg\neg\, p$	$\neg\neg p \vdash \neg\neg\neg\neg\, p \to \neg\neg p$	yes
LenI	$\vdash \neg\neg p \to p$	$\neg\neg p \vdash \neg p \to \neg\neg\neg\, p$	yes
LenI	$\vdash \neg(p \to q) \to (\neg p \to \neg q)$	$\neg\neg q \vdash q$,$p \to q \vdash \neg\neg(p \to q)$	no
LenI	$p \to (q \to \neg p) \vdash p \to \neg q$	$\neg\neg q \vdash q$	yes
Ben-Ari	$\vdash (p \to q) \to (\neg q \to \neg p)$	$\neg\neg p \to p, \vdash \neg q \to (q \to \neg\neg q)$	yes
Ben-Ari	$\vdash (\neg p \to q) \to ((\neg p \to \neg q) \to p)$	$\vdash (\neg p \to p) \to p$	yes
Kelly	$\vdash \neg(p \to q) \to \neg q$	$\vdash (q \to (p \to q)) \to (\neg(p \to q) \to \neg q)$	yes
Kelly	$\vdash (p \to q) \to ((\neg p \to q) \to q)$	$\vdash (\neg q \to q) \to q, \vdash (p \to q) \to (\neg q \to \neg p)$	yes

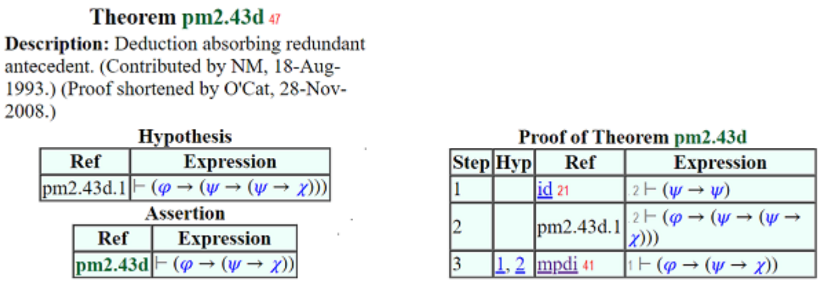
originally only used one of the two lemmas, and the proof by LogAx was longer than the solution in the logic course notes. After some minor changes in the implementation of the heuristics, LogAx uses no lemmas, but generates a shorter proof. Both proofs are given in Appendix 4.A. Since exercises in textbooks also use derived rules, we have only 4 extra exercises with lemmas in these books. All the proofs of these exercises are equal to the LogAx proofs.

To evaluate more proofs we use the large collection of proofs on the Metamath website.[1] This website collects formal proofs, not only for logic statements, but also for mathematical statements. The part on propositional logic contains proofs of well-known theorems, for example from the Principia Mathematica by Russell and Whitehead. In general, a Metamath theorem is presented as follows:

$$\vdash \phi_1 \text{ and } \vdash \phi_2 \text{ and ... and } \vdash \phi_n \Rightarrow\ \vdash \psi$$

So if ϕ_1, ..., ϕ_n are provable, then ψ is provable. Since LogAx cannot deal with general tautologies, we translate a Metamath theorem in a theorem with assumptions. Instead of $\vdash \phi_1$ and $\vdash \phi_2$ and ... and $\vdash \phi_n \Rightarrow\ \vdash \psi$, we prove $\phi_1, \phi_2, ..., \phi_n \vdash \psi$. Since proofs in Metamath build on each other, these proofs seem to be natural candidates to use in a comparison with LogAx with lemmas. However, the best way to compare proofs is not immediately clear. To demonstrate this, we present an example of a Metamath proof in Fig. 4.6. This example shows a proof of $\phi \to (\psi \to (\psi \to \chi)) \vdash \phi \to (\psi \to \chi)$. The proof uses two previous theorems: id ($\vdash \psi \to \psi$) and mpdi (from $\vdash \phi \to \chi$ and $\vdash \phi \to (\psi \to (\chi \to \theta))$ follows $\vdash \phi \to (\psi \to \theta)$). The most direct way to translate this in a LogAx proof with lemmas would be by rewriting mpdi in a theorem with assumptions ($\phi \to \chi, \phi \to (\psi \to (\chi \to \theta)) \vdash \phi \to (\psi \to \theta)$), and using instantiated versions of these theorems as lemmas in the proof. However, to complete this proof, a forward application of Deduction followed by an application of Modus Ponens by LogAx suffices, which makes the comparison not very informative. A more interesting way to compare proofs would be by letting LogAx find useful instantiations of lemmas, but so far, we have not implemented this functionality. In the comparison,

[1]`http://us.metamath.org/mpegif/mmtheorems.html`



Theorem pm2.43d 47

Description: Deduction absorbing redundant antecedent. (Contributed by NM, 18-Aug-1993.) (Proof shortened by O'Cat, 28-Nov-2008.)

Hypothesis

Ref	Expression
pm2.43d.1	⊢ (φ → (ψ → (ψ → χ)))

Assertion

Ref	Expression
pm2.43d	⊢ (φ → (ψ → χ))

Proof of Theorem pm2.43d

Step	Hyp	Ref	Expression
1		id 21	2 ⊢ (ψ → ψ)
2		pm2.43d.1	2 ⊢ (φ → (ψ → (ψ → χ)))
3	1, 2	mpdi 41	1 ⊢ (φ → (ψ → χ))

Figuur 4.6: An example of a Metamath proof

we therefore add just a single instantiated lemma to LogAx. We inline the other lemmas in the Metamath proofs. Since Metamath proofs do not make use of the deduction theorem, a last adaptation we have to make is to remove applications of the deduction theorem in the LogAx proofs. We use the constructive proof of the deduction theorem to remove occurrences of Deduction in LogAx proofs, and compare these proofs with the Metamath proofs. The results are shown in Appendix 4.B. The comparison consists of 24 theorems, 12 with and 12 without negation. Nearly two thirds (15 out of 24) proofs are equal except for the order of the lines. The LogAx proof is shorter than the Metamath proof in eight cases. In seven of these cases LogAx does not use the lemmas. It is not surprising that in these cases, inlined proofs in Metamath are longer: the Metamath proofs are constructed by choosing suitable lemmas. Metamath does not inline proofs and uses lemmas that are known theorems, or variants, for example originating from the Principia Mathematica. Hence a Metamath proof is short when it uses a small set of lemmas, without caring about the total length of the inlined proof.

In Lodder et al. (2017) we compared 30 Metamath proofs with proofs generated by LogAx without lemmas. After inlining the used lemmas in Metamath and removing applications of the deduction theorem in the LogAx proofs, we found that 27 LogAx proofs were equally long as the Metamath proofs. In three cases the LogAx proof was shorter than the Metamath proof. Although the comparison of LogAx with Metamath can only be done indirectly, the results (from the proofs with lemmas nearly two thirds of the LogAx proofs are equal to Metamath proofs, and most of the LogAx proofs have equal length as these proofs) indicate that LogAx indeed generates proofs that are comparable to expert proofs.

4.8.2 Recognition of student solutions

In a second evaluation we investigate whether or not correct student solutions can be recognized by LogAx. Axiomatic proofs are part of a course on logic where we treat also other topics, such as semantics of predicate logic, axiomatic proofs in predicate logic, structural induction and Hoare calculus. Students may hand in homework to earn a bonus point. The homework exercises contain one exercise about propositional logic axiomatic proofs and one on predicate logic axiomatic proofs. Furthermore, the exams usually have an exercise on axiomatic proofs. We use solutions of two homework exercises, and one exam exercise, to determine whether or not LogAx recognizes student proofs. In the exam exercise, students have to prove that $\neg\neg p \to \neg q, r \to q \vdash r \to \neg p$. The correct student solutions to this exercise can be divided into two groups, where each group contains solutions that are equal up to the order of the proof lines. Solutions in the first group contain an application of Axiom a, b and c, and no application of the deduction theorem. Solutions in the second group contain an application of the deduction theorem, and of Axiom c. From the 19 correct solutions, the majority (16) belongs to the second group, and the remaining three solutions to the first group. The example solution provided by LogAx also belongs to the second group. The solutions of the first group do not (yet) appear in our initial DAM, but they do appear in the DAM we dynamically obtain when a student introduces Axiom b, and we use this DAM to provide feedback. In the future we might add an extra heuristic for the use of Axiom b:

- If the top goal equals $\Delta \vdash \phi \to \chi$, and $\Delta' \vdash \phi \to \psi$ and $\Delta'' \vdash \psi \to \chi$ both appear in the DAM, where $\Delta' \cup \Delta'' \subseteq \Delta$, then add an instance $(\phi \to (\psi \to \chi)) \to ((\phi \to \psi) \to (\phi \to \chi))$ of Axiom b to the DAM.

In the first homework exercise students have to prove that $q, \neg p \to (q \to \neg r) \vdash r \to p$. Here almost all student solutions (16) use Axiom a, b and c. Only one student uses the deduction theorem instead of Axiom b. LogAx generates this last proof. Solutions using the three axioms are not part of the DAM that is generated at the start of the exercise, but can be recognized by a dynamically generated DAM. The second homework exercise is an exercise in predicate logic, but it contains a propositional part that amounts to a proof of $(p \to q) \to \neg p \vdash q \to \neg p$. Again, there were two groups of solutions: 13 students use Axiom a and the deduction theorem, and 2 students use an extra application of the deduction theorem instead of Axiom a. In this case the solution generated by LogAx is the solution that does not use Axiom a, but the DAM also contains a solution with Axiom a. We summarize the results in Table 4.3. The first column (preferred) shows the number of solutions that corresponds to the preferred solution of LogAx, the second the number that corresponds to a non-preferred solution, and solutions in the third column can be recognized by a dynamically generated DAM. The conclusion of this evaluation is that with the use of dynamically generated DAMs, we can recognize all student solutions, and also give hints. Still, we might optimize LogAx by adding more heuristics, e.g. such that the solution generated by LogAx for homework exercise 2 equals the student solutions. In the current implementation, heuristics for the use of Axiom b and the deduction theorem interfere: extra heuristics for Axiom b can broaden the DAM, but the algorithm for the distillation of a linear

Tabel 4.3: Recognized solutions

Exercise	preferred	non-pref.	dynamic	total
Exam	16		3	19
Homework 1	1		16	17
Homework 2	2	13		15

proof prefers applications of the deduction theorem, a local decision. We would have to extend this algorithm with global heuristics to ensure that the extracted proof contains instances of axioms when applicable.

4.9 Small-scale experiments with students

We have performed several small-scale experiments with LogAx. The main results in this section are obtained from an experiment with LogAx without lemmas, performed in May 2018. The 18 participants in this experiment were preparing for admission to a master program Computer Science at the Open University of the Netherlands. A course on logic is part of the premaster program. Most students combine their study with a job, hence the online lessons are during the evening. We required participants to submit a solution to the first homework exercise, see Section 4.8.1, before participating in the experiment. Thus, we guaranteed that participants had studied the subject before the experiment. We used their solutions to the exercise as an indication of their prior knowledge. The experiment consisted of a 20-minute (online) instruction, after which students practiced with the tool for 75 minutes. The experiment concluded with a 20-minute posttest. All interactions of the students with LogAx were logged. The 10 exercises in the tool and the questions in the posttest can be found in Appendix 4.C.

We use the results of this experiment

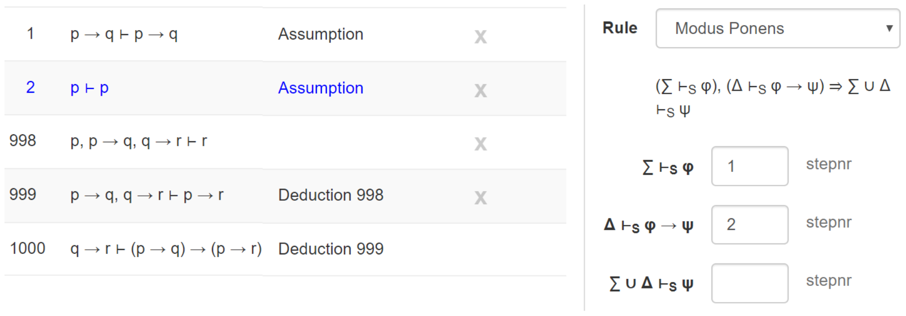
- to evaluate the hints and feedback given by LogAx,
- to analyze the way students use LogAx, and
- to evaluate the effect of using LogAx on students' performance.

4.9.1 Evaluation of hints and feedback

We start with evaluating hints and feedback by answering the following questions:

- does LogAx recognize common mistakes?
- is the feedback sufficient to repair mistakes?
- do hints on subgoals help students to reach these subgoals?

To answer these questions we analyze the loggings of the experiments. Apart from 5 syntax errors, the loggings contain 179 incorrect steps of a total of 1480



1	$p \to q \vdash p \to q$	Assumption	x
2	$p \vdash p$	Assumption	x
998	$p, p \to q, q \to r \vdash r$		x
999	$p \to q, q \to r \vdash p \to r$	Deduction 998	x
1000	$q \to r \vdash (p \to q) \to (p \to r)$	Deduction 999	

Rule: Modus Ponens

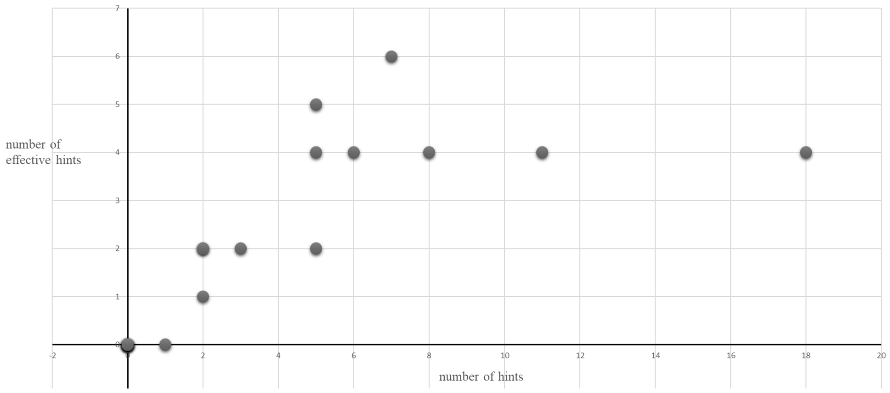
$(\Sigma \vdash_S \varphi), (\Delta \vdash_S \varphi \to \psi) \Rightarrow \Sigma \cup \Delta \vdash_S \psi$

$\Sigma \vdash_S \varphi$	1	stepnr
$\Delta \vdash_S \varphi \to \psi$	2	stepnr
$\Sigma \cup \Delta \vdash_S \psi$		stepnr

Figuur 4.7: A common error: interchanging line 1 and 2 in the dialog box

steps performed by the students. The syntax errors were performed by 5 different students who could repair this error in the next step without asking for extra help. We conclude that the dialog box indeed prevents students from spending time repairing syntax errors. In 24% (43/179) of these steps, a student tries to apply Modus Ponens, but interchanges the first and second line in the dialog box, as shown in Fig. 4.7. This typically occurs when the implication ($\phi \to \psi$) has a smaller line number than the antecedent of this implication (ϕ). During the experiment we did not have a buggy rule implemented for this situation, and hence students received the feedback 'ϕ is not an implication', or 'Modus Ponens is not applicable'. In 34 out of the 43 occurrences of this kind of mistake, this feedback was sufficient for the student to fill in the dialog box correctly. In the other cases students asked for a hint or next step, or continued with another rule. One student did not realize that the implication may precede the antecedent, and consequently constructed the proofs in such a way that implications always have a higher line number than the antecedents.

A second category of mistakes also seems to have its origin in an incorrect use of the dialog box. Examples are backward applications of Modus Ponens where the last line is already motivated. Some buggy applications of Modus Ponens were only recognized when students completed the dialog box 'correctly' (entering the implication in the second field). For instance, an erroneous application of Modus Ponens on formulae $\Sigma \vdash \phi \to \psi$ and $\Delta \vdash \psi$ is not recognized if the student enters the line number of the first line in the uppermost (antecedent) field, and the second in the middle (implication) field. Some misreading of parentheses is recognized by LogAx, but, for example, the misreading of parentheses in an application of Modus Ponens on $\Sigma \vdash p \to (q \to r)$ and $\Delta \vdash (p \to q) \to (r \to (p \to q))$ is not recognized as a common mistake. Also mistakes in the introduction of an axiom or assumption in the dialog box, such as interchanging p and q with a faulty Modus Ponens application on $\Sigma \vdash \neg p \to \neg q$ and $\vdash (\neg q \to \neg p) \to (p \to q)$ as a result, is not recognized. The remaining errors that are not recognized cannot be classified



Figuur 4.8: The effectivity of subgoal hints

as buggy rules, since we cannot find a pattern in, or a misconception as a cause of, these errors. We further analyzed the loggings to see whether in the cases that a common error is detected (86 errors), the error message is sufficient to help a student in making progress. In 60% (52/86) of these cases, a student can proceed without help of the system, in over 8% (7/86) a student gives up on the exercise directly after this mistake, and in 3% (3/86) after one or more erroneous steps, and in the other cases a student can proceed with a hint or next step. If a student needs more help, this does not necessarily mean that the error message is not clear: the loggings suggest that often a student recognizes the mistake (in 60% of the cases a student does not make the same error again during the session), but does not know how to proceed. We conclude that we can improve error messages by recognizing the cases where a student interchanges the lines in the dialog box. In the 43 cases where interchanging the lines was the only mistake, students will receive a message about how to fill in the dialog box correctly. In 15 other cases, where students now get a default message or a message that probably does not refers to the actual mistake, we will also improve the feedback. An example of this situation is the application of Modus Ponens on $\Delta \vdash p \rightarrow q$ and $\Delta' \vdash q$ (entered in this order). At this moment, students receive the error message that q is not an implication, but after recognizing this mistake as a combination of a common error and a swapping of the lines in the dialog box, the error message will be that the formula in the second line should be equal to the left hand side of the implication instead of the right hand side. With these improvements we would have provided specific feedback in 80% ((86 + 43 + 15) / 179) of the mistakes made by students.

Since the possibility to give a hint about a subgoal was new in the LogAx version that we used in this experiment, we evaluate the effect of this type of hints. A student receives a hint that indicates a subgoal to be reached if it takes more than one step to reach this subgoal, and if the subgoal is not already present in the proof as an unmotivated line. LogAx gives a hint about a subgoal in 75 of its 192 hints, and 40 of these subgoals were reached by the students without further assistance of LogAx. In the other cases, the students used next hints which told them the rule to proceed or a next step. This number might seem somewhat disappointing, but a more detailed analysis shows that in general only students who ask a lot of help do not reach the subgoal without extra help. Fig. 4.8 presents a scatter plot with the total number of different subgoal hints given to the student on the x-axis, and the number of reached subgoals after the hint, without further help on the y-axis. The figure shows that students who use fewer than eight different subgoal hints (hints in different exercises or in different stages of their proof), in general reach the subgoal themselves.

4.9.2 Use of LogAx

A second topic in this evaluation is the question: how do students use LogAx. Do students misuse the system by asking too much help, or by randomly filling in the dialog boxes, or do they struggle without using the help offered by LogAx? The results of our experiment show that the use of LogAx is related to performance in the homework exercise, which we use as a pretest. From the 12 students who score at least 0.8 (out of 1) in the pretest, eight students can complete the exercises without using much help or making many mistakes. One student reported that he completed the exercises with pen and paper and used LogAx mainly to check his answers. Two students use quite a lot of hints, also to complete the exercises, and one student seems to misuse the system by performing lots of actions (338 interactions versus an average of 173). From the students who score lower than 0.8 on the pretest, only one student completes all the exercises without much help. In this group help seeking strategies differ considerably: two students hardly ask any help, one student performs 65 help seeking actions. We conclude that in this experiment, in general, good students use LogAx as intended, and can complete exercises without a lot of help. Since only six weaker students participated, we cannot draw hard conclusions, but the loggings seem to indicate that these students either tend to overuse or underuse help. Another observation is that most of the weaker students can complete the first four or five easier exercises, but the last exercises seem to be too difficult for this group.

4.9.3 Evaluation of learning effects

The last question we want to answer is whether LogAx supports students with learning axiomatic proofs. Most of the participants in the experiment were good students, who performed already well on the pretest: the average score on the pretest was 0.84, where the maximum possible score was 1. This might have influenced learning effects negatively. The posttest consisted of two parts. In the first part, students had to point out possible errors in five small proofs. Four of the five proofs

Tabel 4.4: Results of the posttest

Exercise	1a	1b	1c	1d	1e	2
Average score	0.78	0.39	0.72	0.34	0.83	0.61

were incorrect. The incorrect proofs contained common errors collected from the homework exercises, see Section 4.7.1. We deliberately added errors that are possible to make in LogAx and errors that are prevented by the interface, since we wanted to know whether the last type would occur more often in the posttest. In the second part, students had to provide a proof. The scores of the posttest can be found in Table 4.4. The low scores on exercise 1b and 1d are remarkable. The score on exercise 1b was more or less expected, because the exercise contains an error in the set of assumptions after applying Modus Ponens. Since LogAx automatically determines this set, students do not practice in correctly determining this set. Exercise 1a also contained an error that is not possible in LogAx (mixing an application of Modus Ponens with Axiom b), but this error was recognized by students. Exercise 1d applies deduction in 'the wrong direction', a common mistake made by students, which is apparently not sufficiently corrected while practicing with LogAx (the loggings contain only 3 occurrences of this error). The misreading of parentheses in exercise 1e (a possible error in LogAx) is recognized by most students. Students scored lower on exercise 2, an exercise in which a student has to construct a proof, than in the pretest. We hypothesize that this is caused by fatigue (since most of our students combine study with a job, the experiment took place on an evening and the posttest started at 21:15), and the fact that we asked students not to spend more than 20 minutes on the posttest, while they could spend as much time as they needed for the pretest, since this was part of the homework assignment. We also looked at the results on the exam. Students who participated in the experiment receive on average the same score for the exercise on axiomatic proofs as they got for the pretest. Since most students participated in the experiment, we cannot compare their results with students who did not participate. The results of an earlier experiment (January 2018) with 9 students were more or less comparable. The average score on the pretest of this group is a little lower (0.8) and also the score on the posttest proof, exercise 2, is lower (0.52). The results on the exam are a bit higher (0.89), but the exercise was slightly different and the time between the experiment and the exam was considerably shorter: 13 days instead of 40 for the last experiment. We conclude that at this moment we do not have enough data to evaluate learning effects. However, experiments with other tools (Lodder et al., 2019) show that this kind of tutoring systems can be effective.

4.10 Related work

As mentioned in the introduction, there are two e-learning tools that can be used to practice the construction of Hilbert-style axiomatic proofs in propositional logic: Metamath Solitaire (Megill, 2007) and Gateway to logic (Gottschall, 2012). Both tools are proof-editors: a student chooses an applicable rule and the system applies this rule automatically. These systems provide no help on how to construct a proof. There are quite a lot of systems that help students with other kinds of exercises in logic, and many more in other subjects.

In the AProS project, Sieg and collegues have developed Proof Tutor, a tutor that teaches students natural deduction (Sieg, 2007; Perkins, 2007). They have developed an automated proof search method, which differs from the Bolotov method in the use of normal proofs. Their algorithm uses a set of tactics that are explicitly used as hints for students. Perkins (2007) describes how they provide help such as hints or next steps in the case that the partial solution of a student diverges from a generated model solution. First, they check if the subgoals of the partial solution are indeed derivable from the assumptions (we do not need this check since a situation in which a subgoal is not derivable is not possible in LogAx). Second, they check whether the partial solution can be completed by the Proof Tutor; if this is not the case, they let the student erase the lines of the part that does not belong to a generated proof. In LogAx we do not let students erase lines. The consequence is that a final proof may contain unnecessary lines, but also that we will not erase a useful part that is not recognized as useful by LogAx.

We use a strategy language to generate both the solutions and the feedback (Heeren et al., 2010). The use of such a language is related to the use of production rules as for example in Anderson et al. (1995) and Corbett et al. (1997), and the way in which we recognize student solutions is akin to their model-tracing approach. The Geometry Tutor makes use of contextualized rules, which means that a rule will only fire in a specific context. The different axioms that we add in step 2(b) of our version of the algorithm depend on assumptions in the statements $\Delta' \vdash \phi$ and $\Delta'' \vdash \neg\phi$, and these rules could also be perceived as contextualized rules. When more than one rule can be applied in the same situation, tools based on production systems may add preferences to specific rules (Anderson et al., 1995; Jaques et al., 2013). We use the strategy language to specify a preference in the application of the rules.

Ahmed et al. (2013) use a different approach to generate and solve natural deduction exercises. Their main idea is to use truth tables, representing an equivalence class of logical formulae. The representations of these formulae are used in a proof graph of predefined size. Proof generation consists of finding a truth-table-representative of the assumptions and conclusion, searching for a proof using these representatives, and adding rewrite steps (as for example replacing subformulae of the form $\neg\neg\phi$ by ϕ or vice versa) to this proof. In this way they can generate exercises and solutions typically used in education. However, in their approach it is essential that rewrite steps on subformulae are allowed, which is not the case in Hilbert axiomatic systems, and also often not in natural deduction systems.

Answer set programming, as used for example by O'Rourke et al. (2019), is related to the production systems used by Anderson et al. (1995). Their program finds all different solutions of an algebra exercise, using deductive rules and integrity constraints that forbid certain solutions. Also explanations and a subset of misconceptions can be generated automatically. To restrict the search space to a finite space (which is necessary in their program), the number of terms in an equation and the solution length is maximized. In LogAx we do not need to maximize the length of formulae; formula length is not a good measure for the expected proof length. For technical reasons, we maximize calculation time, but thus far LogAx has been able to solve all problems within this limit.

For some domains it is possible to use existing tools to generate solutions or correct student solutions. An example of this approach is Sadigh et al. (2012): to solve state machine problems such as finding a trace for a given model that violates or satisfies a certain property, they use existing model checking tools. This approach can be very useful to produce solutions or grade exercises, but is in general less suitable for giving hints, since steps performed by a tool not always correspond to human steps.

4.11 Conclusion and future work

By using an existing algorithm for natural deduction, we developed a sound and complete algorithm to generate Hilbert-style axiomatic proofs, and introduced a representation of these proofs as a directed acyclic multi graph (DAM). We use these DAMs in a new interactive tutoring tool LogAx to give hints and next steps to students, and to extract model solutions. Comparing the generated proofs with expert solutions shows that the quality of the proofs is comparable to that of expert proofs. The tool recognizes most of the steps in a set of student solutions, and in case a step diverges from the generated proof, LogAx can still provide hints and next steps. This holds both for the original version of LogAx as well as for the extension with lemmas. We derived buggy rules from a set of student solutions, and added these to LogAx. Evaluating with a test set showed that this set covers the majority of student errors. In an experiment with students we discovered that we overlooked a source of errors originating in the user interface (errors made while filling in the dialog box for Modus Ponens). We expect that after adding buggy rules for this kind of errors, LogAx will recognize about 80% of the errors.

We performed several pilot evaluations with LogAx. Since the number of participating students was low, and students performed already well on the pretest, we cannot derive conclusions about the learning effect of LogAx. However, we conclude that well-prepared students use the system as intended, and can complete most of the exercises without using much help or making a lot of mistakes. Students who do not ask more hints than average, reach the subgoal given in a hint without extra help. Future evaluations can investigate whether students indeed learn by using LogAx. It might be necessary to add more easy exercises for weaker students, since the results of the evaluation indicate that the difficulty level of the exercises rises too quickly for this category of students. Also mechanisms to diminish excessive hint use, or stimulate hint use in case of minimal use might help the weaker students to benefit from the use of LogAx.

We developed algorithms for generating DAMs and distilling proofs from this DAM for the domain of Hilbert-style axiomatic proofs. The strategy language used to formulate these algorithms is much broader applicable, and we expect that also some of the ideas used in our algorithms can be applied to other problem domains. For example, the dynamic extension of partial student proofs, or the use of the strategy language to order the introduction of the proof steps in a way that corresponds to an expert's pen-and-paper proof, could be based on techniques similar to LogAx.

Acknowledgments

We thank Marianne Berkhof and Daan de Wit for their work on the student interface, and our students from the course 'Logic and Computer Science' for their permission to use their solutions in our research, and participation in the experiment.

Tabel 4.5: Solution of exercise 11.1.5 in the logic course:

1.	$\vdash q \to (p \to q)$	Axiom a
2.	$\neg\neg q \vdash q$	Lemma
3.	$\neg\neg q \vdash p \to q$	Modus Ponens 1 2
4.	$p \to q \vdash \neg\neg(p \to q)$	Lemma
5.	$\vdash (p \to q) \to \neg\neg(p \to q)$	Deduction 4
6.	$\neg\neg q \vdash \neg\neg(p \to q)$	Modus Ponens 3 5
7.	$\vdash \neg\neg q \to \neg\neg(p \to q)$	Deduction 6
8.	$\vdash (\neg\neg q \to \neg\neg(p \to q)) \to (\neg(p \to q) \to \neg q)$	Axiom c
9.	$\vdash \neg(p \to q) \to \neg q$	Modus Ponens 7 8
10.	$\neg(p \to q) \vdash \neg(p \to q)$	Assumption
11.	$\neg(p \to q) \vdash \neg q$	Modus Ponens 9 10
12.	$\vdash \neg q \to (\neg p \to \neg q)$	Axiom a
13.	$\neg(p \to q) \vdash \neg p \to \neg q$	Modus Ponens 11 12
14.	$\vdash \neg(p \to q) \to (\neg p \to \neg q)$	Deduction 13

4.A Exercise 11.1.5

In Table 4.5 and Table 4.6 we compare the solution of the logic course exercise 11.1.5 by the solution generated by LogAx. Note that the second solution does not use the lemmas and hence is in fact shorter than the solution in the course.

4.B Metamath theorems compared with LogAx with lemmas

Table 4.7 compares a set of proofs in Metamath with generated proofs by LogAx using lemmas. The first column gives the name of the theorem in Metamath, the second column the theorem and the third the lemma that is used. The last two columns show the number of steps. Proofs with the same number of steps are equal except for any differences in the order of the steps.

4.C Exercises used in the experiment and the posttest

The exercises presented by LogAx in the experiment are listed in Table 4.8, while the questions of the posttest are presented below:

Exercise 1
Check whether these proofs are correct. In case of an incorrect proof, indicate the incorrect step, and explain why this step is not correct.

Tabel 4.6: Solution of exercise 11.1.5 generated by LogAx:

1.	$\neg\neg q \vdash q$	Lemma
2.	$p \to q \vdash \neg\neg(p \to q)$	Lemma
3.	$\neg p \vdash \neg p$	Assumption
4.	$\vdash \neg p \to (\neg q \to \neg p)$	Axiom a
5.	$\neg p \vdash \neg q \to \neg p$	Modus Ponens 3 4
6.	$\vdash (\neg q \to \neg p) \to (p \to q)$	Axiom c
7.	$\neg p \vdash p \to q$	Modus Ponens 5 6
8.	$\neg(p \to q) \vdash \neg(p \to q)$	Assumption
9.	$\neg(p \to q) \vdash \neg\neg q \to \neg(p \to q)$	Deduction 8
10.	$\vdash (\neg\neg q \to \neg(p \to q)) \to ((p \to q) \to \neg q)$	Axiom c
11.	$\neg(p \to q) \vdash (p \to q) \to \neg q$	Modus Ponens 9 10
12.	$\neg(p \to q), \neg p \vdash \neg q$	Modus Ponens 7 11
13.	$\neg(p \to q) \vdash \neg p \to \neg q$	Deduction 12
14.	$\vdash \neg(p \to q) \to (\neg p \to \neg q)$	Deduction 13

a

1.	$p \to (p \to \neg q) \vdash (p \to p) \to (p \to \neg q)$	Axiom b
2.	$p \to p \vdash p \to p$	Assumption
3.	$p \to (p \to \neg q) \vdash p \to \neg q$	Modus Ponens 1, 2

b

1.	$p \vdash p$	Assumption
2.	$p \to (q \to r) \vdash p \to (q \to r)$	Assumption
3.	$p, p \to (q \to r) \vdash q \to r$	Modus Ponens 1, 2
4.	$q \vdash q$	Assumption
5.	$q \to r, q \vdash r$	Modus Ponens 3, 4

c

1.	$p \vdash p$	Assumption
2.	$\vdash p \to p$	Deduction 1
3.	$\vdash (p \to p) \to (p \to (p \to p))$	Axiom a
4.	$\vdash p \to (p \to p)$	Modus Ponens 2, 3

d

1.	$p \to q \vdash p \to q$	Assumption
2.	$p \to q, p \vdash q$	Deduction 1

Tabel 4.7: Comparison of Metamath proofs with LogAx with lemmas

name	theorem	used lemma	steps MM	steps LogAx
idd	$\vdash p \to (q \to q)$	$\vdash q \to q$	3	3
pm2.27	$\vdash p \to ((p \to q) \to q)$	$\vdash (p \to q) \to (p \to q)$	9	9
pm2.43d	$p \to (q \to (q \to r)) \vdash p \to (q \to r)$	$\vdash q \to q$	13	13
pm2.43	$\vdash (p \to (p \to q)) \to (p \to q)$	$\vdash p \to ((p \to q) \to q)$	3	12
syldd	$p \to (q \to (r \to s)), p \to (q \to (s \to t)) \vdash p \to (q \to (r \to t))$	$\vdash (s \to t) \to ((r \to s) \to (r \to t))$	21	21
imim1d	$p \to (q \to r) \vdash p \to ((r \to s) \to (q \to s))$	$\vdash p \to (s \to s)$	39	22
imim1	$\vdash (p \to q) \to ((q \to r) \to (p \to r))$	$\vdash (p \to q) \to (p \to q)$	33	16
pm2.83	$\vdash (s \to (p \to q)) \to ((s \to (q \to r)) \to (s \to (p \to r)))$	$\vdash (p \to q) \to ((q \to r) \to (p \to r))$	11	11
com23	$p \to (q \to (r \to s)) \vdash p \to (r \to (q \to s))$	$\vdash r \to ((r \to s) \to s)$	29	26
pm2.04	$\vdash (p \to (q \to r)) \to (q \to (p \to r))$	$\vdash (p \to (q \to r)) \to (p \to (q \to r))$	23	20
com34	$s \to (t \to (p \to (q \to r))) \vdash s \to (t \to (q \to (p \to r)))$	$\vdash (p \to (q \to r)) \to (q \to (p \to r)))$	11	11
loowoz	$\vdash ((p \to q) \to (p \to r)) \to ((q \to p) \to (q \to r))$	$\vdash ((p \to q) \to (p \to r)) \to (q \to (p \to r))$	7	7
pm2.21	$\vdash \neg p \to (p \to q)$	$\vdash \neg p \to \neg p$	7	8
pm2.24	$\vdash p \to (\neg p \to q)$	$\vdash \neg p \to (p \to q)$	9	9
pm2.28	$\vdash (\neg p \to p) \to p$	$\vdash \neg p \to (p \to \neg(\neg p \to p))$	17	17
pm2.18d	$p \to (\neg q \to q) \vdash p \to q$	$\vdash (\neg q \to q) \to q$	7	7
notnot2	$\vdash \neg\neg p \to p$	$\vdash \neg\neg p \to (\neg p \to p)$	29	22
notnotrd	$p \to \neg\neg q \vdash p \to q$	$\vdash \neg\neg q \to q$	7	7
notnotri	$\neg\neg p \vdash p$	$\vdash \neg\neg p \to p$	3	3
con2d	$p \to (q \to \neg r)] \vdash p \to (r \to \neg q)$	$\vdash \neg\neg q \to q$	29	25
pm3.2im	$\vdash p \to (q \to \neg(p \to \neg q))$	$\vdash p \to ((p \to \neg q) \to \neg q))$	49	36
jc	$p \to q, p \to r \vdash p \to \neg(q \to \neg r)$	$\vdash q \to (r \to \neg(q \to \neg r))$	37	11
expi	$\neg(p \to \neg q) \to r \vdash p \to (q \to r)$	$\vdash p \to (q \to \neg(p \to \neg q))$	11	11
pm2.251	$\vdash \neg(p \to q) \to (q \to p)$	$\vdash \neg(p \to q) \to p)$	7	7

e

1.	$p \to (q \to r), p, p \to q \vdash p \to (q \to r)$	Assumption
2.	$p \to (q \to r), p, p \to q \vdash p \to q$	Assumption
3.	$p \to (q \to r), p, p \to q \vdash r$	Modus Ponens 1, 2

Exercise 2
Prove axiomatic: $(p \to q) \to r \vdash q \to r$

Tabel 4.8: Exercises in LogAx.

1.	$\neg q \to \neg p \vdash p \to q$
2.	$p, p \to q, q \to r \vdash r$
3.	$p \to (q \to r) \vdash (p \to q) \to (p \to r)$
4.	$\vdash p \to ((p \to q) \to q)$
5.	$p, p \to q \vdash r \to q$
6.	$p \to q \vdash (r \to p) \to (r \to q)$
7.	$p, \neg p \vdash q$
8.	$q, (p \to q) \to r \vdash r$
9.	$p \to \neg q \vdash p \to (q \to r)$
10.	$q, \neg(p \to q) \vdash r$

5 Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction

5.1 Introduction

Discrete structures play an important role in many domains, and are foundational for mathematics, logic, and computer science. Examples of such structures are natural numbers, data structures such as lists and trees, but also complex structures such as programming languages. Structural induction is a proof technique that is widely used to prove statements about inductively defined, discrete structures. Mathematical induction can be viewed as a special kind of structural induction, using the inductive definition of the natural numbers as the underlying structure.

Because discrete structures and structural induction are foundational for mathematics and computer science, they form an integral part of educational programs. For example, proof techniques are part of the ACM Computer Science curriculum.[1] Courses that address proof techniques often require students not only to learn how to prove consequences in a formal system, but also to reason about formal systems, and to independently construct a proof for a statement. A typical example of an exercise that occurs in many textbooks on logic and proof techniques is the following: prove that the number of left parentheses in a logical formula is equal to the number of right parentheses. Such a property can be proved by structural induction, where the structure of the proof follows the structure of the inductive definition of the logical language. Students have to learn this proof technique to construct more fundamental proofs, such as the soundness of a proof system. Textbooks and teachers typically instruct students on how to do this, provide some examples, and then let students practice with constructing proofs themselves. As with learning any subject, students need feedback when they are learning how to construct their own proofs (Hattie and Timperley, 2007). Such feedback can take several forms: it may be about the progress of a student, about recommending a next task to solve, or about the difference between the proof constructed by a student and an expected proof. In this paper we focus on the latter kind of feedback.

This paper discusses the design of a tutoring system for practicing proving statements about inductively defined, discrete structures. Some core features of the

[1] `https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf`

system are that it gives feedback on the steps a student takes towards a solution, and hints about which step to take next. We use the knowledge about misconceptions students have to determine what feedback to give. Students have some freedom in setting up their proof, and the system helps in reaching the learning goal of independently constructing a proof for statements about inductively defined discrete structures. Two advantages of our system are that it gives immediate feedback, and that it is scalable because feedback is calculated automatically.

This paper is organized as follows. After introducing our terminology in Section 5.2, we discuss related work in Section 5.3. Our research is partly motivated by students' problems with induction, discussed in Section 5.4. Section 5.5 describes the interface and functionality of LogInd, and in Section 5.6 we show how this functionality is realized. A pilot experiment is discussed in Section 5.8, and we conclude in Section 5.9 with conclusions and ideas for future work.

5.2 Terminology

We use an example exercise to illustrate the terminology concerning inductive proofs that we use in this paper. The text of the exercise is:

"The propositional language L has atoms p, q, r, ... and connectives $\neg$, $\wedge$ and $\rightarrow$. We define two functions on this language: a function *prop* counting all occurrences of propositional letters, and a function *bin* counting the number of binary connectives. These functions are inductively defined by:

$$\begin{aligned}
&\mathit{prop}\ (p) &&= 1\\
&\mathit{prop}\ (\neg\phi) &&= \mathit{prop}\ (\phi)\\
&\mathit{prop}\ (\phi \mathbin{\Box} \psi) &&= \mathit{prop}\ (\phi) + \mathit{prop}\ (\psi)\\
&\mathit{bin}\ (p) &&= 0\\
&\mathit{bin}\ (\neg\phi) &&= \mathit{bin}\ (\phi)\\
&\mathit{bin}\ (\phi \mathbin{\Box} \psi) &&= \mathit{bin}\ (\phi) + \mathit{bin}\ (\psi) + 1
\end{aligned}$$

where p is an atom, and $\Box$ is $\wedge$ or $\rightarrow$. Prove with induction that $\mathit{prop}\ (\phi) = \mathit{bin}\ (\phi) + 1$ for any formula ϕ in the language L."

The statement $\mathit{prop}\ (\phi) = \mathit{bin}\ (\phi) + 1$ in the last sentence is the *theorem* or *property* that has to be proven. The structure of an inductive proof for this theorem can be deduced from the inductive definition of the language L. The *base case* consists of a proof of the theorem for atomic formulae. There is an *inductive case* for each of the connectives in the language. For instance, a proof of the conjunction case is a proof that from the assumption that if the theorem holds for ϕ and ψ (i.e. $\mathit{prop}\ (\phi) = \mathit{bin}\ (\phi) + 1$ and $\mathit{prop}\ (\psi) = \mathit{bin}\ (\psi) + 1$) it follows that the theorem also holds for $\phi \wedge \psi$ (i.e. $\mathit{prop}\ (\phi \wedge \psi) = \mathit{bin}\ (\phi \wedge \psi) + 1$). The assumption that the theorem holds for some arbitrary formulae ϕ and ψ is the *induction hypothesis.* A *subproof* is a part of the complete inductive proof where a base case or inductive case is proven.

5.3 Related work

Tools that assist a user in constructing structural induction proofs may have different functionalities or purposes. In this section we describe four different kinds of tools: automated theorem provers, proof assistants with didactic functionality, e-learning tools for mathematical induction, and e-learning tools for structural induction.

Bundy (2001) states that Gödel's incompleteness theorem implies that it is impossible to construct a completely automatic inductive theorem prover, and that Kreisel's result on cut-elimination (Kreisel, 1965) implies that inductive proofs in general will need intermediate lemmas. Hence, automated theorem provers for induction are proof assistants that use several heuristics to try to automatically prove theorems as much as possible. The classical literature on automated theorem proving, for example the handbook of Boyer-Moore (Boyer and Moore, 1998), describes strategies to find structural induction proofs. This includes different ways of using the induction hypothesis (weak and strong fertilization), selection of the induction variable, and the recognition of the need for extra lemmas. More recent research adds the use of rippling as an important technique (Bundy, 2005). To use an automated theorem prover a user should have a thorough understanding of inductive proofs. Because the concept of induction is the learning goal in courses on proof techniques, using such provers in education is in general not very helpful.

Proof assistants such as Tutch (Abel et al., 2001) and Minifn (Osera and Zdancewic, 2013) have been developed for educational purposes. Tutch is based on a high-level proving language, which allows step sizes resembling the steps in a pen-and-paper proof. Minifn tries to integrate functional programming with mathematical induction in a way that students can quickly learn how to use the proof assistant. Although these proof assistants are much easier to use than regular theorem provers, they remain assistants. They do not offer exercises, nor do they provide feedback or hints.

A couple of e-learning tools support learning mathematical induction. In EAsy, an e-assessment system, a student practices with different kinds of proof exercises, using rules and proof strategies from a drop-down menu (Gruttmann et al., 2008). For a problem requiring a proof by induction, the system pre-structures the proof in a base case and an inductive case, and it provides the induction hypothesis. The system performs a selected rule, which ensures that a student cannot make a mistake in applying a rule. Since there are quite a lot of rules and the exercises are non-trivial, a student easily can get lost: in an evaluation 41% of the students mentions having problems in selecting the right rule from the extensive ruleset. Completed proofs are graded automatically, but incomplete proofs have to be graded by hand. EAsy shows a student that she completed a proof or subproof successfully, but does not provide any other feedback.

In the Intelligent Book, a student practices exercises on mathematical induction (Billingsley and Robinson, 2007). This e-learning system uses MathTiles, predefined templates, which have to be completed by the student. An automated theorem prover, Isabelle, is used in the backend to check correctness and provide simplifications. As in EAsy, the system automatically generates the goals for the different cases. The authors state that this is necessary since Isabelle only accepts these goals when they are exactly equal to the goals in the prover. Simplification can be performed automatically, but when, for example, an application of the induction hypothesis is needed, simplification is not accepted. The tool provides hints in some cases, based on teacher scripts.

ComIn-M contains electronic exercise sheets (Rebholz and Zimmermann, 2013). The system is developed as part of the SAiL-M project, and extends an earlier e-assessment tool (Müller and Hiob-Viertler, 2010). These sheets also contain pre-structured mathematical induction exercises, and combine multiple choice exercises about stating the induction hypothesis with open exercises to prove a base case and an inductive case. Students have to state the inductive case before proving it. A nice feature of ComIn-M is the possibility to work in two directions. Feedback and hints are provided automatically.

The website[2] accompanying the textbook Discrete Mathematics: Mathematical Reasoning and Proof with Puzzles, Patterns and Games by Ensley and Winston Crawley (2006) contains a set of interactive exercises in which given a property P, a student first has to deduce $P\ (n+1)$ from $P\ (n)$ for concrete values, and then for an arbitrary value n. In this way, a student gets some intuition behind induction before she proves the inductive case.

We found only two systems addressing structural induction. The most extensive system is Polycarpou's e-book, which is a complete educational environment for learning structural induction. She emphasizes foundational concepts, such as structures, sets and closed sets. A separate chapter introduces inductive definitions. Animations show how these definitions generate inductive sets. Interactivity is restricted to multiple choice, drag and drop, and fill in the blank exercises, which implies that a student does not independently complete an inductive proof.

Stanford's online Introduction to Logic course[3] does offer the possibility to construct complete structural induction proofs in its open online logic course. However, in the course material induction is combined with natural deduction, causing long and abstract proofs. It is possible to ask for a complete solution to an exercise, but the system does not give feedback on mistakes, nor hints on how to proceed. The site also offers a proof assistant that combines structural induction with Hilbert-style proofs.

[2]http://higheredbcs.wiley.com/legacy/college/ensley/0471476021/anim_flash/index.html

[3]http://intrologic.stanford.edu/public/index.php

5.4 Students' problems with structural induction

Students have problems with constructing inductive proofs. Several studies identify misconceptions students have with mathematical induction, such as not recognizing the difference between base cases and inductive cases, and analyze the underlying reasons for these misconceptions (Avital and Libeskind, 1978; Dubinsky, 1986; Ernest, 1984; Harel, 2001; Palla et al., 2012; Pavlekovič, 1998). In her thesis, Polycarpou studies possible causes of problems students have with structural induction (Polycarpou, 2008). Her hypothesis is that a lack of understanding of set theoretical concepts is one of the main causes of these problems. To investigate this hypothesis, she performed an experiment in which students have to answer six questions about a fancy inductively defined language IPO (a fictive programming language). The base elements of this language are lower case characters. There are two inductive rules to construct new words: by concatenating two IPO words by an underscore, and by putting quotes around an IPO word. The first four questions test whether a student understands this definition: the student has to indicate which words are IPO words in a given set of words, give the minimal length of an IPO word, determine whether or not IPO words have a maximum length, and construct an IPO word of length greater than 6. The fifth question prepares for the last question by asking if it is possible to construct a word of length 8. Finally, in the last question students have to tell whether an IPO word can have even length, and in case the answer is no, prove that all IPO words have odd length. Students participating in this experiment are enrolled in a course 'Logic for Computer Science' and they have practiced with mathematical induction in an earlier course on discrete mathematics. As a pre test, students have to answer these exercises two months before the lessons on induction. After these (traditional classroom) lessons a comparable set of exercises is given as a post test.

In the pre test, students particularly experience problems with the exercise on identifying IPO words, the problem about a word with length 8, and the inductive proof. The results on these questions are much better in the post test, but still the inductive proof is too hard for 44% of the students. The author claims that there is a correlation between performance on the first five questions and performance on the inductive proof, but this correlation is not statistically motivated. Instead, she calculates the ratio between students who perform well on both the inductive set exercises and the inductive proof exercise (71%), and the ratio between students who do not perform well on both (83%). She notes that some students find an (incorrect) IPO word with length 8, but manage to prove that all IPO words have odd lengths. Another observation is that some students seem to copy an example treated in the course without the necessary adaptations. These students define inductive cases for negation and conjunction, instead of for the IPO constructors. The conclusion of Polycarpou is that lack of understanding of an inductive definition is indeed a main cause of problems with induction, and that the traditional way of teaching results in procedural knowledge without conceptual understanding for some students.

To investigate whether students entering the master Computer Science at the Open University of the Netherlands, a distance learning university, experience similar problems, we analyze the solutions to a homework assignment of 20 of these students. Before being admitted to the Computer Science master program, students have to take a course in logic. Structural induction is one of the topics in this course. Unlike the students participating in Polycarpou's experiment, almost none of these students has experience with mathematical induction. Furthermore, their background in mathematics is usually weaker than that of bachelor students at a regular university. The homework assignment is optional, but gives extra credits at the exam.

In the first part of the homework assignment (Exercise 1), students have to give an inductive definition of a function *len*, which returns the length of a propositional formula. The next question (Exercise 2) asks to give an inductive definition of a postfix function $*$ that rewrites all conjunctive subformulae $\phi \wedge \psi$ of a formula into the equivalent subformula $\neg(\neg\phi \vee \neg\psi)$. The last question (Exercise 3) asks for an inductive proof of the following property: $len\ (\phi*) \leqslant 3\ len\ (\phi) - 2$ for all formulae ϕ. This assignment differs from Polycarpou's: it does not test the understanding of inductively defined sets, but instead tests the understanding of inductively defined functions (in exercises 1 and 2).

As Polycarpou, we expect correlations between performance on the first two exercises and the last exercise. Since the number of participating students is too low for a statistic test, we perform a similar calculation as Polycarpou. Students who do not receive full points for the first two exercises (11 students), are almost all (10) unable to complete the third exercise. However, from the students who receive full points (9) only 2 manage to complete the inductive proof. We conclude that for these students Exercise 3 is too difficult to complete without help, but most students have fewer problems with the inductive function definitions. Table 5.1 shows the results on the first two exercises. Most students provide a correct definition, but some students add an induction hypothesis to this definition. Since students can make several mistakes, for example, 'no correct cases' and 'an incorrectly added induction hypothesis', the sum of the percentages in Table 5.1 (and also Table 5.3) is more than 100%.

We looked further into mistakes students made in the proof exercise. As shown in Table 5.3, the inductive part of the proof most often goes wrong, and the most common error is assigning a fixed length to a compound formula (for example, $len\ (\phi \wedge \psi) = 3$). The inductive definition does not seem the bottleneck, since 70% of the students (see the columns 'correct' + 'correct use of IH, but incomplete' + 'correct ind def, no use of IH' in Table 5.3) apply these definitions in their proof. The assumption $\phi* = \phi$ made by some students could be a symptom of conceptual misunderstanding of an inductive definition, but could also be used because a student does not know how to apply the induction hypothesis. We conclude that although problems with inductive definitions may certainly play a role

Tabel 5.1: Results for exercises 1 and 2 of the homework assignment

(a) Exercise 1

solution	(N = 20)
correct base and ind. cases	80%
only correct base case	5%
one missing case	5%
no correct cases	10%
incorrectly added IH	15%

(b) Exercise 2

solution	(N = 20)
correct	60%
no base case	30%
incorrect	10%

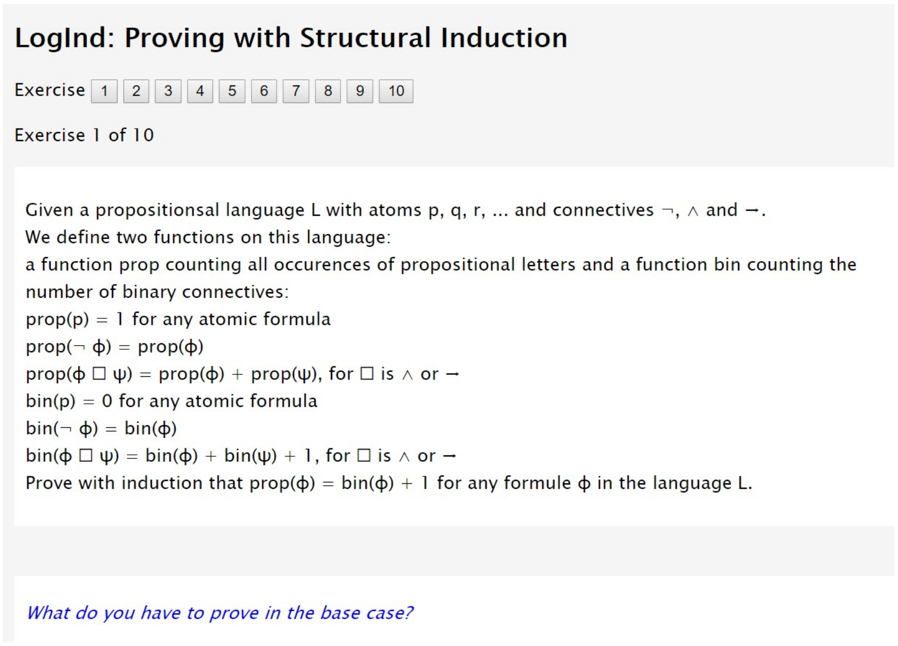
Tabel 5.3: Results and mistakes for Exercise 3 of the homework assignment (N = 20)

solution	base	IH	induction
correct	75%	50%	15%
IH only for one formula ϕ	–	20%	–
assumption ϕ and ψ atomic	–	–	35%
assumption $\phi * = \phi$	–	–	15%
correct use of IH, but incomplete	–	–	10%
correct ind def, no use of IH	–	–	45%
IH as goal	–	–	10%
verbal intuitive argument	–	–	5%

in the results in the inductive proofs, the understanding of the role of the induction hypothesis and the way this hypothesis can be used is perhaps a more important cause of problems. We think that the remedy of Polycarpou (an intelligent tutoring system that mainly focuses on theoretical foundations) probably will not be the best solution for our students, since the concept of inductive sets does not seem to be their main problem, and her approach might be too theoretical. Instead we concentrate on an e-learning tool that guides students interactively through the construction of an inductive proof.

5.5 LogInd, a tool for teaching structural induction

This section describes LogInd, a tool that supports students with constructing inductive proofs. Experience with other intelligent tutoring systems for logic (LogEx for rewriting propositional formulae (Lodder et al., 2019), and LogAx for Hilbert-style axiomatic proofs (Lodder et al., 2017)) shows that students benefit from a system where they can enter solutions stepwise, get feedback after each step, and can ask for a hint or next step at any moment, or receive a worked-out solution.



LogInd: Proving with Structural Induction

Exercise 1 2 3 4 5 6 7 8 9 10

Exercise 1 of 10

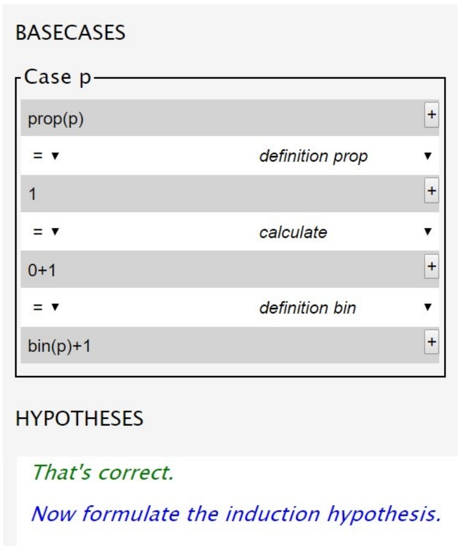
Given a propositionsal language L with atoms p, q, r, ... and connectives ¬, ∧ and →.
We define two functions on this language:
a function prop counting all occurences of propositional letters and a function bin counting the number of binary connectives:
prop(p) = 1 for any atomic formula
prop(¬ ϕ) = prop(ϕ)
prop(ϕ □ ψ) = prop(ϕ) + prop(ψ), for □ is ∧ or →
bin(p) = 0 for any atomic formula
bin(¬ ϕ) = bin(ϕ)
bin(ϕ □ ψ) = bin(ϕ) + bin(ψ) + 1, for □ is ∧ or →
Prove with induction that prop(ϕ) = bin(ϕ) + 1 for any formule ϕ in the language L.

What do you have to prove in the base case?

Figuur 5.1: Starting the first exercise in LogInd

The possibility to add proof steps both backwards and forwards in these systems resembles the way an exercise is solved with pen and paper. We use the same approach in LogInd. Since some students have hardly any idea how to start an inductive proof, LogInd offers guidance in structuring the proof. We also want students to be aware of the kind of steps they perform in the proof, for example, applying the induction hypothesis or an inductive definition of a given function. Hence, LogInd asks for a justification at each step. Students do not have to justify simple calculation steps, such as distributing a multiplication over an addition, and we allow calculation steps at different levels of granularity. Therefore, LogInd checks calculations by normalizing the submitted expression.

LogInd guides a student through a proof by structuring the proof in three parts: a proof of the base case, stating the induction hypotheses, and a proof of the inductive cases. After presenting the exercise, LogInd asks the student first to state what is to be proven in the base case, see Figure 5.1. If this is correct the student is asked to complete the proof of the base case, and to continue with stating the induction hypotheses, see Figure 5.2. For the inductive cases, LogInd again first asks what the different cases are, and what has to be proven in these cases. A complete proof is shown in Figure 5.3.


BASECASES

Case p

prop(p)

= ▾ *definition prop* ▾

1

= ▾ *calculate* ▾

0+1

= ▾ *definition bin* ▾

bin(p)+1

HYPOTHESES

That's correct.

Now formulate the induction hypothesis.

Figuur 5.2: Guidance after the base case is finished

LogInd uses a domain reasoner to provide hints, next steps, feedback, and complete solutions. A domain reasoner is an expert module that performs all reasoning about the domain (Goguadze, 2010). Thus far, LogInd only offers exercises about properties of a propositional language. We introduce the term 'counting function' to describe the set of exercises that can be used in LogInd.

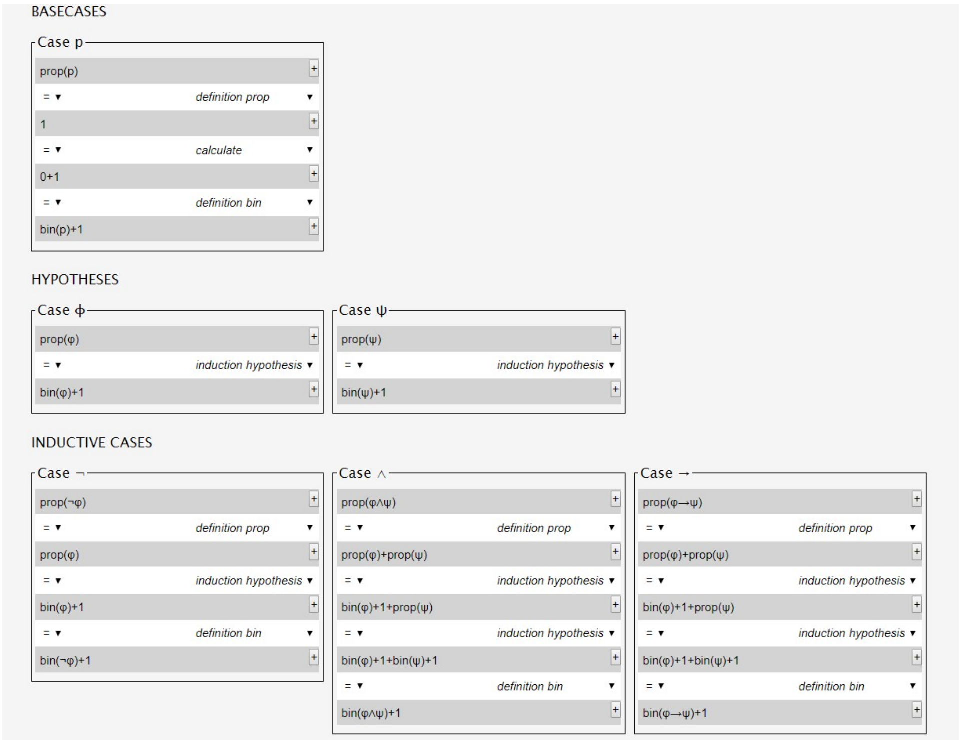
A counting function *count* is an inductively defined function such that

$$\begin{array}{lll} count\ (p_i) & = c_i, & c_i \in \mathbb{N}, \\ count\ (\neg\phi) & = a + b \cdot count\ (\phi), & a, b, \in \mathbb{N} \\ count\ (\phi \square \psi) & = a_\square + b_\square \cdot count\ (\phi) + c_\square \cdot count\ (\psi), & a_\square, b_\square, c_\square \in \mathbb{N} \end{array}$$

where p_i is a propositional letter, and $\square$ a binary connective.

The properties that have to be proven take the following form: $P_1\ (\phi)$ *comp* $P_2\ (\phi)$, where *comp* is a comparator $(=, <, \leqslant, >, \geqslant)$ and $P_i\ (\phi), i = 1, 2$ is either a truth value, number or formula:

- if $P_i\ (\phi)$ is a truth value, it is an expression $V\ (g\ (\phi))$ or a constant where V is a valuation and g an inductive function from the language L to L;
- if $P_i\ (\phi)$ is a natural number, the right-hand side is a linear combination of expressions $f\ (g\ (\phi))$ where f is a counting function and g an inductively defined function from L to L, the left-hand side is a single expression $f\ (g\ (\phi))$;
- if $P_i\ (\phi)$ is a a formula, it is equal to an expression $g\ (\phi)$ where g is an inductively defined function from L to L;


BASECASES

Case p
prop(p)
= ▾ definition prop ▾
1
= ▾ calculate ▾
0+1
= ▾ definition bin ▾
bin(p)+1

HYPOTHESES

Case ϕ
prop(φ)
= ▾ induction hypothesis ▾
bin(φ)+1

Case ψ
prop(ψ)
= ▾ induction hypothesis ▾
bin(ψ)+1

INDUCTIVE CASES

Case ¬
prop(¬φ)
= ▾ definition prop ▾
prop(φ)
= ▾ induction hypothesis ▾
bin(φ)+1
= ▾ definition bin ▾
bin(¬φ)+1

Case ∧
prop(φ∧ψ)
= ▾ definition prop ▾
prop(φ)+prop(ψ)
= ▾ induction hypothesis ▾
bin(φ)+1+prop(ψ)
= ▾ induction hypothesis ▾
bin(φ)+1+bin(ψ)+1
= ▾ definition bin ▾
bin(φ∧ψ)+1

Case →
prop(φ→ψ)
= ▾ definition prop ▾
prop(φ)+prop(ψ)
= ▾ induction hypothesis ▾
bin(φ)+1+prop(ψ)
= ▾ induction hypothesis ▾
bin(φ)+1+bin(ψ)+1
= ▾ definition bin ▾
bin(φ→ψ)+1

Figuur 5.3: The solution of the exercise of Figure 5.1 in LogInd

- valuations V may have predefined properties such as $V\ (p) = 1$, but if the comparator in the theorem is an equality, these properties may also only make use of equalities (since in our proof system it is not possible to prove an equality from inequalities).

The restriction in the second option that the left-hand side is a single term $f\ (g\ (\phi))$ while the right-hand side may contain a linear combination of terms is not a real restriction, since a statement where both the left- and right-hand side contain linear expressions can always be rewritten into this form. The restriction will enable us to treat the induction hypothesis as a rewrite rule. Examples of exercises in this format are:

- Let L be a propositional language with connectives $\wedge$ and $\vee$. Let *ValA* and *ValB* be two valuations such that *ValA* $(p) \leqslant$ *ValB* (p) for any atomic formula p. Then *ValA* $(\phi) \leqslant$ *ValB* (ϕ) for any formula ϕ in the language L.
- Let L be a propositional language with connectives $\wedge$ and $\vee$, and L' the extension of L with negation $\neg$. The function *star* from L to L' replaces

every atom by its negation, conjunctions by disjunctions and disjunctions by conjunctions. The function *length* returns the length of a formula. The following holds: $length(star\ (\phi)) \leqslant 2 \cdot\ length(\phi)$

– Let L be a propositional language with connectives $\neg$, $\wedge$, $\vee$ and $\rightarrow$. Function f replaces every conjunctive subformula $\phi \wedge \psi$ by $\neg(\neg\phi \vee \neg\psi)$, and function g replaces every implicative subformula $\phi \rightarrow \psi$ by $\neg\phi \vee \psi$. Then $f\ (g\ (\phi)) = g\ (f\ (\phi))$ (where '=' means syntactically equal) for any formula ϕ.

This class of problems offers sufficient possibilities for relatively simple exercises, where students can get acquainted with inductive proofs. In the next section we show that for this class we can generate solutions, hints and next steps, without the need for advanced techniques as used by automatic theorem provers.

5.6 Generation of solutions, hints and next steps

In general, automatic proof generation for induction problems is undecidable (Aubin, 1979; Bundy, 2001). Problems that might seem easy, such as the proof for associativity of list concatenation for a single list $((l :: l) :: l = l :: (l :: l))$, already need advanced methods to be proven automatically (in this case generalization of the first occurrence of l) (Bundy, 2001). Automatic theorem provers use sophisticated techniques such as rippling and lemma generation to solve inductive problems (Bundy, 2005, 2001). It is not our goal to teach students these techniques, and by restricting ourselves to the class of problems described in the previous section, we only need a straightforward strategy to solve such exercises.

The problem-solving strategy we use is part of our domain reasoner. The strategy first uses the definition of the language to decide what has to be proven in the base cases, and which inductive cases have to be treated. We only allow a single formula variable in a property, so we do not have to ask which variable will be used for induction. Inductive functions are represented as rewrite rules, just as the induction hypothesis. Apart from some technical details, the strategy first applies the inductive definitions of the functions occurring in the statement. The strategy rewrites both the right-hand and left-hand side of the statement. Hence, the strategy supports the possibility to complete a subproof by working in two directions. In case the inductive function has natural numbers as the codomain, the next step (if necessary) is a distribution of the multiplication such that the left-hand side and right-hand side become linear combinations of terms that occur in the induction hypothesis. Now we can apply the induction hypothesis to occurrences of the left-hand side of this hypothesis in the left-hand side formula in the proof. After this application only some normalizing elementary calculations might be needed to complete the proof. For our restricted class of problems this strategy always finds a solution. We provide a sketch of a proof of this statement in Appendix 5.A. Students will not always follow this strategy. For example, they might apply the induction hypothesis before rewriting the right-hand side of a statement applying the inductive definitions, or vary in the calculations. Also in these cases, LogInd continues with applying the strategy. Hence, a next step can be provided as long as the strategy's rules are applicable. After application of these rules, normalization is sufficient to complete the proof. We expect that almost all student steps will be such that LogInd can indeed provide a hint or next step based on the student solution. The evaluation described in Section 5.8 gives evidence for this claim.

Our strategy differs from the method advocated by Bundy (2001). The difference is in the way we use the induction hypothesis. Bundy recommends strong fertilization: the isolation of the induction hypothesis in an inductive case and replacement of this hypothesis by True. Our strategy uses weak fertilization: substituting the left-hand side of the induction hypothesis by the right-hand side. Bundy advises the use of strong fertilization since the use of weak fertilization generally results in longer and more complicated proofs. For our restricted class of problems, this is not the case. Since most pen-and-paper proofs use weak fertilization, we also use weak fertilization in our strategy.

5.7 Constraints and feedback

From the analysis of the homework assignment, we expect our students to make various kinds of mistakes while practicing with LogInd. Examples of potential mistakes are:

- treating metavariables ϕ and ψ as atoms, for example, resulting in the rewriting of $length(\phi \wedge \psi)$ into 3 in the proof of an inductive case;
- omission of a case, for example negation;
- use of only $=$ and $\leqslant$ when a statement $P_1\ (\phi) < P_2\ (\phi)$ has to be proven;
- forgetting to state the induction hypothesis before using it.

In our other tutoring systems for logic we use buggy rules to generate feedback in case a student makes a mistake. Buggy rules typically relate to mistakes on the level of single steps. An example of such a buggy rule is forgetting to change a disjunction in a conjunction in an application of DeMorgan while rewriting a formula into normal form. Mistakes in an inductive proof can be on the level of a step, for instance rewriting $length(\phi \wedge \psi)$ to 3, but also on the level of a subproof. An example of an error in a subproof is using $\leqslant$ between each of the steps when the goal is to prove an equality. In this case, each of the steps is correct, but the overall relation between the first and last line of the proof is $\leqslant$ instead of $=$. An example of an error on the level of the whole inductive proof is the omission of a case. These mistakes are easily formulated as constraint violations. For example, the composition of the relations between the lines in a proof should imply the relation between left-hand side and right-hand side in the theorem that is proven. We think that constraints can be put to good use for this domain.

Ohlsson (1994) first described the role of constraints in learning. Mitrovic used the concept in the development of an SQL tutor (Mitrovic, 2012) and many other tutors. Constraints characterize correct solutions by providing a relevance condition and a satisfaction condition: if the relevance condition holds, the solution should satisfy the satisfaction condition. Some important reasons for using constraints in the development of tutoring systems are that constraints partially play the role of buggy rules, the construction of which is very time consuming, and that constraints can also be used to give feedback if student solutions diverge from model solutions or are partial (Mitrovic, 2012; Mitrovic et al., 2007).

LogInd uses constraints to provide feedback and guidance. Heeren and Jeuring (2014) describe the diagnose service used by a domain reasoner to provide feedback. Figure 5.4 is based on this description, and shows how we incorporate constraints in the diagnosis. Our diagnose service receives a (partial) student solution and checks whether or not this submission violates a set of constraints. We divide the constraints in constraints on the level of steps, on the level of subproofs, and on the level of proofs.

- Constraints on the level of steps check:
  - if the rewriting of a line (for example the application of an inductive definition) is correct;
  - if the induction hypothesis is applied correctly;
  - if comparators ($=$, $<$ ..) are used correctly in a single step;
  - if the justification is correct.
- Constraints on the level of subproofs check:
  - if the first and last line of each subproof are instances of the left-hand side respectively right-hand side of the theorem;
  - if these instantiations are valid cases (atomic base cases, inductive cases only for connectives in the language);
  - if the induction hypothesis (when present) is correctly formulated;
  - if comparators ($=$, $<$, ..) are used correctly at the level of subproofs (i.e. if the composition of the comparators in the subproof equals the comparator in the theorem).
- Constraints on the level of complete proofs check:
  - if all inductive cases correspond to connectives in the language;
  - if the induction hypothesis is stated before use, with the same metavariables;
  - if an inductive case or special base case is missing.

To check a proof at the level of a step, the domain reasoner compares the student submissions with the result of the application of possible rules, and accepts the student submission if it is similar to one of these results. Here, a simple normalizing calculation transforms the submission and the generated result into the same expression. For example, application of the induction hypotheses in the example of

Figure 5.3 results in $bin\ (\phi) + 1 + bin\ (\psi) + 1$, but a submission $bin\ (\phi) + bin\ (\psi) + 2$ is also accepted.

If no constraint is violated, the domain reasoner compares the new submission with the previous (last correct) submission, and determines if these are similar. If these submissions are similar, the domain reasoner gives feedback about this. A submission that is not similar may follow the implemented strategy, which is diagnosed as 'expected'. When the step does not follow the strategy, but is recognized by the domain reasoner, the diagnosis is a 'detour'. This happens, for example, when a student starts with completing the inductive case for implication before negation. The interface will tell the student that this step is correct, and the student can continue with the exercise. Since there are no violations, every step in the submission is already recognized, which means that the last option (no rule detected) only happens when the student submission contains more than one new line. Again, the interface will provide a message 'this is correct'.

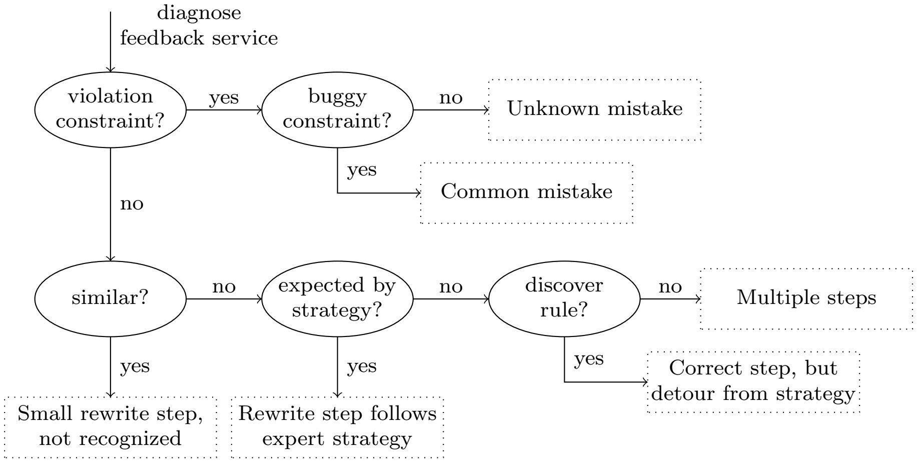
Our experience is that the diagnosis 'failure' of a constraint is not enough to provide informative feedback. For example, a constraint on the exercise in Figure 5.1 could be that the first line of an inductive case should be an instantiation of the left-hand side of the theorem $prop\ (\phi) = bin\ (\phi) + 1$, with ϕ substituted by $\neg\alpha$, $\alpha \wedge \beta$ or $\alpha \rightarrow \beta$, where $\alpha \neq \beta$ and $\alpha, \beta \in \{\phi, \psi\}$. Now a student can violate this constraint in different ways, for example by

- using a connective that is not in the language;
- instantiating with an atomic formula;
- instantiating with a metavariable that is not used in the induction hypothesis;
- introducing an expression that is not an instantiation at all.

Each of these violations stem from a different misconception, and we want to give different feedback messages in each case. We solve this by specifying failure messages for different constraints, and call this use of constraints 'buggy constraints' as proposed by Kodaganallur et al. (2005). One of the problems of the use of constraints as mentioned by Kodaganallur et al. (2005) and Mitrovic (2012) is the violation of two or more constraints at the same time. We solve this by ordering the constraints: for example, constraints about instantiations get a higher priority than constraints about the application of a rule.

Apart from 'strong' constraints that may not be violated, we also use soft constraints to guide a student through a proof. These constraints check for each of the subproofs if they are introduced and if they are finished. After a diagnosis, the user interface can call the feedback service 'constraints', which reports the status of each of the subproofs. The result can be used to provide a message such as: 'the base case is finished, continue with the formulation of the induction hypothesis'.

The way we use constraints in LogInd differs in some aspects from the original use. One difference is the fact that LogInd has a strategy which produces solutions, and while checking the next step of a student, LogInd first tries to recognize this step. LogInd can hence be conceived as a constraint-based solver as described



Figuur 5.4: Structure of the diagnose feedback service: the incorporation of constraints is new

in Kodaganallur et al. (2005). Moreover, we use constraints not only to provide feedback on errors, but also to guide a student.

5.8 Evaluation

In April 2019 we performed a small pilot experiment with a group of 15 students taking an online logic course in preparation of admission to the master in Computer Science at the Open University of the Netherlands. Before the experiment, these students handed in the homework assignment described in Section 5.4. The experiment consisted of an online instruction about the use of LogInd, followed by the possibility to practice. During the experiment students could ask questions by using the chat functionality of the learning environment. The questions that we would like to answer with this experiment are:

1. does LogInd behave as expected, i.e., provide hints and next steps, and give a correct diagnosis?
2. what kind of problems do students have while working with LogInd?
3. how do students use LogInd?
4. can we see effects of the use of LogInd in the way students perform pen-and-paper exercises?

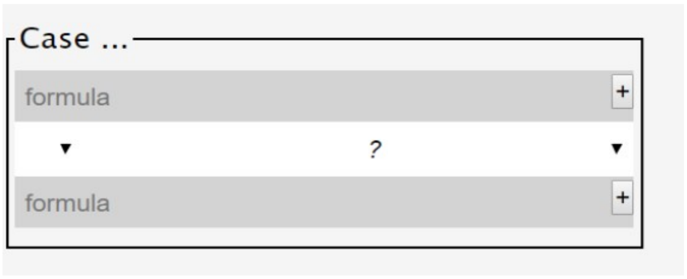
In Section 5.6 we claim that we can provide a hint or next step at any moment, also if a student does not follow the preferred strategy by LogInd. We analyze the loggings to check this claim. From the 1612 calls obtained from student interactions (a diagnosis, a hint, a next step, or a full solution), 398 calls ask for a hint or a next step. In 25 cases (6%) LogInd cannot provide such a step. Further analysis shows that since students repeat their call several times, this only happens in five different cases (1%). In all of these cases LogInd could not provide a hint or next step because it diagnoses the exercise as 'ready'. This was a bug, resulting from a use of LogInd that we had not anticipated: some students did not start a base case with the statement that they have to prove, but they submit the first two lines of a proof of the base case, for example the subproof:

$$\begin{array}{l} \mathit{prop}\ (p) \\ \ = (\mathit{definition\ prop}) \\ 1 \end{array}$$

In such a case LogInd decided that this step was correct and that this subproof was finished since all steps are motivated, without checking whether indeed the base case has been proven. When a student asked for a hint after the other subproofs were (correctly) finished, LogInd could not provide such a hint. We repaired this bug, and since this was the only reason that no hint or next step was available, we expect that LogInd now indeed provides this kind of feed forward (i.e. hints and next steps) in all circumstances.

We use the remarks made by students in the chat during the experiment and the loggings to answer the second question. From these remarks and the loggings we learned that students had quite a lot of problems with the interface. Students should start an exercise with a response to the question 'what do you have to prove in the base case?'. They should fill in their answer in a template as shown in Figure 5.5. For the first exercise, the exercise of the example in Section 5.2, this means that on the first line, a student should enter $\mathit{prop}\ (p)$, then choose the equality sign (=) from the drop-down list and enter the right-hand side $\mathit{bin}\ (p) + 1$ in the bottom line. In their first attempt, none of the participating students entered this first step correctly. Ten students did not realize that they first had to answer this question before completing the proof or did not know what to prove. They provided answers such as for example $\mathit{prop}\ (p) = 1$ or $\mathit{prop}\ (\phi) = \mathit{bin}\ (\phi) + 1$. Three students entered the whole statement $\mathit{prop}\ (p) = \mathit{bin}\ (p) + 1$ on the first line, one student added a wrong justification (definition of the inductively defined function *prop*), and one student replaced the ? by an empty string, which at that time was not accepted by LogInd.

A second source of problems was the use of the send button. A student only gets feedback after clicking this button. If a student enters several lines before clicking this button, it was hard to find the place where the feedback referred to, especially since at the time of the experiment we were still developing the constraints. So a



Figuur 5.5: Template for starting a base case

student might receive the message ‘this step is not correct’ without a clue which step should be repaired. Also, when a student asked for a hint after receiving an error message, this hint was based on the latest correct submission. Hence, this hint might relate to the application of a rule in (for example) a base case, while the student was working on an inductive case.

To answer the third question, how do students use LogInd, we also looked at the loggings. The problems described in the previous section combined with the conceptual and technical problems students have with induction caused a high hint and next step use (25% of the student interactions in the loggings). However, students did enter steps themselves and asked for a diagnosis (1078 requests, 67%), where 619 steps were diagnosed as correct. Half of the participating students were able to construct (parts of) a subproof, some of them using hints or next steps in between, but the other students had too many problems with the interface. Students who solved the homework assignment before the experiment, could use LogInd without too many problems.

To answer the last question, we analyzed resubmissions of students’ homework. Students whose homework assignment was not correct had to submit the assignment again. We hoped to use these improved assignments as a kind of post test. However, three students submitted their corrections before the experiment, and one student did not submit a new version. The results of the remaining seven students can be found in Table 5.4. We use the same characterizations of solutions as in Table 5.3, except for the solution label ‘incorrect use of IH’, which did not occur in the first submissions. The first column shows the results on the first submissions of the homework assignment by the group of students who practiced with LogInd and submitted a new version after the experiment. The results of this second submission is shown in the second column. The last two columns show the same data for the group of students who did not practice with LogInd. The second submission is in both groups better than the first attempt. The number of students is too low to conclude if practicing with LogInd has more effect than the comments by the teacher on the first attempt.

Solution	LogInd (N = 7)		No LogInd (N = 4)	
	1st	2nd	1st	2nd
Correct	–	2	–	–
Assumption ϕ and ψ atomic	3	1	4	1
Assumption $\phi * = \phi$	1	3	0	1
Correct use of IH but incomplete	1	–	–	–
Incorrect use of IH	–	1	–	–
Correct ind def, no use of IH	3	5	4	2

Tabel 5.4: Results and mistakes in the first submission of homework assignment 3 and in the improved second submission (number of students)

5.9 Conclusion and future work

We have discussed the design of a tutoring system for learning how to prove statements about inductively defined discrete structures. As far as we know, LogInd is the first such tutoring system. A student constructs her proof stepwise, and LogInd provides help (hints, next steps, and elaborate feedback on errors) at each step. A pilot evaluation showed that LogInd indeed can provide help in almost all situations. Half of the students in the experiment could construct (parts of) a proof by themselves, but the other half had too many problems with the interface and the exercises. The number of students who participated in the experiment was too low to decide whether students indeed learn by using LogInd. We noticed that homework submissions by students who had practiced with LogInd were more clearly structured, and contained more justifications of different steps.

In a next experiment we will evaluate the constraints: is the information in a feedback message correct, and do these messages help students to correct their submission? We will test an alternative interface, where students can enter their steps in a text field, and we will also compare a guided version with a non-guided version. In the future we might also incorporate exercises about induction in other domains, such as for example lists or functional programs.

Acknowledgments

We thank Aad van Lieburg and Anja Paalvast for their work on the student interface, and our students for their permission to use their solutions in our research, and their participation in the experiment.

5.A Sketch of a completeness proof for the strategy used by LogInd

We give a sketch of the completeness proof for the strategy used by LogInd. First we specify the class of functions that are used in in our exercises for transforming a formula in another formula. We call these inductively defined functions acceptable.

Definition 1. *An inductively defined function g from the propositional language L to L is acceptable if the inductive definition can be written in the following form:*

- $g\ (p_i) = \phi_i$ *for atomic formula* p_i
- $g\ (\neg\phi) = [g\ (\phi)\ /\ s]\ \psi$
- $g\ (\phi_1 \Box \phi_2) = [g\ (\phi_1)\ /\ s_1, g\ (\phi_2)\ /\ s_2]\ \psi_\Box$

The definition states that the inductive cases are obtained by substituting a variable s by $g\ (\phi)$ in a formula ψ, or the variables s_1 and s_2 by $g\ (\phi_1)$ and $g\ (\phi_2)$ in a formula $\psi_\Box$. All inductive definitions of functions from L to L can be written this way. For example, the function *star* in the second example in Section 5.5 can be defined by:

$$\begin{aligned}
g\ (p_i) &= \neg\ p_i\\
g\ (\neg\phi) &= [g\ (\phi)\ /\ s]\ (\neg s)\\
g\ (\phi_1 \wedge \phi_2) &= [g\ (\phi_1)\ /\ s_1, g\ (\phi_2)\ /\ s_2]\ (s_1 \vee s_2)\\
g\ (\phi_1 \vee \phi_2) &= [g\ (\phi_1)\ /\ s_1, g\ (\phi_2)\ /\ s_2]\ (s_1 \wedge s_2)
\end{aligned}$$

In our proof we need the following lemma:

Lemma 1. *For any counting function f and formulae ϕ and ψ, $f\ ([\phi\ /\ p]\ \psi)$ is a linear expression in $f\ (\phi)$.*

Bewijs. Proof with induction on ψ:
Since f is a counting function, there exists constants c_i, a, b, $a_\Box$, $b_{1\Box}$, and $b_{2\Box}$ such that

$$\begin{aligned}
f\ (p_i) &= c_i,\\
f\ (\neg\phi) &= a + b \cdot f\ (\phi),\\
f\ (\phi_1 \Box \phi_2) &= a_\Box + b_{1\Box} \cdot f\ (\phi_1) + b_{2\Box} \cdot f\ (\phi_2)
\end{aligned}$$

For atomic formulae ψ, $f\ ([\phi\ /\ p]\ \psi)$ is either $f\ (p_i)$ (a constant) or $f\ (\phi)$ and hence linear in $f\ (\phi)$.
For the inductive cases we assume that $f\ ([\phi\ /\ p]\ \psi_1) = c_1 + d_1 \cdot f\ (\phi)$ and $f\ ([\phi\ /\ p]\ \psi_2) = c_2 + d_2 \cdot f\ (\phi)$.
Then:

$$\begin{aligned}
&f\ ([\phi\ /\ p]\ (\neg\psi_1))\\
=\ &f\ (\neg[\phi\ /\ p]\ \psi_1)
\end{aligned}$$

$$\begin{aligned}
&= a + b \cdot f\ ([\phi \,/\, p]\ \psi_1) \\
&= a + b \cdot (c_1 + d_1 \cdot f\ (\phi)) \\
&= a + b \cdot c_1 + b \cdot d_1 \cdot f\ (\phi)
\end{aligned}$$

And:

$$\begin{aligned}
&\ f\ ([\phi \,/\, p]\ (\psi_1 \Box \psi_2)) \\
&= f\ ([\phi \,/\, p]\ \psi_1 \Box [\phi \,/\, p]\ \psi_2) \\
&= a_\Box + b_{1\Box} \cdot f\ ([\phi \,/\, p]\ \psi_1) + b_{2\Box} \cdot f\ ([\phi \,/\, p]\ \psi_2) \\
&= a_\Box + b_{1\Box} \cdot (c_1 + d_1 \cdot f\ (\phi)) + b_{2\Box} \cdot (c_2 + d_2 \cdot f\ (\phi)) \\
&= a_\Box + b_{1\Box} \cdot c_1 + b_{2\Box} \cdot c_2 + (b_{1\Box} \cdot d_1 + b_{2\Box} \cdot d_2) \cdot f\ (\phi)
\end{aligned}$$

□

In the same way we can prove that for any counting function f and formulae ϕ_1, ϕ_2 and ψ, $f\ ([\phi_1 \,/\, s_1, \phi_2 \,/\, s_2]\ \psi)$ is a linear expression in $f\ (\phi_1)$ and $f\ (\phi_2)$. We omit the proof. Using Lemma 1 we can prove the next lemma:

Lemma 2. *For any counting function f and acceptable inductively defined function g, after applying g*

- *$f\ (g\ (p_i))$ is a constant for atomic formulae p_i*
- *$f\ (g\ (\neg\phi))$ is a linear expression in $f\ (g\ (\phi))$*
- *$f\ (g\ (\phi_1 \Box \phi_2))$ is a linear expression in $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$*

Bewijs. For atomic formulae, $g\ (p_i)$ is a propositional formula, and hence, $f\ (g\ (p_i))$ is a constant.
Since g is acceptable, there exists a formula ψ and variable s such that $f\ (g\ (\neg\phi)) = f\ ([g\ (\phi) \,/\, s]\ \psi)$, which is linear in $f\ (g\ (\phi))$ according to Lemma 1.
In the same way: there exists a formula $\psi_\Box$ and variables s_1 and s_2 such that $f\ (g\ (\phi_1 \Box \phi_2)) = f\ ([g\ (\phi_1) \,/\, s_1, g\ (\phi_2) \,/\, s_2]\ \psi_\Box)$, which is linear in $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$ by the generalization of Lemma 1. □

Theorem 1. *The strategy used by* LogInd *can generate an inductive proof for any correct statement of the form $P_1\ (\phi)$ comp $P_2\ (\phi)$, where comp is a comparator ($=, <, \leqslant, >, \geqslant$), $P_2\ (\phi)$ is a linear combination of expressions $f_i\ (g_i\ (\phi))$, and $P_1\ (\phi)$ is a single expression $f\ (g\ (\phi))$, for which f and f_i are counting functions, and g and g_i are inductively defined functions from L to L.*

Bewijs. The strategy starts with the application of the inductively defined functions. We first consider the base case. After the application of inductively defined functions, the left-hand side of the equation is a constant and the right-hand side a linear combination of constants, which is rewritten into a single constant using arithmetic, by Lemma 2. A number comparison suffices to check if the base case holds. In the inductive cases, the application of the inductively defined function in the left-hand side results in a linear expression in $f\ (g\ (\phi))$ (case negation) or

$f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$ (case binary connective). The right-hand side is a linear combination of linear expressions in $f_i\ (g_i\ (\phi))$ or in $f_i\ (g_i\ (\phi_1))$ and $f_i\ (g_i\ (\phi_2))$. The next step in the algorithm is the application of the induction hypothesis. Replacing occurrences of $f\ (g\ (\phi))$ or $f\ (g\ (\phi_1))$ and $f\ (g\ (\phi_2))$ by the right-hand side of the induction hypothesis results in another linear combination of $f_i\ (g_i\ (\phi))$ or $f_i\ (g_i\ (\phi_1))$ and $f_i\ (g_i\ (\phi_2))$. Normalizing both left-hand side and right-hand side now suffices to prove the inductive cases. $\square$

6 Epilogue

6.1 Conclusion

In this thesis we developed ITSs for various topics in logic: rewriting propositional formulae in normal form and proving equivalence using rewriting, Hilbert-style axiomatic proofs, and structural induction. We tried to answer the following research questions about ITSs for these topics:

R1 How can we describe the expert knowledge of these topics in a domain reasoner?

R2 How can we generate feedback and feed forward?

R3 What is the effect of the use of the designed tools in logic education?

Here we summarize the results.

To answer the first question (**R1**) we used of the Ideas framework in which rules and solution strategies can be represented. We perceive the exercises in our ITSs as rewriting problems. Here, the term 'rewriting' can be used for the rewriting of a single formula, but also for the rewriting of a partial proof. Our description uses all the five knowledge components from (Heeren and Jeuring, 2020):

- rules, a collection of allowed rewriting steps
- problem-solving procedures or strategies
- normal forms, to decide whether different formulae are equivalent, for example associative variants such as $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$
- buggy rules, a collection of common mistakes
- constraints, for checking properties

Most of the ITSs built on top of Ideas rewrite relatively small steps, as for example a rewriting in LogEx by an application of DeMorgan. In such ITSs, the backend only has to keep the last rewriting of a formula or pair of formulae. LogAx and LogInd differ from these ITSs since in LogAx and LogInd the rewriting steps are defined on the complete partial solution instead of the last entered formula. To represent axiomatic proofs we use DAMs, and inductive proofs are hierarchically structured. To describe solution procedures, we make use of the strategy language of Heeren et al. (2010), which we extended with amongst others a preference operator and the possibility to generate solutions dynamically. Whereas the generation of normal forms, equivalence proofs and inductive proofs is rather straightforward, the generation of axiomatic proofs is a two-step process, consisting of the construction of the DAM and the distillation of a linear proof from this DAM.

We use the solution strategy to provide feed forward (**R2**). In this way, we can produce a hint about the next step to perform, perform a next step, or provide a complete solution directly. For the axiomatic exercises in LogAx, we provide also a subgoal as a reaction to a hint request. To generate these, we keep track of a stack of subgoals during the distillation of a linear proof. We defined buggy rules to provide feedback about common mistakes in LogEx and LogAx. This is feedback at the step level. Since the representation in LogEx only consists of the last rewritings, we cannot provide feedback concerning a whole or partial solution. For example, we cannot detect that a solution contains a cycle. We use constraints to provide feedback on the level of subproofs or proofs level in LogInd, and use these constraints also to guide the student with unsolicited feed forward about the structure of an inductive proof.

To evaluate the use of our logic tools (**R3**), we performed several pilot experiments with the three ITSs and one larger scale experiment with LogEx. In this last experiment we did not find statistically significant differences between students who worked with a version with full functionality and with restricted functionality, but the first group performed significantly better on the exam than students who did not participate. Analysis of the loggings provided insight in the way students use the tool. Probably an experiment where students get more time to practice with our tools is needed to measure differences in learning effects.

6.2 Future work

In the previous chapters we already mentioned several directions for future work directly related to the subject of the chapter. We will not repeat these here, but give some ideas for other directions of future research.

Extensions to predicate logic

LogEx and LogAx are restricted to propositional logic. Students using these tools express an interest in extended versions covering also predicate logic. A natural extension of LogEx would be the rewriting of predicate logic formulae in prenex normal form (a form with all quantifiers in front of the formula). We expect that it will be possible to define rewriting rules and buggy rules more or less in the same way as for the rewriting of propositional formulae, where special care will be needed to formulate conditions in, for example, the rewriting of $\forall x\ \phi \vee \psi$ in $\forall x\ (\phi \vee \psi)$. Recognizing whether a next step is equivalent to a previous one, in case a student combines several steps, will be harder, but may be possible in a decidable fragment of predicate logic.

Bolotov extends his algorithm for generating natural deduction proofs to predicate logic (Bolotov et al., 2005). With an adaptation of this part of the algorithm, an extension of LogAx to predicate logic should be possible.

Other topics in logic

Other directions of future research are different topics in logic. An extension of LogEx for predicate logic could serve as a basis for general resolution; skolemization can be defined as a rewrite rule and, for example, the algorithm of Martelli-Montanari can be implemented for unification (Martelli and Montanari, 1982). It would also be interesting to develop ITSs for modal logic or Hoare calculus.

Other ways to support students

Thus far, we have not made use of student models. Adding a student model will extend the possibilities for providing personalized feedback, for example if a student makes the same mistake more often, or systematically uses a non-efficient strategy to solve an exercise. With a student model, it will also be possible to offer exercises tailored to the student such as exercises that need a certain rule when a student has difficulty with this rule, or more or less difficult exercises. This would also involve classifying exercises. Another possibility will be to personalize the amount and type of feedback, or to allow advanced students to perform more steps at once.

Samenvatting

Is de volgende redenering correct?

> *Sommige medewerkers van de Open Universiteit houden niet van computers.*
>
> *Alle docenten bij de vakgroep Informatica van de Open Universiteit zijn ook medewerker bij de Open Universiteit.*
>
> *Dus sommige docenten bij de vakgroep Informatica van de Open Universiteit houden niet van computers.*

Veel mensen blijken moeite te hebben met het bepalen van de juistheid van dergelijke redeneringen. Logica biedt methodes om redeneringen te formaliseren en vervolgens met formele methodes te onderzoeken of de redenering al dan niet klopt.

Logica is onderdeel van het curriculum van onder andere informatica, filosofie en wiskunde. Studenten ontwikkelen hun vaardigheden in logica door veel te oefenen. Een deel van de oefenopgaven zijn opgaven die de student stapsgewijs op moet lossen, en waarbij er bovendien verschillende manieren zijn om de opgave op te lossen. Een student die dit soort opgaven lastig vindt, zal hierbij hulp kunnen gebruiken. Hulp kan bestaan uit een aanwijzing hoe verder te gaan, als de student vastloopt, maar ook uit het constateren van fouten en eventuele misconcepties achter de fouten. Docenten kunnen deze hulp bieden, maar er zal niet altijd een docent beschikbaar zijn. In dergelijke gevallen kunnen Intelligente Tutor Systemen (ITS) deze rol vervullen. Een ITS is een systeem dat opgaven aanbiedt en de student feedback geeft. Als een opgave verschillende mogelijke oplossingen heeft (misschien zelfs oneindig veel), en studenten allerlei fouten kunnen maken, is het ondoenlijk om voor elke opgave afzonderlijk de aanwijzingen en feedback uit te schrijven. In het IDEAS[1] onderzoeksproject wordt gewerkt aan methoden om op een efficiënte manier kwalitatief goede ondersteuning te bieden bij dit type opgaven, voor allerlei soorten vakgebieden. In dit proefschrift richten we ons op tutor systemen voor logica. Daarbij willen we de volgende vragen beantwoorden:

> *Hoe kunnen we de benodigde kennis om logica opgaven op te lossen representeren?*
>
> *Hoe kunnen we aanwijzingen en feedback automatisch genereren?*

[1]IDEAS staat voor 'Interactive Domain-specific Exercise Assistants', een samenwerkingsproject tussen de Universiteit Utrecht en de Open Universiteit

Wat is het effect van het gebruik van de ontwikkelde ITS in logica onderwijs?

In ons onderzoek hebben we ons bij het beantwoorden van deze vragen beperkt tot drie deelgebieden van de logica: het herschrijven van formules, axiomatische bewijzen en bewijzen met inductie.

Het eerste deelgebied, het herschrijven van formules, maakt gebruik van standaardequivalenties: uitspraken die op grond van hun structuur altijd equivalent zijn. Een voorbeeld hiervan zijn de uitspraken “het is niet zo dat ik geen kaas eet” en “ik eet wel kaas”. Deze twee uitspraken zijn equivalent omdat de negatie van de negatie van een uitspraak equivalent is met diezelfde uitspraak zonder negaties. Herschrijven van formules kan nuttig zijn om deze te vereenvoudigen, om ze in een gewenste standaardvorm te brengen, of om te laten zien dat twee formules equivalent zijn. In onze ITS LogEx kunnen studenten oefenen met deze opgaven. Lesboeken geven vaak een ‘recept’ waarmee je een formule om kunt zetten naar een standaardvorm. Het komt echter regelmatig voor dat er een snellere en mooiere manier is om tot zo’n standaardvorm te komen. Omdat we willen dat studenten niet blindelings een recept volgen maar dat ze blijven nadenken over wat ze aan het doen zijn, hebben we voor een aantal gevallen heuristieken ontwikkeld waarmee zulke snellere oplossingen te vinden zijn. Als een student op papier moet laten zien dat twee formules equivalent zijn, dan zal ze vaak eerst wat aan de eerste formule rekenen, dan overgaan naar de tweede, en vervolgens misschien weer doorgaan met de eerste. Deze mogelijkheid om te switchen van oplossingsrichting hebben we in LogEx ingebouwd.

We hebben LogEx uitgeprobeerd, onder andere in een experiment met informatica studenten van een hogeschool. In dit experiment gebruikten we twee versies van LogEx: één waarin studenten op elk moment om een aanwijzing konden vragen en na iedere stap feedback op hun uitwerking kregen en één waarin studenten geen hint, maar wel een volledige oplossing konden krijgen en waarin ze pas na het voltooien van de opgave feedback ontvingen. De leereffecten maten we door van te voren een pretest en na afloop een posttest af te nemen. Het verschil in leereffect tussen de twee groepen was niet significant, maar de eerste groep scoorde wel significant beter op tentamenopgaven over het zelfde onderwerp dan studenten die niet deel hadden genomen aan het experiment. Het experiment leverde ook een flinke hoeveelheid gegevens over de manier waarop studenten met LogEx werken, doordat alle interactie tussen de student en het systeem werd gelogd. Uit deze data blijkt bijvoorbeeld dat tijdens het werken met LogEx studenten steeds minder fouten gaan maken.

Het tweede deelgebied gaat over axiomatisch bewijzen. In LogAx leert een student hoe je kunt bewijzen dat een bepaalde uitspraak volgt uit een reeks gegevens. De student mag hierbij gebruik maken van een klasse uitspraken die altijd waar zijn, de axioma’s, en van twee afleidingsregels, Modus Ponens en de deductiestelling. De eerste afleidingsregel is een formalisatie van redeneringen zoals: ‘als het regent dan worden de straten nat’, ‘het regent’, conclusie: ‘de straten worden nat’. De tweede afleidingsregel is hiervan min of meer het omgekeerde: hoe bewijs je de implicatie: ‘als het regent dan worden de straten nat’? Dat kun je doen door aan te nemen dat het regent, wat je een extra gegeven oplevert, en te laten zien dat hieruit volgt dat de straten nat worden. Om studenten aanwijzingen te kunnen geven bij deze opgaves, moesten we om te beginnen een manier vinden om verschillende oplossingen te representeren. Vaak kan dat met behulp van boom-structuren. In dit geval zou dan hetgeen je wilt bewijzen in de wortel staan, de bladeren van de boom bevatten axioma’s en aannames, en knooppunten tussen takken verbinden tussenresultaten. Maar omdat in dit axiomatische systeem er twee verschillende afleidingsregels zijn, is deze representatie niet voldoende omdat er verschillende verbindingen tussen dezelfde tussenresultaten kunnen zijn. De structuur die we daarom nodig hebben, heet een DAM (directed acyclic multigraph). In ons onderzoek hebben we onderzocht hoe we een bestaand algoritme voor een ander bewijssysteem aan kunnen passen om zo’n DAM te maken en hoe vervolgens uit de DAM een lineair bewijs is te halen. Het aantal bewijzen in een bepaalde DAM is eindig, terwijl er in principe oneindig veel manieren kunnen zijn om een bewijs te leveren. We hebben daarom ons algoritme dynamisch gemaakt, zodat deze bij een afwijkend gedeeltelijk bewijs van een student toch een vervolgstap kan vinden.

Het laatste deelgebied gaat over inductie. Het idee achter inductieve bewijzen is uit te leggen aan de hand van het volgende voorbeeld. Hoe bewijs je dat alle hazen lange oren hebben? Het bewijs begint met één of meer basisgevallen, in dit geval het bewijs dat de Adam- en Eva-haas lange oren hadden. Vervolgens laat je zien dat het hebben van lange oren erfelijk is. Daarvoor neem je een willekeurig paar ouder-hazen, waarvan je aanneemt dat ze lange oren hebben, en tenslotte toon je aan dat uit deze aanname, de zogenaamde inductiehypothese, volgt dat de kinderen de eigenschap ook hebben. Een eerste stap bij het geven van dit soort bewijzen bestaat uit een analyse van wat er precies te bewijzen is in de basisgevallen en in de inductieve gevallen (waarin je bewijst dat kinderen de eigenschap erven). Om studenten hiermee op weg te helpen zijn we begonnen met een begeleide versie van ons systeem LogInd waarin we studenten eerst vragen wat ze in elk van de gevallen moeten bewijzen, of wat de inductiehypothese in dit specifieke geval is. Pas als het deelbewijs op deze manier correct is opgezet, vraagt het systeem de student om het deelbewijs te voltooien. Studenten die wat verder zijn kunnen oefenen in een versie zonder begeleiding, waarin ze vrijer zijn in de volgorde van het opzetten van het bewijs.

In de eerste twee deelgebieden gebruiken we ‘buggy’ regels, om veel gemaakte fouten te kunnen herkennen, en te voorzien van feedback. Om dit te illustreren geven we eerst een voorbeeld van een correcte herschrijving en laten dan zien wat studenten hierin vaak fout doen. De uitspraak “het is niet zo dat ik soep en pudding eet” is te herschrijven in de equivalente uitspraak “ik eet geen soep *of* ik eet geen pudding”. Een veel gemaakte fout is de herschrijving naar “ik eet geen soep *en* ik eet geen pudding”. Hier is de student vergeten om het voegwoord ‘en’ uit de eerste uitspraak te vervangen door het voegwoord ‘of’. De buggy regel herkent deze fout en genereert een boodschap om de student hier op te wijzen.

Bij een inductief bewijs kunnen er fouten op veel verschillende niveaus optreden: een student kan op regelniveau een fout maken, maar ook op het niveau van de verschillende deelbewijzen. Mogelijk ontbreekt een deelbewijs, of wordt er juist op basisniveau te veel bewezen. Om dergelijke fouten te herkennen maken we gebruik van constraints, voorwaarden waar het bewijs aan moet voldoen, die een boodschap opleveren zodra een student een constraint overtreedt.

In dit proefschrift hebben we laten zien dat het zeer goed mogelijk is om in complexe domeinen zoals het leveren van axiomatische of inductieve bewijzen automatisch aanwijzingen en feedback te genereren. De gegenereerde aanwijzingen en feedback helpen studenten bij het construeren van oplossingen op een manier zoals experts ze ook construeren, en lijken in kleinschalige experimenten studenten goed te helpen.

Dankwoord

Ooit, in een ver verleden ben ik begonnen aan een proefschrift dat ik, om redenen die er nu niet toe doen, nooit heb voltooid. Dat er nu, tegen het einde van mijn wetenschappelijke carrière toch een proefschrift ligt, is mede te danken aan velerlei bijdragen.

Om te beginnen bedank ik mijn promotoren, die mij door commentaar en discussies hielpen bij opzet, uitvoering en verwerking van het onderzoek. Bastiaan, jij ook vooral bedankt voor het wegwijs maken in Haskell, voor de vele uren die we samen achter de computer doorbrachten waarin we via pair programming aan de tutoren werkten, en voor alle hulp bij LaTeX problemen. Johan, hartelijk dank voor je kritische bijdragen aan onze artikelen, en je onverminderde aandacht voor structuur en zorgvuldig formuleren. Zonder Harrie Passier's enthousiaste uitnodiging om mee te denken over een logicatutor was ik nooit aan dit onderzoek begonnen. Eerst vanuit Utrecht, later op afstand, was Alex Gerdes altijd bereid om te helpen met computerproblemen.

Ook studenten waren onmisbaar voor de totstandkoming van dit proefschrift. Dank voor jullie bijdrage in verschillende fasen van je studie: deelnemend of ondersteunend aan het onderzoek (de persoonlijke bedankjes hiervoor staan bij de desbetreffende artikelen), als 'proefpersoon' in een van de experimenten of door het beschikbaar stellen van uitwerkingen van huiswerk voor onderzoek. Op deze plaats ook mijn dank aan Hieke Keuning die het mogelijk maakte om een tutorsysteem uit te proberen met de studenten van haar hogeschool.

Inspiratie voor dit onderzoek ontleende ik aan de conferenties gewijd aan Tools for Teaching Logic. Deze boden me een podium voor mijn werk en de gelegenheid om collega-onderzoekers te ontmoeten.

Dank ook aan mijn leidinggevenden bij de Open Universiteit Nederland, die me de gelegenheid gaven om dit proefschrift af te ronden in een tempo dat rekening hield met een parttime aanstelling waarbinnen beperkt tijd was voor het promotietraject.

Tenslotte, dank aan familie en vrienden. Omdat ik dit traject gestart ben met open verwachtingen over het hoe en wanneer van een eindresultaat, heb ik er alleen in kleine kring over verteld. Toch wil ik jullie allen bedanken voor het 'er zijn'. Jammer genoeg maakt mijn vader de promotie niet meer mee, hij was trots op de (bijna) afronding en had zich verheugd op het bijwonen van de promotieplechtigheid.

Curriculum vitae

Josje Lodder
10 september 1956 geboren in Den Haag

1968 – 1974	Gymnasium β, Rijksscholengemeenschap Erasmus, Almelo
1974 – 1978	Doctoraal wiskunde, Rijksuniversiteit Utrecht
1978 – 1981	Wetenschappelijk medewerker, Rijksuniversiteit Utrecht
1981 – 1987	Diverse aanstellingen als docent wiskunde middelbare school en lerarenopleiding
1983 – 1987	Conservatorium piano, DM, Conservatorium Maastricht
1988 – 1991	Conservatorium piano, UM, Conservatorium Maastricht
1987 – heden	Docent bij de vakgroep Informatica aan de Open Universiteit

Bibliography

Abel, A., Chang, B.-Y. E., and Pfenning, F. (2001). Human-readable machine-verifiable proofs for teaching constructive logic. In Egly, U., Fiedler, A., Horack, H., and Schmitt, S., editors, *Proceedings of the Workshop on Proof Transformations, Proof Presentations, and Complexity of Proofs, Siena 2001*, pages 37–50.

Aguilera, G., de Guzmán, I. P., Ojeda, M., and Valverde, A. (2000). Master theses for providing feedback to the logic classroom. In Manzano, M., editor, *Proceedings of the First International Congress on Tools for Teaching Logic*, pages 169–173.

Ahmed, U., Gulwani, S., and Karkare, A. (2013). Automatically generating problems and solutions for natural deduction. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1968–1975.

Aleven, V., McLaren, B., Roll, I., and Koedinger, K. (2004). Toward tutoring help seeking. In Lester, J. C., Vicari, R. M., and Paraguaçu, F., editors, *Intelligent Tutoring Systems*, pages 227–239, Berlin, Heidelberg. Springer Berlin Heidelberg.

Aleven, V., Mclaren, B. M., Sewall, J., and Koedinger, K. R. (2009). A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal on Artificial Intelligence in Education*, 19(2):105–154.

Alvin, C., Gulwani, S., Majumdar, R., and Mukhopadhyay, S. (2014). Synthesis of geometry proof problems. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 245–252. Association for the Advancement of Artificial Intelligence (AAAI).

Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. (1995). Cognitive tutors: lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207.

Arun-Kumar (2002). Introduction to logic for computer science. Retrieved from `http://www.cse.iitd.ernet.in/~sak/courses/ilcs/logic.pdf`.

Association for Computing Machinery (ACM) and IEEE Computer Society Joint Task Force on Computing Curricula (2013). Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science. Retrieved from `http://www.acm.org/education/CS2013-final-report.pdf`.

Aubin, R. (1979). Mechanizing structural induction part i: Formal system. *Theoretical Computer Science*, 9(3):329 – 345.

Avital, S. and Libeskind, S. (1978). Mathematical induction in the classroom: Didactical and mathematical issues. *Educational Studies in Mathematics*, pages 429–438.

Barnes, T. and Stamper, J. (2008). Toward automatic hint generation for logic proof tutoring using historical student data. In Woolf, B. P., Aïmeur, E., Nkambou, R., and Lajoie, S., editors, *Intelligent Tutoring Systems*, pages 373–382, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bartsch, R. A., Bittner, W. M. E., and Jr., J. E. M. (2008). A design to improve internal validity of assessments of teaching demonstrations. *Teaching of Psychology*, 35(4):357–359.

Beeson, M. J. (1998). Design principles of MathPert: Software to support education in algebra and calculus. In Kajler, N., editor, *Computer-Human Interaction in Symbolic Computation*, pages 89–115. Springer-Verlag.

Ben-Ari, M. (2012). *Mathematical Logic for Computer Science, 3rd edition.* Springer.

Benthem, J. v., Ditmarsch, H. v., Ketting, J., Lodder, J., and Meyer-Viol, W. (2003). *Logica voor informatica, derde editie.* Pearson Education.

Billingsley, W. and Robinson, P. (2007). Student proof exercises using mathstiles and isabelle/hol in an intelligent book. *Journal of Automated Reasoning*, 39(2):181–218.

Black, P. and Wiliam, D. (1998). Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice*, 5(1):7–74.

Bolotov, A., Bocharov, A., Gorchakov, A., and Shangin, V. (2005). Automated first order natural deduction. In *Proceedings IICAI'05: the 2nd Indian International Conference on Artificial Intelligence*, pages 1292–1311.

Bornat, R. (2017). Jape. Retrieved from `https://www.cs.ox.ac.uk/people/bernard.sufrin/personal/jape.org/MANUALS/natural_deduction_manual.pdf`.

Boud, D. and Molloy, E. (2013). *Feedback in higher and professional education: Understanding it and doing it well.* Routledge, United Kingdom.

Boyer, R. and Moore, J. (1998). *A Computational Logic Handbook.* Academic Press International series in Formal Methods. Academic Press.

Broda, K., Ma, J., Sinnadurai, G., and Summers, A. (2006). Friendly e-tutor for natural deduction. In *Proceedings TFM'06: the Conference on Teaching Formal Methods: Practice and Experience.*

Brown, J. S. and VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4(4):379–426.

Bundy, A. (2001). The automation of proof by mathematical induction. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 845–911.

Bundy, A. (2005). *Rippling: Meta-Level Guidance for Mathematical Reasoning.* Cambridge Tracts in Theoretica. Cambridge University Press.

Burris, S. (1998). *Logic for mathematics and computer science.* Prentice Hall.

Castro, L. and Toro, M. A. (2004). The evolution of culture: from primate social learning to human culture. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 101(27), page 10235–10240.

Chi, M. (2009). Active-constructive-interactive: A conceptual framework for differentiating learning activities. *Topics in Cognitive Science*, 1(1):73–105.

Cody, C., Mostafavi, B., and Barnes, T. (2018). Investigation of the influence of hint type on problem solving behavior in a logic proof tutor. In *Artificial Intelligence in Education 19th International Conference, AIED 2018*, pages 58–62. Springer.

Corbett, A. T., Koedinger, K. R., and Anderson, J. R. (1997). Intelligent tutoring systems. In Helander, M., Landauer, T. K., and Prahu, P., editors, *Handbook of Human-Computer Interaction, Second Edition*, pages 849–874. Elsevier Science.

Crooks, T. J. (1988). The impact of classroom evaluation practices on students. *Review of Educational Research*, 58(4):438–481.

Dalen, D. (2004). *Logic and Structure.* Universitext (1979). Springer.

Dostálová, L. and Lang, J. (2007). Organon — the web tutor for basic logic courses. *Logic Journal of IGPL.*

Dostálová, L. and Lang, J. (2011). Organon: Learning management system for basic logic courses. In Blackburn, P., Ditmarsch, H., Manzano, M., and Soler-Toscano, F., editors, *Tools for Teaching Logic*, volume 6680 of *Lecture Notes in Computer Science*, pages 46–53. Springer Berlin Heidelberg.

Dubinsky, E. (1986). Teaching mathematical induction 1. *The journal of mathematical behavior*, pages 305–317.

Eagle, M., Johnson, M. W., and Barnes, T. (2012). Interaction networks: Generating high level hints based on network community clusterings. In *EDM*, pages 164–167.

Enderton, H. (2001). *A Mathematical Introduction to Logic.* Elsevier Science.

Ensley, D. E. and Winston Crawley, J. (2006). *Discrete Mathematics with Student Solutions Manual Set.* Wiley.

Ernest, P. (1984). Mathematical induction: A pedagogical discussion. *Educational Studies in Mathematics*, 15(2):173–189.

Evans, C. (2013). Making sense of assessment feedback in higher education. *Review of Educational Research*, 83(1):70–120.

Galafassi, F. F. P. (2012). *Agente pedagógico para mediação do processo de ensino-aprendizagem da dedução natural na lógica;.* PhD thesis, Universidade do Vale do Rio dos Sinos.

Galafassi, F. F. P., Santos, A. V., Peres, R. K., Vicari, R. M., and Gluz, J. C. (2015). Multi-plataform interface to an its of proposicional logic teaching. In Bajo, J., Hallenborg, K., Pawlewski, P., Botti, V., Sánchez-Pi, N., Duque Méndez, D. N., Lopes, F., and Julian, V., editors, *Highlights of Practical Applications of Agents, Multi-Agent Systems, and Sustainability - The PAAMS Collection: International Workshops of PAAMS 2015, Salamanca, Spain, June 3-4, 2015. Proceedings*, pages 309–319. Springer International Publishing.

Goguadze, G. (2010). *ActiveMath - generation and reuse of interactive exercises using domain reasoners and automated tutorial strategies.* PhD thesis.

Goguadze, G. (May 2011). *ActiveMath - Generation and Reuse of Interactive Exercises using Domain Reasoners and Automated Tutorial Strategies.* PhD thesis, Universität des Saarlandes, Germany.

Goldin, I. and Carlson, R. (2013). Learner differences and hint content. In Lane, H. C., Yacef, K., Mostow, J., and Pavlik, P., editors, *Artificial Intelligence in Education*, volume 7926, pages 522–531. Springer Berlin Heidelberg.

Goldin, I., Koedinger, K., and Aleven, V. (2012). Learner differences in hint processing. In *Proceedings of the 5th International Conference on Educational Data Mining.*

Goldrei, D. (2005). *Propositional and Predicate Calculus, A Model of Argument.* Springer.

Goldson, D., Reeves, S., and Bornat, R. (1993). A Review of Several Programs for the Teaching of Logic. *The Computer Journal*, 36:373–386.

Gottschall, C. (2012). The gateway to logic. Retrieved from `https://logik.phl.univie.ac.at/~chris/gateway/formular-uk.html`.

Grivokostopoulou, F., Perikos, I., and Hatzilygeroudis, I. (2013). An intelligent tutoring system for teaching fol equivalence. In *AIED Workshops.*

Gruttmann, S., Böhm, D., and Kuchen, H. (2008). E-assessment of mathematical proofs — chances and challenges for students and tutors. In *Proceedings of the 2008 International Conference on Information Technology in Education*, Wuhan,China.

Hake, R. (1998). Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *Am. J. Phys.*, 66(1):64–74.

Harel, G. (2001). The development of mathematical induction as a proof scheme: A model for dnr-based instruction. In S. Campbell, R. Z., editor, *Learning and teaching number theory*, pages 185–212. Kluwer Academic.

Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, New York, NY, USA, 1st edition.

Hattie, J. (2012). *Visible Learning for Teachers: Maximizing Impact on Learning.* Education (Routledge). Routledge.

Hattie, J. and Timperley, H. (2007). The power of feedback. *Review of Educational Research*, 77(1):81–112.

Hayes, W. L. (1988). *Statistics, 4th edition.* Holt, Rinehart and Winston, Inc.

Heeren, B. and Jeuring, J. (2014). Feedback services for stepwise exercises. *Science of Computer Programming, Special Issue on Software Development Concerns in the e-Learning Domain*, 88:110–129.

Heeren, B. and Jeuring, J. (2020). Automated feedback for mathematical learning environments. In *ICTMT*, pages 17–25.

Heeren, B., Jeuring, J., and Gerdes, A. (2010). Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370.

Herding, D. (2013). *The tutor-in-the-loop model for formative assessment.* PhD thesis, RWTH Aachen University.

Herding, D., Zimmermann, M., Bescherer, C., Schroeder, U., and Ludwigsburg, P. (2010). Entwicklung eines frameworks für semi-automatisches feedback zur unterstützung bei lernprozessen. In *DeLFI*, pages 145–156.

Hirst, H. P. and Hirst, J. L. (2015). *A Primer for Logic and Proof (2015 edition).* Retrieved from `http://www.appstate.edu/~hirstjl/primer/hirst.pdf`.

Huertas, A. (2011). Ten years of computer-based tutors for teaching logic 2000-2010: Lessons learned. In *Proceedings of the Third International Congress Conference on Tools for Teaching Logic*, TICTTL'11, pages 131–140, Berlin, Heidelberg. Springer-Verlag.

Hurley, P. (2008). *A Concise Introduction to Logic.* Cengage Learning.

Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press.

Jaehnig, W. and Miller, M. (2007). Feedback types in programmed instruction: A systematic review. *The Psychological Record*, 57.

Jaques, P., Seffrin, H., Rubi, G., de Morais, F., Ghilardi, C., Bittencourt, I., and Isotani, S. (2013). Rule-based expert systems to support step-by-step guidance in algebraic problem solving: The case of the tutor pat2math. *Expert Systems with Applications*, 40:5456–5465.

Kalmár, L. (1935). Über die axiomatisierbarkeit des aussagenkalküls. *Acta scientiarum mathematicarum*, 7:222–243.

Kelly, J. (1997). *The essence of logic.* The essence of computing series. Prentice Hall.

Kim, R., Weitz, R., Heffernan, N., Krach, N., and Edu, N. (2009). Tutored problem solving vs. "pure" worked examples. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pages 3121–3126.

Kodaganallur, V., Weitz, R. R., and Rosenthal, D. (2005). A comparison of model-tracing and constraint-based intelligent tutoring paradigms. *Int. J. Artif. Intell. Ed.*, 15(2):117–144.

Koedinger, K. R. and Aleven, V. (2007). Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Rev.*, 19(3):239–264.

Kreisel, G. (1965). Mathematical logic. In Saaty, T., editor, *Lectures on modern mathematics*, volume 3, pages 95–195. John Wiley & Sons, Inc., New York, London, and Sydney.

Leary, C. and Kristiansen, L. (2015). *A Friendly Introduction to Mathematical Logic.* SUNY Geneseo.

Liu, Z., Mostafavi, B., and Barnes, T. (2016). Combining worked examples and problem solving in a data-driven logic tutor. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems - Volume 9684*, ITS 2016, pages 347–353, New York, NY, USA. Springer-Verlag New York, Inc.

Lodder, J. and Heeren, B. (2011). A teaching tool for proving equivalences between logical formulae. In Blackburn, P., Ditmarsch, H., Manzano, M., and Soler-Toscano, F., editors, *Tools for Teaching Logic*, volume 6680 of *Lecture Notes in Computer Science*, pages 154–161. Springer-Verlag.

Lodder, J., Heeren, B., and Jeuring, J. (2015a). A domain reasoner for propositional logic. Technical Report UU-CS-2015-021, Department of Information and Computing Sciences, Utrecht University.

Lodder, J., Heeren, B., and Jeuring, J. (2015b). A pilot study of the use of logex, lessons learned. *CoRR*, abs/1507.03671. Proceedings of the Fourth International Conference on Tools for Teaching Logic (TTL2015).

Lodder, J., Heeren, B., and Jeuring, J. (2016). A domain reasoner for propositional logic. *Journal of Universal Computer Science*, 22(8):1097–1122.

Lodder, J., Heeren, B., and Jeuring, J. (2017). Generating Hints and Feedback for Hilbert-style Axiomatic Proofs. In Caspersen, M. E., Edwards, S. H., Barnes, T., and Garcia, D. D., editors, *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, March 8-11, 2017*, pages 387–392. ACM.

Lodder, J., Heeren, B., and Jeuring, J. (2019). A comparison of elaborated and restricted feedback in LogEx, a tool for teaching rewriting logical formulae. *Journal of Computer Assisted Learning*, 35(5):620–632.

Lodder, J., Heeren, B., and Jeuring, J. (2020). Providing Hints, Next Steps and Feedback in a Tutoring System for Structural Induction. *Electronic Proceedings in Theoretical Computer Science*, 313:17–34.

Lodder, J., Jeuring, J., and Passier, H. (2006). An interactive tool for manipulating logical formulae. In Manzano, M., Pérez Lancho, B., and Gil, A., editors, *Proceedings of the Second International Congress on Tools for Teaching Logic.*

Lodder, J., Passier, H., and Stuurman, S. (2008). Using ideas in teaching logic, lessons learned. *Computer Science and Software Engineering, International Conference on*, 5:553–556.

Lodder et al., J. (2018). *Logica en informatica; Lecture notes (in Dutch).* Open Universiteit Nederland.

Lukins, S., Levicki, A., and Burg, J. (2002). A tutorial program for propositional logic with human/computer interactive learning. In *SIGCSE 2002*, pages 381–385.

Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282.

Mathan, S. A. and Koedinger, K. R. (2005). Fostering the intelligent novice: Learning from errors with metacognitive tutoring. *Educational Psychologist*, 40(4):257–265.

Matsuda, N. and VanLehn, K. (2005). Advanced geometry tutor: An intelligent tutor that teaches proof-writing with construction. In *Proceedings of the 2005 Conference on Artificial Intelligence in Education: Supporting Learning Through Intelligent and Socially Informed Technology*, pages 443–450, Amsterdam, The Netherlands, The Netherlands. IOS Press.

McKendree, J. (1990). Effective feedback content for tutoring complex skills. *Human-computer Interaction*, 5:381–413.

Megill, N. D. (2007). *Metamath: A Computer Language for Pure Mathematics.* Lulu Press, Morrisville, North Carolina.

Mendelson, E. (2015). *Introduction to Mathematical Logic.* Discrete Mathematics and Its Applications. CRC Press, sixth edition.

Merrill, D. C., Reiser, B. J., Ranney, M., and Trafton, J. G. (1992). Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems. *Journal of the Learning Sciences*, 2(3):277–305.

Minica, S. (2015). RAESON: A tool for reasoning tasks driven by interactive visualization of logical structure. *CoRR*, abs/1507.03677.

Mitrovic, A. (2012). Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, 22(1):39–72.

Mitrovic, A., Martin, B., and Suraweera, P. (2007). Intelligent tutors for all: The constraint-based approach. *Intelligent Systems, IEEE*, 22:38–45.

Miwa, K., Terai, H., Kanzaki, N., and Nakaike, R. (2014). An intelligent tutoring system with variable levels of instructional support for instructing natural deduction. *Transactions of the Japanese Society for Artificial Intelligence*, 29(1):148–156.

Morgan, T., Uomini, N., and Rendell, L. e. a. (2015). Experimental evidence for the co-evolution of hominin tool-making teaching and language. *Nature Communications*, 6.

Morrison, D. M. and Miller, K. B. (2017). Teaching and learning in the pleistocene: A biocultural account of human pedagogy and its implications for aied. *International Journal of Artificial Intelligence in Education*, 28:439–469.

Mostafavi, B. and Barnes, T. (2016). Evolution of an intelligent deductive logic tutor using data-driven elements. *International Journal of Artificial Intelligence in Education*, pages 1–32.

Mostafavi, B. and Barnes, T. (2017). Evolution of an intelligent deductive logic tutor using data-driven elements. *International Journal of Artificial Intelligence in Education*, 27(1):5–36.

Müller, W. and Hiob-Viertler, M. (2010). Intelligent assessment in math education for complete induction problems. In Zhang, X., Zhong, S., Pan, Z., Wong, K., and Yun, R., editors, *Entertainment for Education. Digital Techniques and Systems: 5th International Conference on E-learning and Games, Edutainment 2010, Changchun, China, August 16-18, 2010. Proceedings*, pages 317–325, Berlin, Heidelberg. Springer Berlin Heidelberg.

Nachar, N. (2008). The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*, 4:13–20.

Nakevska, M., van der Sanden, A., Funk, M., Hu, J., and Rauterberg, M. (2014). *Interactive Storytelling in a Mixed Reality Environment: The Effects of Interactivity on User Experiences*, pages 52–59.

Narciss, S. (2008). Feedback strategies for interactive learning tasks. In Spector, J., Merrill, M., van Merriënboer, J., and Driscoll, M., editors, *Handbook of Research on Educational Communications and Technology.* Mahaw, NJ: Lawrence Erlbaum Associates.

Narciss, S. (2013). Designing and evaluating tutoring feedback strategies for digital learning environments on the basis of the interactive tutoring feedback model. *Digital Education Review*, 23:7–26.

Narciss, S., Sosnovsky, S. A., Schnaubert, L., Andres, E., Eichelmann, A., Goguadze, G., and Melis, E. (2014). Exploring feedback and student characteristics relevant for personalizing feedback strategies. *Computers & Education*, 71:56–76.

Natriello, G. (1987). Evaluation processes in schools and classrooms. Technical Report 12, Johns Hopkins Univ., Baltimore, MD. Center for Social Organization of Schools.

Nievergelt, Y. (2002). *Foundations of Logic and Mathematics: Applications to Computer Science and Cryptography.* Birkhäuser Boston.

Nwana, H. S. (1990). Intelligent tutoring systems: an overview. *Artificial Intelligence Review*, 4(4):251–277.

Ohlsson, S. (1994). Constraint-based student modeling. In Greer, J. E. and McCalla, G. I., editors, *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, pages 167–189, Berlin, Heidelberg. Springer Berlin Heidelberg.

Øhrstrøm, P., Sandborg-Petersen, U., Thorvaldsen, S., and Ploug, T. (2013). Classical syllogisms in logic teaching. In *Lecture Notes in Computer Science*, volume 7735, pages 31–43.

O'Rourke, E., Butler, E., Tolentino, A. D., and Popović, Z. (2019). Automatic generation of problems and explanations for an intelligent algebra tutor. In *AIED*.

Osera, P.-M. and Zdancewic, S. (2013). Teaching induction with functional programming and a proof assistant. In *SPLASH Educators Symposium (SPLASH-E), 2013*.

Palla, M., Potari, D., and Spyrou, P. (2012). Secondary school students' understanding of mathematical induction: structural characteristics and the process of proof construction. *International Journal of Science and Mathematics Education*, 10(5):1023–1045.

Pavlekoviĉ, M. (1998). An approach to mathematical induction - starting from the early stages of teaching mathematics. *Mathematical communcations*, pages 135–142.

Perkins, D. (2007). Strategic proof tutoring in logic. Master's thesis, Carnegie Mellon. Retrieved from `http://archive.org/details/thesis_201502`.

Perrenet, J. and Groen, W. (1993). A hint is not always a help. *Educational Studies in Mathematics*, 25(4):307–329.

Polycarpou, I. (2008). *An Innovative Approach to Teaching Structural Induction for Computer Science*. PhD thesis, Florida International University.

Prank, R. (2014). A tool for evaluating solution economy of algebraic transformations. *Journal of Symbolic Computation*, 61:100–115.

R. Anderson, J., T. Corbett, A., Koedinger, K., and Pelletier, R. (1995). Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4:167–207.

Race, P. (2005). *Making learning happen – A guide for post-compulsory education*. Sage.

Razzaq, L. and Heffernan, N. (2009). To tutor or not to tutor: That is the question. In *Proceedings of the 14th International Conference on Artificial Intelligence in Education, AIED 2009*, pages 457 – 464.

Razzaq, L., Heffernan, N. T., and Lindeman, R. W. (2007). What level of tutor interaction is best? In *Proceedings of the 2007 Conference on Artificial Intelligence in Education: Building Technology Rich Learning Contexts That Work*, pages 222–229, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Rebholz, S. and Zimmermann, M. (2013). Applying computer-aided intelligent assessment in the context of mathematical induction. In Pan, Z., Cheok, A. D., Müller, W., Iurgel, I., Petta, P., and Urban, B., editors, *Transactions on Edutainment X*, volume 7775 of *Lecture Notes in Computer Science*, pages 191–201. Springer Berlin Heidelberg.

Robson, D., Abell, W., and Boustead, T. (2012). Encouraging students to think strategically when learning to solve linear equations. *International Journal for Mathematics Teaching and Learning.* Retrieved from `http://www.cimt.org.uk/journal/robson.pdf`.

Rodríguez-Gómez, G. and Ibarra-Sáiz, M. (2015). *Assessment as Learning and Empowerment: Towards Sustainable Learning in Higher Education*, pages 1–20.

Sadigh, D., Seshia, S. A., and Gupta, M. (2012). Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems. In *Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*, WESE '12, New York, NY, USA. Association for Computing Machinery.

Salden, R. J., Koedinger, K., Renkl, A., Aleven, V., and McLaren, B. (2010). Accounting for beneficial effects of worked examples in tutored problem solving. *Educational Psychology Review*, 22(4):379–392.

Schoenfeld, A. (1987). Cognitive science and mathematics education: An overview. In Schoenfeld, A., editor, *Cognitive Science and Mathematics Education*, chapter 1, pages 1–32. Lawrence Erlbaum Associates.

Shrestha, P., Maharjan, A., Wei, X., Razzaq, L., Heffernan, N. T., and Heffernan, C. (2009). Are worked examples an effective feedback mechanism during problem solving? In *Proceedings of the Annual Meeting of the Cognitive Science Society*, pages 1876–1881.

Shute, V. J. (2008). Focus on formative feedback. *Review of Educational Research*, 78(1):153–189.

Shute, V. J. and Regian, J. W. (1993). Principles for evaluating intelligent tutoring systems. *Journal of Artificial Intelligence in Education*, 4(2-3):245–271.

Sieg, W. (2007). The AProS project: Strategic thinking & computational logic. *Logic Journal of the IGPL*, 15(4):359–368.

Stamper, J. C., Barnes, T., and Croy, M. (2011a). Enhancing the automatic generation of hints with expert seeding. *Int. J. Artif. Intell. Ed.*, 21(1-2):153–167.

Stamper, J. C., Eagle, M., Barnes, T., and Croy, M. (2011b). Experimental evaluation of automatic hint generation for a logic tutor. In *Proceedings of the 15th International Conference on Artificial Intelligence in Education*, AIED'11, pages 345–352, Berlin, Heidelberg. Springer-Verlag.

Suppes, P. (1971). Computer-assisted instruction at Stanford. Technical Report 174, Stanford, institute for mathematical studies in the sociai sciences.

Sweller, J., Ayres, P., and Kalyuga, S. (2011). *The Worked Example and Problem Completion Effects*, pages 99–109. Springer New York, New York, NY.

Sweller, J. and Cooper, G. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2:59–89.

Van Ditmarsch, H. (1998). User interfaces in natural deduction programs. In *Informal proceedings of the Workshop on User Interfaces for Theorem Provers Eindhoven University of Technology*, pages 87–95.

van Gog, T., Kester, L., and Paas, F. (2011). Effects of worked examples, example-problem, and problem-example pairs on novices' learning. *Contemporary Educational Psychology*, 36(3):212 – 218.

VanLehn, K. (2006). The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265.

VanLehn, K. (2011). The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221.

Varga, K. P. and Várterész, M. (2006). Computer science, logic, informatics education. *Journal of Universal Computer Science*, 12(9):1405–1410.

Vrie, E. M. v. d. and Lodder et al., J. S. (2009). *Discrete wiskunde A, Lecture notes (in Dutch).* Open Universiteit Nederland.

Wasilewska, A. (2018). *Logics for computer science.* Springer.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Yacef, K. (2005). The logic-ita in the classroom: A medium scale experiment. *Int. J. Artif. Intell. Ed.*, 15(1):41–62.

Zimmermann, M. and Herding, D. (2010). Entwicklung einer computergestützten lernumgebung für bidirektionale umformungen in der mengenalgebra. *Beiträge zum Mathematikunterricht 2010.*