



Linda Mannila (Grandell)

Teaching Mathematics and Programming

New Approaches with Empirical Evaluation

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Dissertations
No 124, November 2009

Teaching Mathematics and Programming - New Approaches with Empirical Evaluation

Linda Mannila (Grandell)

*To be presented, with the permission of the Faculty of Technology of Åbo Akademi
University, for public criticism in Auditorium Gamma, the ICT building,
on November 27, 2009, at 12 noon.*

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland

2009

Supervisors

Prof. Ralph-Johan Back
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5, 20520 Turku
Finland

Reviewers

Prof. Lauri Malmi
Department of Computer Science and Engineering
Helsinki University of Technology
Konemiehentie 2, 02015 TKK
Finland

Doc. Juha Oikkonen
Department of Mathematics
University of Helsinki
Yliopistonkatu 5, 00014 University of Helsinki
Finland

Opponents

Prof. Lauri Malmi
Department of Computer Science and Engineering
Helsinki University of Technology
Konemiehentie 2, 02015 TKK
Finland

Doc. Juha Oikkonen
Department of Mathematics
University of Helsinki
Yliopistonkatu 5, 00014 University of Helsinki
Finland

ISBN 978-952-12-2364-8
ISSN 1239-1883

Abstract

Programming and mathematics are core areas of computer science (CS) and consequently also important parts of CS education. Introductory instruction in these two topics is, however, not without problems. Studies show that CS students find programming difficult to learn and that teaching mathematical topics to CS novices is challenging. One reason for the latter is the disconnection between mathematics and programming found in many CS curricula, which results in students not seeing the relevance of the subject for their studies. In addition, reports indicate that students' mathematical capability and maturity levels are dropping.

The challenges faced when teaching mathematics and programming at CS departments can also be traced back to gaps in students' prior education. In Finland the high school curriculum does not include CS as a subject; instead, focus is on learning to use the computer and its applications as tools. Similarly, many of the mathematics courses emphasize application of formulas, while logic, formalisms and proofs, which are important in CS, are avoided. Consequently, high school graduates are not well prepared for studies in CS.

Motivated by these challenges, the goal of the present work is to describe new approaches to teaching mathematics and programming aimed at addressing these issues:

- *Structured derivations* is a logic-based approach to teaching mathematics, where formalisms and justifications are made explicit. The aim is to help students become better at communicating their reasoning using mathematical language and logical notation at the same time as they become more confident with formalisms.
- The *Python* programming language was originally designed with education in mind, and has a simple syntax compared to many other popular languages. The aim of using it in instruction is to address algorithms and their implementation in a way that allows focus to be put on learning algorithmic thinking and programming instead of on learning a complex syntax.
- *Invariant based programming* is a diagrammatic approach to developing programs that are correct by construction. The approach is based on elementary propositional and predicate logic, and makes explicit the underlying mathematical foundations of programming. The aim is also to show how mathematics in general, and logic in particular, can be used to create better programs.

In addition to describing how these approaches can be used in education, we have conducted empirical studies in authentic classroom settings at high school and university level to examine the suitability of the approaches in a teaching context. The findings indicate several benefits of introducing the approaches in novice education, and also point out places for improvement related to the approaches *per se* as well as to the way in which they are used in instruction.

Keywords: introductory programming education, mathematics education, high school, secondary school education, novices, structured derivations, invariant based programming, Python

Acknowledgements

The road to finishing this thesis has had its ups and downs, and now, when the end is in sight, I am happy to have the opportunity to express my deepest gratitude to those who have guided and encouraged me on the way.

First and foremost, I want to thank my supervisor, Professor Ralph-Johan Back, who has inspired my work since the very beginning. Ralph has given me valuable advice and helped me stay focused on the goal ahead, especially at times when the final destination was not easy for me to see. Knowing that you believed in me has made all the difference, and I am grateful to you for helping me come to the point where I get to write these lines. I also want to thank Professor Tapio Salakoski for keeping an eye on the progress of my work, and for being available when I needed his advice.

I greatly appreciate that Professor Lauri Malmi at Helsinki University of Technology and Docent Juha Oikkonen at Helsinki University agreed to review this thesis and to function as opponents at the public defence. Thank you for your valuable comments.

Being based on research conducted in many institutions over a time span of several years, the findings presented in this thesis are the result of team work rather than the accomplishment of one single person. Data collection and analysis have benefited from the dedicated work by a number of colleagues, including Mia Peltomäki, Solveig Wallin, Johannes Eriksson and Patrick Sibelius. It has been, and still is, a true pleasure to work with all of you. A special thanks to Mia – you have been a valuable discussion partner, colleague and, most of all, a very dear friend.

I would like to thank the Department of Information Technologies at Åbo Akademi University for giving me the opportunity to pursue the doctoral degree by offering me a position as a PhD student. Each research leader at the department deserves a special thanks for contributing to the scientific and high quality research environment that I have had the pleasure to work in during my studies. In particular, I am grateful to Professor Barbro Back and Professor Kaisa Sere, who have been, and continue to be, strong role models of women in the IT field. I value your advice very much.

I am grateful to the Academy of Finland, who has supported the Centre for Reliable Software Technology (CREST) within which my research has been conducted. I also want to thank *Stiftelsens för Åbo Akademi forskningsinstitut* and Turku Centre for Computer Science (TUCS) for supporting my travel to conferences and workshops. I also want to express my gratitude to the Federation of Finnish Technology Industries, whose generous support made it possible for us to initiate the IMPED resource center.

I also want to thank all my other colleagues – past and present. The administrative personnel at the department – Britt-Marie Villstrand, Christel Engblom, Tiina Haanila, Pia-Maria Kallio, Minna Asplund, Görel Salomaa, and Stina Störling-Sarkkila – thank you for always being there to answer my questions and to give practical advice. Many thanks to Jockum Lillsund for helping me with just about anything related to the hardware and software I have been using. Many fellow researchers and colleagues both at home and abroad have had a positive impact on my work, including Petri Sallasmaa, Teemu Rajala, Viorel Preoteasa, Marina Waldén, Herman Norrgrann, Anders Berglund, Arnold Pears, Mats Daniels, Lars-Åke Larzon, Michael deRaadt, Angela Carbone, Raymond Lister, Sue Fitzgerald, Annamari Soini, Kerstin Fagerström, Tomi Mäntylä, Torbjörn Lundqvist, Mikko Laakso, Ragnar Wikman, Åke Gustavson, Mats Neovius, Kim Solin, Larissa Meinicke, Cristina Seceleanu, Päivi Kinnunen, and Essi Isohanni. Thank you all for fruitful discussions, joyful laughters and memorable moments.

There is a countless number of persons outside the academic walls who I am forever grateful to for their continuous encouragement. Thanks to my parents-in-law, Gitta and Reino Mannila, as well as to all my relatives and friends, who have supported me although it was not always clear to them what I was doing at the university. Thank you all for being there and for giving me something else to think about.

My parents, Rita and Stig Grandell – thank you for always believing in me, letting me choose my own paths in life and providing a place for me to charge my batteries. My brother, Ronnie Grandell – you have been a greater support than you can imagine. Our discussions and long walks have always helped me move forward with new insights and ideas. I owe you more than words can ever express.

Above all, I want to thank my husband Petri and our precious daughter Nadja. You are the light of my life, and make my world a wonderful place to wake up to every morning. I love you.

Turku, October 2009

Linda Mannila

Preface

To set the stage for the thesis, it seems appropriate to first consider why I would conduct research in this area, that is, mathematics and programming education. In summary, this work is the result of bringing together three underlying reasons.

First, I have always been fascinated by computers and computer technology, but I never really understood what computer science as a discipline was about. We had some computer courses at school, but most of the time we students were the ones educating our teacher instead of the other way around. In addition, the courses were completely focused on using the computer as a tool to write documents, create spreadsheets and play games. I think it is safe to say that I did not actually have a clue about what computer science was when I enrolled for studies in the field. I hoped that it would have something to do with computers, but little did I know about the importance of abstraction, formalisms and mathematics. The Swedish term for my degree program at the time, *informationsbehandling*, which literally translated would be something like “information processing” did not give much away either. Most people thought I was going to study to become a journalist or a librarian.

Second, ever since I was a little girl, I have enjoyed explaining things to others, both in everyday life and at school. It was therefore no wonder that I found myself teaching both before and after I received my Master’s degree. Once again I was faced with the same question on the nature of computer science, but now from a different perspective – how could I teach the fundamentals in as good a way as possible?

Third, I actually never cared much for mathematics in elementary school, but in junior high I had a great mathematics teacher who sparked my interest. The interest stayed alive throughout high school, and when I enrolled at the university, the negative attitudes towards mathematics among many of my peers came as a surprise. The further I progressed in my studies, the more I realized that I really did need the skills I had learned from mathematics. Not necessarily the techniques for how to, say, solve a differential equation, but rather the reasoning and thinking skills developed when “doing mathematics”. The question was how I could help my students see the same thing.

Before moving on, I also want to position myself as a researcher. I am not an educationalist. Rather, I am a computer scientist with a genuine interest in educational aspects. Being a computer scientist conducting education related research means that I am not able to see or reveal everything that an educationalist would. In order to improve

my knowledge of educational aspects, I have studied pedagogics, educational technology and research methods in education rather extensively both before and while working on this thesis. Nevertheless, I will still lack the deep and extensive understanding that only a person with solid experience in the field has.

However, to be able to evaluate and improve methods used in computer science education, one needs solid understanding of the underlying principles of the discipline. As a computer scientist, I also have personal experience in learning the programming and mathematics taught in the introductory curriculum from scratch. By sharing the same discipline context it becomes easier to know what issues to focus on. Similarly, when analyzing the data, the “insider knowledge” makes it easier to interpret the results and decide on improvements since technical and other discipline specific concepts are familiar. For the research presented in this thesis, I therefore see it as a strength that I am a computer scientist rather than an educationalist.

List of Original Publications

This thesis is based on the following nine publications, which are referred to in the text by the corresponding Roman numerals (I-IX).

- Paper I Back, R-J., Mannila, L., Peltomäki, M. & Sibelius, P. Structured Derivations: a Logic Based Approach to Teaching Mathematics. In *FORMED 2008: Formal Methods in Computer Science Education*. Budapest, Hungary, March 2008.
- Paper II Mannila, L. & Wallin, S. Promoting Students' Justification Skills Using Structured Derivations. In *ICMI 19: 19th ICMI Study Conference on Proof and Proving in Mathematics Education*. Taipei, Taiwan, May 2009.
- Paper III Back, R-J., Mannila, L. & Wallin, S. "It Takes Me Longer, But I Understand Better" – Student Feedback on Structured Derivations. *TUCS Technical Reports*, number 943. Turku Centre for Computer Science, April 2009. (Submitted for publication)
- Paper IV Back, R-J., Mannila, L. & Wallin, S. Student Justifications in High School Mathematics. In *CERME 6: Sixth Conference of European Research in Mathematics Education*. Lyon, France, January-February, 2009.
- Paper V Grandell, L., Peltomäki, M., Back, R-J. & Salakoski, T. Why Complicate Things? Introducing Programming in High School Using Python. In *Proceedings of the 8th Australasian Computing Education Conference (ACE2006)*, Hobart, Tasmania, pp. 71-80, 2006.
- Paper VI Mannila, L., Peltomäki, M. & Salakoski, T. What About a Simple Language? Analyzing the Difficulties in Learning to Program. *Computer Science Education*, 16(3), pp. 211-228. Routledge, 2006.
- Paper VII Mannila, L. Novices' Progress in Introductory Programming Courses. *Informatics in Education*, 6(1), pp. 139-152. Institute of Mathematics and Informatics, Lithuanian Academy of Sciences, Lithuania, 2007.

Paper VIII Back, R-J., Eriksson, J. & Mannila, L. Teaching the Construction of Correct Programs Using Invariant Based Programming. In *Proceedings of the 3rd South-East European Workshop on Formal Methods*. Thessaloniki, Greece, December 2007.

Paper IX Mannila, L. Invariant Based Programming in Education - An Analysis of Student Difficulties. *TUCS Technical Reports*, number 944. Turku Centre for Computer Science, April 2009. Accepted for publication in *Informatics in Education*.

The work presented in this thesis has not been conducted in isolation, but is the result of a joint effort of several contributors. In the European computer science community, authors are usually listed in alphabetical order. Hence, the contribution of a given author cannot be determined based on the author listing.

Paper I - VI and Paper VIII are the result of shared contributions from several researchers. The author of this thesis played a major role in all phases of the research process (design, data collection, analysis) in these studies and was also responsible for the final drafting of the articles. In Paper VII and Paper IX, the author was the sole author.

Contents

I	Research Summary	1
1	Introduction	3
1.1	Background	3
1.2	Focus of Research	5
1.3	Research Questions	6
1.4	Structure of the Thesis	7
2	Research Context	9
2.1	Motivation and Learning	9
2.2	Nature of Computer Science	11
2.3	Programming in Education	11
2.4	Mathematics in Education	19
3	Evaluated Approach I: Structured Derivations	27
3.1	Overview	27
3.2	Using Structured Derivations in Education	32
4	Evaluated Approach II: Python	37
4.1	Overview	37
4.2	Using Python in Education	40
5	Evaluated Approach III: Invariant Based Programming	43
5.1	Overview	43
5.2	Using Invariant Based Programming in Education	51
6	Research Framework	55
6.1	Education Research in CS	55
6.2	Development as a Research Activity	57
6.3	Research Design in This Thesis	58
7	Overview of Publications	63
7.1	Structured Derivations in Education	63
7.2	Python in Education	66

7.3	Invariant Based Programming in Education	68
8	Discussion	71
8.1	Structured derivations	71
8.2	Python	74
8.3	Invariant Based Programming	75
8.4	Applicability of Developmental Research	78
8.5	Implications for Teaching	79
8.6	Evaluating the Results	83
8.7	Relevance	88
9	Final Words	89
9.1	Conclusions	89
9.2	Future Research	90
II	Original Publications	115

Part I

Research Summary

Chapter 1

Introduction

In this chapter, we give an overview of the thesis and a brief background to motivate the work.

1.1 Background

Programming makes up an essential part of the standard practices of computer science (CS), and can be seen as one of the core skills that a computer scientist should achieve competence in [46]. Another such area is mathematics, as CS “depends on mathematics for many of its fundamental definitions, axioms, theorems, and proof techniques. In addition, mathematics provides a language for working with ideas relevant to CS, specific tools for analysis and verification, and a theoretical framework for understanding important computing ideas.” [104, p. 40] .

Due to their evident importance for the entire discipline, programming and mathematics constitute essential parts of the CS curriculum [104]. However, teaching programming and mathematics is not without problems. Programming has traditionally been viewed as a difficult topic, and research has shown that novices face several problems when learning to program [72, 120, 131, 174, 196]. In order to address the difficulties, different approaches have been suggested, for instance, based on curricula, pedagogical principles, tools and language choice [154]. Nevertheless, after years of research and attempts to improve the learning outcomes, programming still appears difficult to students.

Teaching mathematics is also challenging, but for somewhat other reasons. Whereas students enrolling at a CS department expect and anticipate to learn programming, the relevance of mathematics is seldom as clear; rather, students exhibit an attitude problem towards studying mathematics. This can be partly explained by the separation between theory and practice, which is commonplace in CS curricula. Instead of viewing mathematics as an integral part of programming, it is seen as a separate "add-on" [84]. Clearly, convincing students of the relevance of mathematics for their career as computer scientists under such circumstances is challenging.

The challenges at CS departments can also be traced back to lower levels of educa-

tion. For instance in Finland, most students who enroll for studies at university level have a background in high school education,¹ where CS is not taught as an individual subject [71]. Instead of giving students the opportunity to learn about concepts such as algorithms, data structures and design, the computer and its applications are used as tools in other subjects. The rationale for this is to guarantee that all students graduating from high school are capable of using the computer efficiently in various ways. Although this is a sound reason, it neglects those students who would be interested in learning more about the underlying science. In addition, when students' main exposure to CS is from a user's point of view, it is no wonder that those enrolling at CS departments have a hard time understanding the need for mathematics. After all, they have been able to "study CS" earlier, without the need to involve any mathematics.

There are also reports on a drop in mathematical capability and maturity levels among Finnish high school graduates [129]. The students are still offered extensive training in mathematics; depending on the choice of syllabus, a student can have up to ten compulsory and three elective mathematics courses. However, a review of the curriculum development indicates that courses have been lightened and requirements have been lowered [115, 128]. In addition, the focus of the mathematics taught has changed. Calculators and formula collections are used extensively, and "doing mathematics" has to a rather large extent become a matter of inputting numbers in the right spots. Sorvali [195] argues that this cannot even be called mathematics; for something to be viewed as mathematics it needs to involve creativity and logical reasoning. Logic, formalisms and proofs are, however, avoided in Finnish high school mathematics and are currently only covered in one elective course [71]. The lack of logic and proof is not only a problem in the Finnish context, but is reported on elsewhere as well (for instance, [93, 97, 185]).

As a consequence of the lack of mathematical maturity among beginning students, formal techniques are usually avoided in introductory CS education; these are viewed as difficult topics requiring prerequisite knowledge of advanced mathematics and logic, which novices simply do not have [200] and show little interest in learning. Consequently, theoretical courses have traditionally been deferred to later in the university studies. Thus, we get caught in a vicious circle where the courses that could improve students' appreciation for mathematics are postponed due to students lacking the skills necessary to complete the courses. When the material is introduced at more advanced levels, allowances must be made since the typical student's maturity level for the material is deficient and therefore must be developed.

As a result, some major problems are typically observed at CS departments: in addition to the widely recognized challenges with learning "practical programming", most students also lack confidence in the use of formalisms and logic, and do not see the relevance of theory in CS. These three issues make up the driving force behind this thesis.

¹In the Finnish educational system, high schools are referred to as upper secondary schools, providing general education to students aged 16-19.

1.2 Focus of Research

The present work builds on nine original publications listed on pages vii-viii above. The primary goal of the research is to contribute to the discussion on how programming and mathematics can be taught at high school and introductory university level, with a particular focus on developing the skills needed when studying at a CS department. This is accomplished by presenting and evaluating three teaching approaches:

- *Structured derivations* is a logic-based approach to teaching mathematics, offering a systematic presentation of mathematical solutions and proofs. A structured derivation has a fixed format, where formalisms and justifications are made explicit and logic becomes a tool for doing mathematics rather than an object of mathematical study. The aim is to help students become better at communicating their reasoning using mathematical language and logical notation at the same time as they become more confident with formalisms.
- *Python* is a dynamic interpreted programming language, which was originally designed with education in mind. Compared to many other popular languages, Python has a simple syntax, which may avoid the risk of novices having to pay too much attention to a verbose syntax at the expense of the core concepts of programming. The aim of using Python in instruction is thus to address algorithms and their implementation in a way that allows main focus to be put on algorithmic thinking and programming concepts rather than on sorting out and memorizing a complex syntax.
- *Invariant based programming* is a diagrammatic approach to developing programs that are correct by construction. The approach is based on elementary propositional and predicate calculus, and does hence not require prior knowledge in advanced mathematics or logic. The aim is to make explicit how, and why, a program works as well as to show how logic can be used to create better programs.

Essentially, each approach introduces a new notation for doing mathematics or constructing programs. Although the approaches may seem rather distinct at first, they share many similarities. Proofs in invariant based programming can be written as structured derivations and an invariant based program can be translated into executable code (for instance, in Python). Consequently, the invariant based approach serves as a concrete link between programming and mathematics. The methods are described in more detail in chapters 3,4 and 5, as well as in the corresponding publications (Part II).

The research approach is empirical and all studies are based on courses that have taken place in authentic settings, i.e. classrooms at high schools or at the university (with the author or a colleague as the teacher). Data have been collected through questionnaires, observations, exams and interviews, which have been analyzed using both qualitative and quantitative methods.

The research is built on the developmental research methodology, which is described in section 6.2. In line with this methodology, the results of the studies have been used

to inform the further development of the approaches, modify how these have been used in the teaching situation (for instance, by changing the order in which topics are introduced) and to suggest points of interest for further investigation. The goal is thus not to implement a complete curriculum or a “ready-made” product, but to contribute to the further development of 1) mathematics and programming education with a special focus on students who are or seek to become CS majors on one hand, and 2) the teaching approaches introduced on the other.

1.3 Research Questions

The detailed research questions addressed vary from study to study, but all of them aim at exploring one or more of the following main research questions:

- *How can the approaches be introduced in education?* This is not a question to be explicitly investigated, but is rather addressed in a descriptive manner.
- *How are the approaches experienced by the students?* This question is important for structured derivations and invariant based programming in particular, as these methods are quite different from the traditional approaches to teaching mathematics and programming at introductory level. A method that is not appreciated by students can still yield good results; this may, however, not be successful in the long run. Factors such as motivation, affect and values² are important for student success [8], since teaching that is considered uninteresting or unappealing tend to lead to students dropping out [33, 155]. This is an especially important aspect given the decrease in enrollment to CS departments; in such situations it becomes important to choose teaching approaches that appeal to the students [119]. To some extent, this question also touches upon the effect the newly introduced methods have on student performance compared to using the traditional approaches.
- *What are the potential difficulties and how can they be avoided/eased?* This question aims at investigating what difficulties students encounter when using the respective approaches, and is addressed by analyzing student created solutions and programs as well as student feedback. The aim is to provide insight into how the approaches could be modified in order to better suit a certain educational setting. In addition, we seek to address this question at a more general level: although the studies have been conducted using specific teaching approaches, the aim is to provide results that can be used to contribute to the understanding of difficulties faced when using other similar approaches to teaching mathematics and programming. For instance, the results from the studies on invariant based programming can be used to indicate where difficulties of proving program correctness lie, regardless of the method of instruction used.

²These factors and their relationship to learning are covered in section 2.1.

The study specific questions are listed in chapter 7 as the respective publications are briefly summarized. In addition to addressing these questions, a secondary aim is to evaluate the use of the developmental research approach for studying and developing new teaching methods.

For each of the three approaches, one paper has served as a background publication also discussing how the approach has been introduced in education (Paper I for structured derivations, Paper V for Python and Paper VIII for invariant based programming). Due to the highly exploratory nature of introducing a teaching approach for the first time in a given context, the main goal of these papers is not necessarily to address any specific research questions. Rather, the intention is to describe the respective approaches, give a model for how they could be introduced in an educational setting, and point out interesting issues for further investigation.

1.4 Structure of the Thesis

This thesis consists of two parts: a summary and nine empirical publications. The main purpose of the summary is to synthesize and discuss the aggregated results presented in the publications. After this introductory chapter, the contextual framework will be described in more detail in chapter 2. Chapters 3-5 introduce the evaluated approaches and chapter 6 contains an overview of the research design and the methodological choices made in the empirical studies. With these contextual and methodological considerations as a background, the nine publications and the corresponding results are summarized in chapter 7. The findings are discussed in chapter 8, which also contains comments on the limitations and relevance of the research presented. The final chapter (9) concludes the thesis and gives suggestions for future research. The summary is followed by the original publications in numerical order.

Chapter 2

Research Context

The goal of this chapter is to provide a contextual framework for the research. First, we give a brief background on motivation and learning, after which we present the rationale for our work from the perspective of traditional education in mathematics and programming at high school and university level. A discussion on the aspect of notation in these educational contexts is also included.

2.1 Motivation and Learning

As was briefly argued when presenting the second main research question in section 1.3, student attitudes towards teaching and the methods applied is a crucial factor to consider when designing education. Although motivational factors and approaches to learning are not explicitly covered in the studies of this thesis, it is important to have a basic understanding for how these are related and how they can affect attitudes towards learning new approaches.

Motivation has been called the “*sine qua non* for learning” [125, p. 378], in other words, there would not be any learning without motivation. A common definition of motivation is, however, hard to find. One popular way of defining it is based on its behavioral outcomes, for instance, as something that “creates the ‘movement’ of learning” [66, p. 8] and explains “what gets people going, keeps them going, and helps them finish tasks” [159, p. 104]. In this sense, motivation is seen as providing “a source of energy that is responsible for *why* learners decide to make an effort, *how long* they are willing to sustain an effort, *how hard* they are going to pursue it, and *how connected* they feel to the activity” [175]. Another common definition is to view motivation as a product, referring to “a willingness, desire, or condition of arousal or activation” [8, p. 369].

Since motivation affects student choices and levels of engagement, it will consequently influence learning, achievement and performance [8]. The motivational outcomes are, however, also affected by other factors apart from motivation, such as prior knowledge and ability level. Students may, for instance, perform well without being highly motivated. In addition, students with the same motivational goals may end up with different

outcomes, depending on what choices they make, where they focus their attention and how persistent they are in pursuing the goal [125].

Many factors are relevant when trying to understand and explain motivation. Three related constructs in this context are *goals*, *values* and *affect*, which are all associated with students' persistence, effort of engagement and performance level [8]. For instance, students who value a task or find it interesting are likely to show high levels of engagement towards and enjoyment in it. In the context of our work in general, and the second main research question in particular, the most relevant construct to consider is that of affect. Affect, i.e. the experience of emotions, has an impact on the goals students choose to pursue and can lead to valuing particular tasks, behaviors and activities. When designing education it is hence essential to focus on making the activities as enjoyable as possible, since positive initial experiences with an activity are transformed into a "more sustained value of interest for similar tasks. Over time and with repeated exposure to a domain, students develop more enduring values." [8, p. 383] What happens in the classroom also influences, for better or for worse, the motivational levels that students bring to class in the first place [49].

Types of motivation and their connection to learning are another relevant aspect to consider. Motivation can be either extrinsic or intrinsic [52, 103]. *Extrinsic motivation* refers to behavior where an activity is performed for external reasons or rewards; in an educational setting, such reinforcements can be marks, grades and different kinds of qualifications. *Intrinsic motivation*, on the other hand, is derived from within the person or the activity itself, and the activity is maintained because the person thinks it is meaningful and genuinely interesting; in a school setting such motivation is exhibited by students who learn out of interest and a wish to find something out.

These two types of motivation are closely related to approaches to learning. The main distinction is that between deep and surface learning, which was originally identified by Marton and Säljö in the 1970s [130] and since further elaborated by, for instance, Biggs [28], Entwistle [65] and Ramsden [168]. The intention of a *deep approach* to learning is "to understand ideas for yourself", whereas a *surface approach* aims at "[coping] with course requirements" [66, p. 19]. Deep learning is thus characterised by a student's attempt to search for meaning in a task, looking for patterns and underlying principles, critically analyzing new ideas and relating them to previous knowledge and experience. Students adopting a deep approach to learning tend to be intrinsically motivated (for instance, studying the underlying theory of mathematical formulas in order to develop a better understanding for the topic) [65, 130]. Surface learning, in contrast, relies on learning by heart, studying without reflecting and treating material as unlinked pieces of knowledge. Students adopting a deep approach become actively interested in the course content, whereas those exhibiting surface learning feel pressured and find it difficult to make sense of new material. Surface learners tend to be extrinsically motivated (for instance, learning mathematical formulas by heart merely to get a good grade) [65, 130].

Clearly, the two types of motivation and learning play a role in students' willingness to learn new topics and thus also their attitudes towards new teaching approaches. For instance, a student who adopts a deep approach to learning mathematics and wants to

learn for her own sake may not have anything against learning a new teaching approach; a student who, on the other hand, is only interested in passing the course and adopts a surface approach would probably not see any point in spending time on also getting familiar with a new approach.

2.2 Nature of Computer Science

One could argue that CS of today is not the same as it was twenty years ago, but rather that the technological advances, the Internet, the mobilization, the new threats and security issues have rendered the discipline a rather different one. One could therefore contend that CS students in the 21st century need to learn different things than did their peers in the 1980s and 1990s; that students of today need to learn how to apply, design and understand practice, not how to reason in the small, use logic and understand theory.

However, looking at it from another perspective, CS is still a *science* based on fundamentals that do not change, regardless of the rapid development of the technology and its applications. In order for the applications to keep developing, somebody needs to study and advance the underlying science.

The latest curriculum guidelines published by the ACM, *Computing Curricula 2001* (CC2001) [104],¹ summarize the body of knowledge in CS and the number of hours suggested for core topics ranging from none to 43 hours. Hours in this context refer to the time required for presenting the material in a traditional lecture format. This time does not include any out-of-class work, which according to the guidelines should be three times the time spent in class on a certain topic. A unit listed as 4 hours will thus in practice involve 16 hours of work (4 in class and 12 outside).

The core topics are defined as topics that are required of all CS students. Of the 14 bodies of knowledge, discrete structures and programming fundamentals are considered the top two; according to the recommendations, the former should contain at least 43 hours of core topics and the latter 38 hours. These two topics are also the only ones for which all listed subtopics are considered core ones. Programming and (discrete) mathematics can thus be seen as fundamental parts of CS.

2.3 Programming in Education

Role of programming in CS The implementation of CC2001 is divided into three levels: introductory, intermediate and advanced courses. Six strategies for how to implement the introductory courses are given, and programming makes up an essential part of each of these strategies. It thus appears clear that programming is considered an important component of the introductory curriculum. The authors of CC2001 point out that this has been the case throughout the history of the discipline, as the development

¹CC2001 is one of the many curricula developed for CS education within the ACM. The first curricula guidelines were published in 1968.

of programming skills has been the primary emphasis in most introductory courses. The authors assert that this has many positive aspects, starting from as simple a reason as the fact that students like programming. In addition, programming is an essential skill that must be mastered by CS students, and placing it early in the curriculum ensures that students have the necessary facility with programming when they enroll in intermediate and advanced courses. Programming courses also meet external (employer) needs.

CC2001 also mentions some negative aspects of this focus on programming in the introductory curriculum. First, it may introduce the risk of students getting a limited sense of what the discipline is about. A strong emphasis on practical programming skills also leads to theoretical topics being deferred until later. Whereas including theory early could help students understand the practical material, it may not have the same “immediate relevance” when introduced later on. For majors, the lack of theory early on may reinforce the misconception that mathematics and theoretical topics are irrelevant. Non-majors, on the other hand, who only take introductory courses, will not get exposed to the underlying theoretical foundations at all. CC2001 also states that programming courses often focus on syntax and the characteristics of a programming language, which results in students spending the majority of their attention on these relatively unimportant details rather than on developing the underlying algorithmic skills.

State of programming education CS has traditionally been considered a university topic [24] and is well developed at that level of education [204]. Studying CS in general, and programming in particular, however, develops skills such as logical reasoning, algorithmic thinking, design and structured problem solving, which are valuable to anyone, not only computer scientists. Consequently, CS could be considered an important part of general education, and along these lines model curricula for high school CS were developed already in the 1990s, for instance, by the ACM [137] and in Israel [74].

In Finland, on the other hand, the development has gone in the opposite direction: while CS used to be included in the high school core curriculum, this is no longer the case. Starting with the curriculum published in 1994, CS is covered in other subjects through the use of the computer and its applications [71, 108]. As a result, the only experience most high school graduates have from CS (both at home and at school) is based on the practical use of computers. Such experience does not develop the important meta skills mentioned above. Considering also that 1) the goal of Finnish high schools is to prepare students for future studies and life in general and 2) research indicates that pre-university CS courses increase the chance of students succeeding at university level [5, 37, 89, 201, 213], the lack of CS education in Finnish high schools can be considered a severe shortage.

Novice difficulties Regardless of the level of education, programming is commonly considered a challenging topic to learn and hence also to teach. When learning to program, novices are not only faced with solving a problem; they also have to learn the syntax and semantics of a programming language and how to express solutions in a for-

mat that the computer can understand. During this process many difficulties may be encountered for a variety of reasons. In the following we will review some of the research that has been conducted with regard to novice programmers' difficulties. Other good overviews are, for instance, given in the collection of studies on the novice programmer edited by Spohrer and Soloway [197] and the extensive review of novice difficulties compiled by Robins et al. [174].

When learning to program, students create their own understanding as they interpret the learning material. The resulting understanding is commonly referred to as a *mental model* [167, 215]. Having viable mental models is important when learning to program, as non-viable models tend to result in different kinds of misunderstandings [123]. Research has, however, indicated that novice programmers tend to have non-viable mental models of, for instance, the computer [109], basic programming concepts such as value and reference assignment [123], and more advanced topics like recursion [80].

Spohrer and Soloway [196] found that a few bug types constitute the majority of the errors made by novice programmers. They called the main problems "plan composition problems", meaning that students find it difficult to put the pieces of a program together correctly. Nine plausible reasons for this type of difficulties are identified; for instance, mapping from natural language to a programming one and not dealing with unexpected cases in the implementation. Another, not as frequent, reason for bugs was "construct based problems", that is, difficulties related to learning the semantics of programming language constructs. These can occur, for instance, when a novice assumes that the computer will interpret a construct in the same manner he or she does intuitively. Another reason for construct based problems is inconsistency; students may understand how a construct works in one situation and assume it will work similarly in other, different situations as well.

More recently, novice difficulties have been investigated in multi-national and multi-institutional studies [120, 131]. The results from these indicate that beginning programmers find it difficult to abstract a problem from its description and lack the skills needed to trace the execution of short pieces of code after their first programming course. The latter suggests that not only is producing code difficult, but also reading it.

An international survey among beginning programmers [114] indicated that students find it difficult to detect bugs and handle errors in their programs. This is argued to be an example of students' inability to understand a program in its entirety, instead of as only a collection of code fragments. Similar findings have been presented in the literature [121], suggesting that novices read a program line by line instead of as a whole.

The results of the survey [114] also showed that students perceive abstract concepts, such as recursion, pointers and references and abstract data types, as more difficult than other topics. Novices' difficulties with abstraction has also been demonstrated in other studies [36]. Beginning students tend to have concrete models of programming code in the form of operations and control flow, with only little domain-level knowledge in the form of function and data flow [48].

Research suggests that novices demonstrate several misunderstanding when it comes to central object-oriented concepts like object and class [60]. Another study showed that

students consider object-oriented concepts and design more difficult to understand than topics such as variables, methods and arrays [36]. Object-oriented programming has been recognized as a Threshold Concept in CS education [31]. Threshold Concepts is a framework which has recently received more attention in CS, and used to refer to parts of the curriculum that students must understand in order to make progress in the field [176]. These concepts are potentially troublesome for students [31].

Research has also pinpointed difficulties with specific programming constructs. Variables have traditionally been considered difficult among students [87, 181, 193], who for instance believe that a variable can contain two values at the same time. It is common for students to see variables as boxes, which is an analogy unsuitable in the programming context. Parameters in general and parameter passing in particular also pose problems to novice programmers [72, 124]. As an example of a difficulty related to a control structure, one can mention the so called “off-by-one” bug, which is a common boundary problem when dealing with iteration [196].

Formal methods The prevalent way of teaching programming relies on a “trial and error” approach, where focus is on getting students to produce code using a given programming language. Testing and debugging programs are important parts of learning to program, but as the famous quote by Dijkstra [58] states, “[p]rogram testing can be used to show the presence of bugs, but never to show their absence!” (p. 6). No matter how many tests the program may have passed successfully, just one test on which it fails is enough to show that it is not correct. At this point we feel the need to point out what we mean by a program being correct in this thesis, since studies have shown that students and teachers may interpret program correctness quite differently [111]. Students tend to conceptualize correctness as relative; for instance, a program is considered correct as long as it does at least what it is supposed to do, regardless if it in addition also does something else. In this work, however, correctness is used as an absolute concept; a program is correct only if its semantics is consistent with regard to a given specification.

As software grows in scale and functionality, so does its complexity. When systems become more complex, the likelihood of defects is increased. Some of these errors are harmless, whereas others may cause a disastrous loss in terms of time, money, and even human life. While testing can come a long way in finding defects, there are other techniques that can be employed to minimize the risk of introducing errors in the programs in the first place.

Developments in the field of formal methods were made already in the 1960s. Naur [144] introduced the invariant concept in 1966 when he showed how the correctness of algorithms could be established using “General Snapshots”. These snapshots stated the relationship between program variables and were written as comments in the code using informal language. Around the same time, a similar idea was independently developed by Floyd [73], who proposed correctness verification of flowcharts using logical notation and assertions (loop invariants) attached to points in a program. Hoare [101] continued on Floyd’s work by making a key change in the presentation showing how mathematical logic, rather than flowcharts, could be used to verify program code. Dijkstra [55] used the

work by Naur, Floyd and Hoare as the basis when proposing a correct-by-construction approach to program development, where the program and proof should be developed simultaneously, with the latter leading the way. For a historical review of the developments in the area of program verification, the reader is referred to the article by Jones [106].

Despite the early developments, formal methods are still commonly underutilized in industry and in education. The situation in industry can be seen as the result of many factors [90, 161]. Some would argue that the popular, but evidently false, belief that formal methods can guarantee zero-defect software is one of the reasons why these techniques have not achieved wider acceptance [90]. Another common belief is that using formal methods is expensive and time consuming [91]. There is also a big gap between the theoreticians and the practitioners: “[f]or the theoretician programs are mathematical objects that never fail if we can just get their specification right and verify the code. For the practitioner formal methods use obscure notation, deal with toy examples, and will never scale.” [161, p. 63] In addition, software professionals must meet tight deadlines and anything that poses even the least risk of schedule delays is out of the question. Industry has well established processes for software development, and any change can be seen as a risk not worth taking.

Formal methods can be quite cost-effective, as they make it possible to develop low-defect software while reducing the need for testing, debugging and changes. Most errors are introduced in software during the early phases of development, and the later they are discovered, the more expensive it is to fix them. Using formal methods, potential errors can be discovered early on, hence reducing the overall cost of a project [91]. For formal methods to be attractive to a larger extent in industry, there is, however, a need for suitable tools and adequate examples [92]. In addition, the techniques must fit into the existing work processes and be transparent to the user [161]. Finally, it is important to remember that formal methods do not have to replace other methods in software development. Rather, they should be used in places where they add value and profit from synergy effects with other activities [91].

In education, formal methods are similarly eschewed for several reasons. One of the main ones is the perceived difficulty level of the techniques, requiring mathematical maturity and skills that most students do not have [200]. This aspect is further discussed below (section 2.4).

Another reason is the typical curriculum strategy at CS departments where it is common to divide courses into “areas of ‘theory’ and ‘practice’... [which] causes both faculty and students to view the theory of computing as separate and distinct from the practice of computing” [6, p. 73]. In addition, most textbooks on introductory programming do not discuss program correctness or related concepts such as loop invariants [9]. Consequently, students get only little exposure to correctness concepts [7] and get the impression that formal techniques are only applicable in theoretical courses [136].

In addition, students are more likely to be motivated by gaining skills that they believe are relevant, valued in industry and bring immediate benefits [169]. The nature of programming may reduce the experienced need for formal methods, as it is completely

possible to break design rules when constructing software and still end up with a working program. When well-known companies can get away with releasing buggy software, it is not easy to convince novice CS students that they need to learn how to write correct programs. Taken together, this lack of interest carries on to what Dijkstra coined as “mental resistance” among students towards formal methods [81].

Student attitudes and industry needs are also used by CS faculty as arguments against teaching formal methods [179]. If such teachers do teach something related to the topic, they will most likely not be enthusiastic or show a true interest in what they are teaching. And a “I don’t really believe in this, I just have to teach it” mentality hardly goes far in having a positive impact on students’ attitudes to the topic at hand.

Aspect of notation Regardless of whether one takes a practical or a more formal approach to software development, one needs a notation with a given syntax and semantics. Many formal approaches rely on the use of logical notation, whereas a programming language is needed in practical programming courses in order to convert algorithms into executable code.

What language to use in introductory programming courses has been the topic of extensive debate throughout the history of programming education. Despite the popularity of commercial languages such as Java and C++, there has been much debate surrounding the suitability of these languages in education, especially at introductory level (for instance, [27, 44, 88, 172]). These types of languages are intended to be used for large-scale program development and have not been developed specifically with education in mind.

Milbrandt [139] suggests that a programming language to be used in education should be easy to learn, structured in design, universal in use and powerful in computing capacity. The language should also have a simple syntax, use meaningful keywords, provide easy I/O handling and output formatting. In addition, Milbrandt points out the importance of immediate feedback and useful diagnostic tools. Many of these criteria are echoed by McIver and Conway [134], who list seven ways in which introductory programming languages make teaching of introductory programming difficult. They also put forward seven principles of programming language design aiming to assist in developing good pedagogical languages.

Pascal and Logo were both designed with education in mind, fulfilling many of the characteristics mentioned by Milbrandt, and many studies have indicated the suitability of these languages in education (for instance, [187, 189]). Unfortunately, both languages suffer from drawbacks that have led to decreasing significance over the years. Lately, scripting languages, such as Python² and Ruby on Rails³, have received more attention in educational settings, as these are considered more flexible and easier to use [212].

Despite the availability of more pedagogically suitable languages, the commercial ones remain popular. A census of introductory programming courses within Australia and New Zealand [50] revealed that industry relevance – not pedagogical considerations – was the

²Python is further discussed in chapter 4.

³<http://www.rubyonrails.org/>

most prominent reason why instructors chose their current teaching language. This suggests that academics perceive pressure to choose a language that may be marketable to students, even if the students themselves would not be aware of what is required in industry. According to the TIOBE Programming Community Index⁴ for September 2009, Java and C are the top two languages if looking at, for instance, the number of skilled engineers worldwide and third party vendors. According to the census, these languages are likely also used in education.

To address the problems associated with teaching Java in introductory courses, the ACM called together the ACM Java Task Force [173]. In its final report, the Task Force states that CS education faces a general challenge arising from “two self-reinforcing characteristics of modern programming languages that have a profoundly negative effect on pedagogy:” [173, p. 1].

- *Complexity*: The number of details that students must master ... has grown much faster than the corresponding number of high-level concepts.
- *Instability*: The languages, APIs, and tools on which introductory computer science education depends are changing more rapidly than they have in the past.

Some of the issues recognized by the Java Task Force have been remedied over time as new versions of Java have been released, whereas others still remain a concern. Since Java is one of the most widely used languages in programming education [50, 154], it appears relevant for us to review some of the problems discussed by the Java Task Force in greater detail:

- Scale is considered the most serious problem for those teaching Java or any other industrial-strength language [173]. Even though the languages themselves may be quite simple, they are not sufficient for writing useful programs; hence, students also need to use various API libraries. The authors argue that the “existence of these huge libraries makes it difficult for students and teachers to learn the language without suffering from conceptual overload.” [173, p. 10] Unfortunately, this problem is likely to grow as each new release of, for instance, Java is larger than the previous ones.
- All Java programs must include a `main` method. In order to write their first program, students are thus faced with keywords (`public`, `static`, `void`), the `String` class and array notation. In cases where Java is used to taught imperative programming, the need to define a class at the beginning of the program poses a similar problem. Since these concepts cannot be explained at the beginning of a programming course, teachers find themselves in a situation where they have to tell their students that they just have to use these words as something of a recipe for now, whereas their meaning will become clear later on. The members of the Java Task Force point out that the compulsory `main` method also raises another problem,

⁴<http://www.tiobe.com/tpci.htm>

which is potentially even more difficult to deal with. Since no object is involved when the `main` method is called, all other methods also need to be declared static; this is said to undermine the idea of object-orientation.

- In Java, there is a need to handle exceptions in even simple programs. As an example the Java Task Force mentions the static `Thread.sleep` method, which makes it possible for the programmer to delay execution at some point in a program. To pause execution for five seconds, it is, however, not enough to write `Thread.sleep(5000)`. Since the method can throw an `InterruptedException`, this needs to be handled using a `try-catch` block:

```
try {
    Thread.sleep(5000);
}
catch (InterruptedException e) {
    React to the exception or simply ignore it
}
```

- Java also lacks a simple mechanism for accepting input from the keyboard. The introduction of the `Scanner` class in Java 5.0 improved the situation, but reading user input still requires a separate class to be instantiated:

```
Scanner s = new Scanner(System.in);
int n = s.nextInt(); // Reads an integer from the keyboard
```

All in all, Java is a popular, powerful language, but also quite verbose as it enforces notational overhead that has nothing to do with learning to think algorithmically or developing structured programs. The Java Task Force, like many others, has therefore developed classes and/or packages providing common functionalities while hiding some of the underlying notation.

Question of paradigm When choosing a programming language, one also needs to make a decision with regard to what programming paradigm to use. The programming paradigm influences the way of thinking about programming, and affects, for instance, programming style and coding technique. The four fundamental paradigms are the imperative, the object-oriented, the functional and the logical paradigms. Concepts from each of these are important for a CS major to learn [180]. For the introductory programming course, however, most discussion in recent years has revolved around the imperative and the object-oriented paradigms, i.e. whether to adopt an objects-later or an objects-first approach [34].

Some of the argued benefits of the imperative approach are, for instance, that it is easier to understand and makes it quite straightforward to translate algorithms into code. The object-oriented paradigm, on the other hand, is considered superior, for instance, due to its popularity. Since it has to be introduced at some point, the proponents argue that

it should be taught from the very beginning to avoid difficulties arising from paradigm shifts [53].

The review of novice difficulties above pointed out problems faced when learning to program using an imperative (for instance, related to the use of subroutines and control structures) as well as an object-oriented (for instance, related to the high abstraction level) approach. The object-oriented principles may also take much time to teach, hence leaving less time for basic concepts and traditional algorithms [79, 102]. In addition, to model a program at an abstract level without first knowing the programming techniques needed to implement it is considered similar to “designing knitting models without knowing the knitting techniques required to produce basic knitting patterns.” [180, p. 257]

Despite the argued benefits and drawbacks of the two paradigms, the results of a recent empirical study [61] showed no difference with regard to student learning between an objects-first and an objects-later approach. The findings also indicated that the topics found difficult by the students were the same regardless of paradigm. Consequently, the authors concluded that the difficulties with learning to program lie beyond the choice of paradigm.

2.4 Mathematics in Education

Role of mathematics in CS The importance of mathematics for computer scientists has been debated over the years. Arguments have been made in favor of a less mathematically rigorous CS curriculum [117], and at the same time educators have raised concern over the seemingly math-phobic direction that CS education has taken [205]. Many reports stress the importance of mathematics to the CS curriculum [23, 98, 99, 104]. In addition to the debate on the extent to which mathematics should be taught in CS, there have also been discussions on what type of mathematics should be taught [166].

CC2001 was the first curricula guidelines to introduce discrete structures as an introductory course [99], stating that “[d]iscrete structures is foundational material for CS. ... more and more sophisticated analysis techniques are being brought to bear on practical problems. To understand the computational techniques of the future, today’s students will need a strong background in discrete structures.” [104, p. 86]

In general terms, mathematical thinking can be defined as “[a]pplying mathematical techniques, concepts, and processes, either explicitly or implicitly, in the solution of problems; in other words, mathematical modes of thought that help us solve problems in any domain.” [99, p. 117] More specifically, an increased emphasis on mathematical thinking in introductory CS courses may be beneficial for students by, for instance: [99]

- enhancing their ability to think broad and outside the box,
- developing clarity and precision of thought,
- providing confidence in using symbols, mathematical notation and abstraction,

- increasing preciseness in problem analysis and modeling,
- developing their ability to apply formal techniques in design, specifications etc.,
- aiding in performing informal and formal correctness arguments,
- enhancing their ability to reason from the unknown to the known, and
- improving their communication skills using formal notation.

Promoting mathematical thinking in the CS curriculum does not necessarily imply that more mathematics courses should be introduced; doing so could lead to a reinforcement of the view that mathematics and CS are distinct topics [99]. Rather, mathematics needs to be integrated with the CS courses in a way that makes visible the relationships and dependencies between the two.

State of mathematics education Compared to CS, mathematics is taught extensively in Finnish high schools. Students are offered the choice of two different mathematics syllabi with different foci [71]: the basic syllabus includes six compulsory courses (two elective) and focuses on developing the capabilities needed to “use mathematics in different situations in life and in further studies” (p. 119), whereas the advanced syllabus including ten compulsory courses (three elective) focuses on learning to “understand the nature of mathematical knowledge” (p. 122).⁵ The advanced syllabus is practically the norm for students enrolling for university studies in, for instance, mathematics, CS, medicine, engineering and physics.

Despite the extensive training provided in mathematics, Finnish high school graduates no longer possess the same mathematical skills as they did only a few decades ago [129]. Many explanations can be found for this trend. Over the years, the courses in the advanced syllabus have been lightened and the requirements have been lowered [115, 128]. At the same time, the focus of education has changed. Instead of learning to derive and interpret formulas and understand their meaning, high school mathematics has been left at the level of inputting values in formulas that are given for free. This was acknowledged as a big challenge already in the late 1990s [127]. The change of focus can at least partly also be seen as a result of the drop in capability levels in early education. For instance, Nveri [145] has found that younger childrens’ ability to perform simple calculations in their head or with pen and paper dropped substantially in the time period 1981-2003. She argues that there are two explanations for this development: changes made to the mathematics curriculum and the introduction of calculators in the classroom. If students entering high school have problems with even basic calculations it is obvious that high school mathematics has to take off at that point. Consequently, less time is left for going through more advanced topics in detail.

One of the most frequently criticized lack in high school mathematics is that of proof and logic [17, 93, 97, 185]. Despite the large number of courses offered within

⁵For a listing of the courses, see Appendix 2.

the advanced syllabus, proof and formal reasoning is only mentioned in the learning objectives for one: an elective course on logic and number theory. This is also the only course that introduces logical notation and truth values [71].

As a natural result of the lack of proper training in logic and proof at high school level, a majority of students who enroll for university level studies have little, if any, background in these essential topics. University mathematics, both for majors and minors, is not about inserting values in formulas, but deals with using formalisms and mathematical modes of thought – something school mathematics have not properly prepared students for. This is not a problem only in Finland. For instance, the report on the *Grand Challenges in Computing Education* states that mathematics education is facing large problems as "[s]tudents are less confident than ever in their mathematical abilities ... Many would claim that the matter is exacerbated at the point when students embark on higher education and this is caused by a gulf that exists between the expectation of academic staff and the reality of student ability; the latter is often caused by changes to school curricula that seem to be invisible to university staff." [133, p. 14]

Although being able to reason rigorously and comfortably in mathematics is important in CS, many students thus show little understanding for and interest in mathematics in general, and formal notation, logic and proofs in particular. For instance, Almström [6] found that novice CS students experience more difficulty with the concepts of mathematical logic than with other CS concepts. Gries [83] notes that students exhibit poor reasoning skills even after having completed several mathematics courses, and still fear mathematics and notation. In addition, he claims that proof construction remains a mystery to most students.

Unsystematic notations Just as syntax is inherent to programming, there is a well established notation for formulas etc. in mathematics. This is, however, not the case for how mathematical solutions and proofs are to be presented.

Although avoided, logic still abounds in high school mathematics, but it is hidden behind informal and unsystematic notations. Consider as an example two common ways of expressing solutions to an equation: the plus-minus ($x = \pm 4$) and the comma ($x = 4, x = -4$) notations. These do not clearly state whether the notations stand for “and” or “or”, and could thus be interpreted as “ x is both 4 and -4 at the same time”. If instead expressed using logical disjunction ($x = 4 \vee x = -4$), the result would have a precise and unambiguous meaning.

Similarly, many high school mathematics topics involve logically quantified expressions, but the formal quantifiers (\forall, \exists) are not explicitly used. Arithmetic quantifiers (Σ, Π , etc) are also commonly hidden as these are expressed using ellipsis (“...”); this notation can, however, easily be used incorrectly even by mistake, for instance by concealing too many terms and thus not revealing the pattern correctly.

As an example of a hidden logical quantifier, consider the statement “the global maxima of a function is its largest value in the entire function domain”. Similarly, an even number can be informally defined as “an integer that is a multiple of two”. This type of statements are commonplace in high school mathematics, and while their formal cor-

respondences are clear to teachers, students may not see the precise interpretations as clearly. The corresponding formal definitions, on the other hand, are exact and unambiguous:

- “the function f has a global maxima in point x_0 ” = $(\forall x \cdot f(x_0) \geq f(x))$ ⁶
- “ n is an even number” = $(\exists k \in \mathbb{Z} \cdot n = 2k)$

Taken together, these examples suggest that the use of unsystematic notations is unsafe as it is prone to result in ambiguity and mistakes. Instead of hiding the logic, its explicit use makes mathematical expressions more precise, shorter and clearer. When introducing logic we also get access to a set of inference rules which can be used in transformations and manipulations. No such precise rules are available when using natural language and informal descriptions.

Even more concerning than the hiding of logical notation, is its inconsistent and incorrect use. In the following, we will look at two examples⁷ taken from material for a web based mathematics course in the advanced syllabus, provided by the *Finnish National Board of Education* within the “*Etälukio*” project.⁸ The examples have been freely translated to English.

The first example contains an error in the second step: if simplifying the discriminant in the quadratic formula $((-2)^2 - 4 \cdot 4 \cdot (-8))$, we get $4 + 128$, i.e. 132, which does not lead us to the suggested solution $t = \pm 2$. Rather, we arrive at the solution $t = \frac{1 \pm \sqrt{33}}{4}$. The reader is asked to disregard this unfortunate error, as errors of this type are not the focus of the discussion here.

⁶Throughout this thesis we will use the following notation for expressing quantifiers: $(\#x : p(x) \cdot q(x))$, where $\#$ denotes a specific quantifier, $p(x)$ the domain and $q(x)$ the expression to be quantified.

⁷http://www.oph.fi/etalukio/pitka_matematiikka/kurssi2/maa2_esim8.html (Esimerkki 1), http://www.oph.fi/etalukio/pitka_matematiikka/kurssi2/maa2_esim9.html (Esimerkki 3)

⁸<http://www.oph.fi/etalukio/english.html>

Example 1: Solve the equation $4x^4 - 2x^2 - 8 = 0$.

This is a biquadratic equation, which is converted into a quadratic one using the substitution $t = x^2$. The resulting equation is solved using the quadratic formula.

$$4t^2 - 2t - 8 = 0$$

$$t = \frac{2 \pm \sqrt{(-2)^2 - 4 \cdot 4 \cdot (-8)}}{2 \cdot 4}$$

$$t = \pm 2$$

The solutions to the original equation are arrived at by returning to the variable x and solving the corresponding equations.

$$x^2 = 2 \vee x^2 = -2$$

$$\Rightarrow x = \pm\sqrt{2}$$

There are several problems with this example. The most eye-catching one is the use of implication in the final step. This is a severe error, as implication makes it possible to get too many solutions ($p \Rightarrow p \vee q$). Thus, we could have arrived at a larger solution set, containing incorrect solutions.

The convention commonly adopted in equation solving is to list equivalent versions of the original equation beneath each other, without the need to write out the equivalence symbol explicitly. However, in the example above, this convention is broken in the final step when implication is introduced. This inconsistent use of explicit relationships can clearly be confusing to students: where should a relationship be given and when can it be left out?

The disjunction on the second to last line should be simplified as follows:

$$x^2 = 2 \vee x^2 = -2 \Leftrightarrow x^2 = 2 \vee F \Leftrightarrow x^2 = 2$$

The second disjunct seems to magically disappear, which may lead to further confusion among students (“Which disjunct was used to arrive at the final answer?”, “Where did the other disjunct go, and why?”). Without additional comments or introduction of the truth value *False*, weaker students may even be led to believe that both disjuncts are used: the first ($x^2 = 2$) to get the solution ($x = \sqrt{2}$) by removing the exponent on the left side and adding a square root to the right side, and the second ($x^2 = -2$) to arrive at the solution ($x = -\sqrt{2}$) similarly by getting rid of the exponent and introducing a square root.

Example 2: Solve the equation $2x^3 - 4x^2 - 6x = 0$.

We first factorize the polynomial by $2x$ and then use the zero product rule.

$$2x^3 - 4x^2 - 6x = 0$$

$$\Rightarrow 2x(x^2 - 2x - 3) = 0$$

$$\Rightarrow 2x = 0 \vee x^2 - 2x - 3 = 0$$

The solution to the first equation is $x = 0$, and we get the solutions to the second one using the quadratic formula.

$$x = \frac{2 \pm \sqrt{2^2 - 4 \cdot 1 \cdot (-3)}}{2 \cdot 1}$$

$$\Rightarrow x = -1 \vee x = 3$$

The final answer is thus $x = 0$, $x = -1$ or $x = 3$.

Again, implication is used instead of equivalence, just as in the previous example. In addition, it here occurs in more places, adding to the confusion regarding when relationships have to be stated; both examples are on equation solving, yet the same symbol is used in different ways. Moreover, using disjunction the first solution ($x = 0$) could easily have been kept along when solving the quadratic equation, thus keeping the entire solution in one single chain and reducing the risk of forgetting solutions along the way:

$$2x = 0 \vee x^2 - 2x - 3 = 0 \Leftrightarrow x = 0 \vee x = \frac{2 \pm \sqrt{2^2 - 4 \cdot 1 \cdot (-3)}}{2 \cdot 1} \Leftrightarrow x = 0 \vee x = -1 \vee x = 3$$

These two examples show that if teachers are not aware of the logical interpretation of common symbols and instead rely on their intuitive interpretation, they are likely to use the symbols in the incorrect way. For instance, they may use implication as a symbol for the phrase “the next step is”, which is not how it should be interpreted. Likewise, the careless use of logic (for instance, by leaving out a disjunct without explaining why, or using relations inconsistently) is harmful, as it is prone to cause confusion among students. The confusion can occur either right away (for a student who is already familiar with logical relations) or later on (when the student faces, for instance, implication in another course). If students face a handful of this kind of issues in every mathematics course they take, the cumulative effect can be detrimental.

Lack of structure In addition to the unsystematic notation used to hide the logic, mathematical solutions are commonly also characterized by a lack of clear structure.

Mathematics teachers typically adopt personal presentation styles, and students either follow their teacher's examples or come up with their own ways of writing solutions. Research has nevertheless suggested that a standard format could be of great assistance when learning mathematics [75, p. 70]:

if ordinary students are to make genuine progress in mathematics, almost all need standard templates of this kind to provide a framework

- (a) within which their solutions can be presented and checked,
- (b) by means of which they can be expected to organise a sequence of steps in a way that makes plain to the reader the validity of the final conclusion, and
- (c) through which their understanding of proof, and their acceptance of responsibility for identifying and correcting errors can mature.

The lack of a clear notation for how a mathematical solution or proof is to be written results in several problems. A mathematical solution or proof presented without any particular structure is often not given as a coherent whole. Rather, fragments of the solution or the proof are dispersed in several places on the paper, making it difficult to see how they are to be connected. An example of this is seen in Example 2 above, where one solution is taken aside in the middle of the solving process.

Furthermore, there is an issue of implicit information. Unfortunately, students are not used to justify their solutions [59] as they are usually asked to explain their reasoning only when they have made an error, while the need to justify correctly solved problems is de-emphasized [38]. Without the explanations, the reasoning that drives the solution forward remains implicit [59, 116]. This is especially problematic in the mathematics classroom, as arguments and explanations that the teacher might have given orally do not "get caught" in students' notes. With only the main steps available, much information that might have aided in understanding and reconstructing the solution is left out [110]. Justifications are not only important to the student, but also to the teacher, as the explanations (not the final answer) make it possible for the teacher to study the growth of mathematical understanding [38, 59, 100, 160, 192].

Similarly, studies have indicated problems in the way proofs are approached and presented in education. As an example we look at the following conventional proof of union distributivity over intersection originally found in a university level mathematics textbook, but here extracted from an article by Gries and Schneider [85].

Example 3: Prove $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

We first show that $A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$. If $x \in A \cup (B \cap C)$, then either $x \in A$ or $x \in B \cap C$. If $x \in A$, then certainly $x \in A \cup B$ and $x \in A \cup C$, so $x \in (A \cup B) \cap (A \cup C)$. On the other hand, if $x \in B \cap C$, then $x \in B$ and $x \in C$, so $x \in A \cup B$ and $x \in A \cup C$, so $x \in (A \cup B) \cap (A \cup C)$. Hence, $A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$.

Conversely, if $y \in (A \cup B) \cap (A \cup C)$, then $y \in A \cup B$ and $y \in A \cup C$. We consider two cases: $y \in A$ and $y \notin A$. If $y \in A$, then $y \in A \cup (B \cap C)$, and this part is done. If $y \notin A$, then, since $y \in A \cup B$ we must have $y \in B$. Similarly, since $y \in A \cup C$ and $y \notin A$, we have $y \in C$. Thus, $y \in B \cap C$, and this implies $y \in A \cup (B \cap C)$. Hence $(A \cup B) \cap (A \cup C) \subseteq A \cup (B \cap C)$. The theorem follows.

Gries and Schneider point out several limitations in this proof. First, it does not state the axioms and theorems that justify the inferences. Moreover, it is not obvious how the facts written in the proof are connected. The proof is verbose, which makes it hard to digest and “get a hang of”. Still, this type of proofs are given as models to students.

Leron [116] argues that the normal linear approach to proofs does not fulfill an important role of mathematical presentations, namely that of communication. In addition, he claims that it is unrealistic for teachers to believe that the traditional approach will do merely because “it was good enough for them during their studies”.

Finally, the lack of a standard format leads to students receiving mixed messages. For instance, Dreyfus [59] notes that many mathematics textbooks offer intuitive explanations in one solution, use examples to clarify another, and give a rigorous proof for yet another one. The differences between these are however not made explicit, and students are left with three different views of what *could* constitute a proof. As a result, students do not know what is expected from them, i.e. they do not know what counts as an acceptable mathematical justification.

In the following three chapters, we will present three approaches aimed at addressing many of the issues discussed in this chapter.

Chapter 3

Evaluated Approach I: Structured Derivations

In this chapter, we describe an alternative approach to teaching mathematics, based on a systematic notation for proofs and derivations and the explicit use of logical notation and inference rules. In addition to describing the approach, we also review potential benefits and challenges of introducing the approach in education.

3.1 Overview

Program verification relies on the ability to construct proofs, and in this context the traditional way of presenting mathematical proofs was deemed insufficient [208]. Program developers needed to create detailed correctness proofs, but what mathematicians considered enough detail was not enough. If proofs of this kind were written in the traditional way, they tended to become “long and verbose, or complicated and laborious” (p. 0). It was therefore concluded that computer scientists would need a new way of presenting proofs if they were to efficiently prove the correctness of anything but simple programs.

An approach that won ground was the *calculational* style [56, 57, 68, 208], which makes it possible to develop and present calculations in a rigorous manner. Dijkstra [56, 57] and Gries and Schneider [82, 83, 85] proposed the introduction of the calculational approach in education already in the 1990s. The arguments for doing so were many [85]. First, the Leibniz principle of substituting equals for equals was deemed easy to teach as it is a style already familiar from elementary algebra. Moreover, avoiding resting a proof on informal arguments in natural language was considered a benefit, as logic makes the proofs unambiguous and less verbose. In addition, the equational logic was found flexible, lending itself to being used in different mathematical domains.

Structured derivations [16, 18, 19, 20, 21] is a proof format developed by Back and von Wright, first as a way for presenting proofs in programming logic, and later adapted to provide a practical approach to presenting proofs and derivations in high school math-

ematics including exact formalisms. A structured derivation has a precise mathematical interpretation, and the syntax and the layout are precisely defined.

The format is a further development of the calculational proof style, where a mechanism for decomposing proofs through the use of *subderivations* has been added. The calculational approach is limited to writing proof fragments, and longer derivations are commonly decomposed into several separate subproofs. Using structured derivations with subderivations, on the other hand, the presentation of a complete proof or solution is kept together, as subproofs can be presented exactly where they are needed. In addition, structured derivations makes it possible to handle assumptions and observations in proofs. In essence, the format can be seen as combining the benefits of the calculational style with the decomposition facilities of natural deduction.

In the following, the approach will be described with a few examples.

Example 1 We start by proving an inequality: $(1 + a)(1 + b)(1 + c) \geq 1 + a + b + c$ when $a, b, c \geq 0$. The structured derivation that proves this theorem is shown below. The derivation starts by introducing the problem on a bulleted line. Next we state the assumptions that we are allowed to make, on lines starting with a dash. In this case, the assumption is that a , b and c are all non-negative. The proof starts with the “ \Vdash ” line.

- Show that $(1 + a)(1 + b)(1 + c) \geq 1 + a + b + c$
- when $a, b, c \geq 0$
- \Vdash {combining $=$ and \geq gives \geq }
 $(1 + a)(1 + b)(1 + c)$
- $=$ {multiply the second and the third parentheses}
 $(1 + a)(1 + b + c + bc)$
- $=$ {multiply the remaining parentheses}
 $1 + b + c + bc + a + ab + ac + abc$
- \geq {subtract the non-negative expression $ab + ac + bc + abc$. The expression is non-negative since a, b, c are positive}
 $1 + a + b + c$

□

The proof transforms an initial expression to some desired form in a stepwise manner, where each step is a relation between two terms. The first step states that $(1 + a)(1 + b)(1 + c) = (1 + a)(1 + b + c + bc)$ and justifies this by a mathematical operation that is known to preserve equality. The proof proceeds in this way step-by-step, until we

finally have shown that the expression that we started with is greater or equal to the last expression, $1 + a + b + c$.

The justification of a derivation step should explain why the indicated relationship holds between the terms. There is plenty of space for writing the justifications, since these are written on separate lines between the terms. This also emphasizes the importance of justifying each step, as equal amount of space is provided for arithmetic expressions as for the justifications. The relation between the terms is written in a separate column to make it clearly visible.

Example 2 The standardized proof format is one of the central ideas of structured derivations. A second important aspect is the use of logical notation and logical reasoning in high school mathematics. Consider as an example finding the real roots of the equation $(x - 1)(x^2 + 1) = 0$.

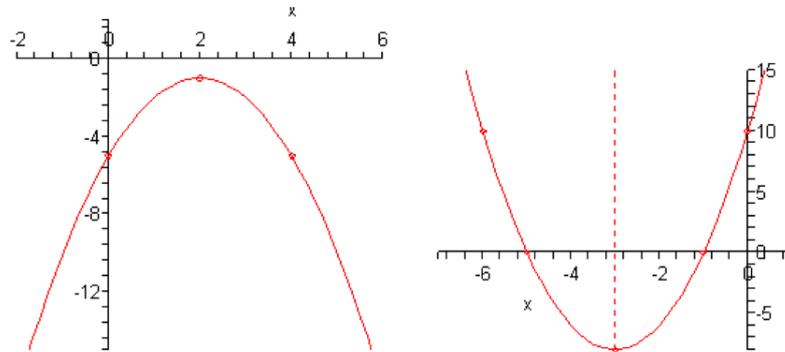
$$\begin{array}{l}
 \bullet \quad (x - 1)(x^2 + 1) = 0 \\
 \Leftrightarrow \quad \{\text{zero product rule: } ab = 0 \Leftrightarrow a = 0 \vee b = 0\} \\
 \quad \quad \quad x - 1 = 0 \vee x^2 + 1 = 0 \\
 \Leftrightarrow \quad \{\text{add 1 to both sides in left disjunct}\} \\
 \quad \quad \quad x = 1 \vee x^2 + 1 = 0 \\
 \Leftrightarrow \quad \{\text{add } -1 \text{ to both sides in right disjunct}\} \\
 \quad \quad \quad x = 1 \vee x^2 = -1 \\
 \Leftrightarrow \quad \{\text{a square is never negative: } a^2 < 0 \Leftrightarrow F\} \\
 \quad \quad \quad x = 1 \vee F \\
 \Leftrightarrow \quad \{\text{disjunction rule: } p \vee F \Leftrightarrow p\} \\
 \quad \quad \quad x = 1
 \end{array}$$

□

As discussed in section 2.4, informal notations and descriptions are commonly used to hide the underlying logic in high school mathematics. The structured derivation above, however, shows how explicit logical notation and logical reasoning can be used both to structure the derivation and to explain the individual reasoning steps. In this specific derivation, disjunction is used to keep two different branches of the derivation together. In the final step, a specific logical inference rule is applied.

Example 3 Whereas the two examples above illustrate shorter derivations, the following one is longer and demonstrates the use of subderivations. We typically use a subderivation when we need to establish an auxiliary fact without interrupting the main line of reasoning. The subderivation is indented one step to the right and starts with a bullet, just like the main derivation. An ellipsis “...” is used to indicate where the main derivation continues.

In this example, the task is to determine the values of constant a such that the function $f(x) = -x^2 + ax + a - 3$ is always negative. The problem is expressed using a universally quantified formula, and the reasoning relies on two figures; this example thus also shows how graphs and other illustrations can be used in a structured derivation.



- Determine the values of constant a such that the function $f(x) = -x^2 + ax + a - 3$ is always negative:

$$(\forall x \cdot -x^2 + ax + a - 3 < 0)$$
- \Leftrightarrow {the function is a parabola that opens downwards as the coefficient for x^2 is negative; such a function is always negative if it does not intersect the x -axis (figure on the left)}

$$(\forall x \cdot -x^2 + ax + a - 3 \neq 0)$$
- \Leftrightarrow {a second degree equation lacks solutions when its discriminant D is less than zero}

$$D < 0$$
- \Leftrightarrow {substitute values for D }
 - Compute the discriminant D :

$$D$$
 - = {the discriminant for the equation $Ax^2 + Bx + C = 0$ is $B^2 - 4AC$ }

$$a^2 - 4(-1)(a - 3)$$

$$= \quad \{\text{simplify}\}$$

$$a^2 + 4a - 12$$

$\dots \quad a^2 + 4a - 12 < 0$

$\Leftrightarrow \quad \{\text{the function } a^2 + 4a - 12 \text{ is a parabola that opens upwards as the coefficient for } a^2 \text{ is positive; such a function is negative between the intersection points with the } x\text{-axis (figure on the right)}\}$

- $\bullet \quad \text{Compute the places where } a^2 + 4a - 12 \text{ intersects the } x\text{-axis:}$

$$a^2 + 4a - 12 = 0$$

$\Leftrightarrow \quad \{\text{square root formula}\}$

$$a = \frac{-4 \pm \sqrt{4^2 - 4 \cdot 1 \cdot (-12)}}{2 \cdot 1}$$

$\Leftrightarrow \quad \{\text{simplify}\}$

$$a = 2 \vee a = -6$$

$\dots \quad -6 < a < 2$

□

The original proposition is reduced to a form that explicitly gives the answer to the problem ($-6 < a < 2$). The main derivation includes two subderivations: one for determining the value of the discriminant and one for solving the second-degree equation that arises.

Hiding and showing subderivations A computer based editor for structured derivations makes it possible to display and hide the subderivations at will. Hiding the subderivations allows us to see the overall structure of the proof, whereas showing them lets us study and edit the proof at a more detailed level.

The following derivation is the same as the one above now with both subderivations hidden.

- $\bullet \quad \text{Determine the values of constant } a \text{ such that the function } f(x) = -x^2 + ax + a - 3 \text{ is always negative:}$

$$(\forall x \cdot -x^2 + ax + a - 3 < 0)$$

$\Leftrightarrow \quad \{\text{the function is a parabola that opens downwards as the coefficient for } x^2 \text{ is negative; such a function is always negative if it does not intersect the } x\text{-axis (figure on the left)}\}$

$$(\forall x \cdot -x^2 + ax + a - 3 \neq 0)$$

\Leftrightarrow {a second degree equation lacks solutions when its discriminant D is less than zero}
 $D < 0$
 \Leftrightarrow {substitute values for D }
 \dots $a^2 + 4a - 12 < 0$
 \Leftrightarrow {the function $a^2 + 4a - 12$ is a parabola that opens upwards as the coefficient for a^2 is positive; such a function is negative between the intersection points with the x -axis (figure on the right)}
 \dots $-6 < a < 2$
 \square

3.2 Using Structured Derivations in Education

Structured derivations was introduced in education for the first time in 2001 when an extensive study was initiated at a Finnish high school [17, 156, 157]. All ten compulsory courses in the advanced mathematics syllabus were rewritten and given using structured derivations. The study was based on a “test group - control group” setting, where the former was taught using structured derivations and the latter was taught in the same way as previously. These groups were followed during their entire high school period (three years). Based on exam grades, students in the test group consistently outperformed the control group in all ten courses as well as in the matriculation exam.¹ In addition, the drop out rate was lower in the test group.

The approach has also been used on a large unbiased set of mathematical problems, as over 200 assignments from the matriculation exam test in advanced mathematics have been solved. This indicates that the approach is feasible for solving problems in a multitude of mathematical domains.

In the remainder of this section, we will discuss the rationale for introducing structured derivations in education, and bring light on some potential challenges. The discussion directly addresses the issues in mathematics education presented in chapter 2.

Introduces formalism and preciseness in mathematics education Using structured derivations for presenting solutions and derivations, formalism and preciseness become natural parts of doing mathematics. As a result, students become used to interpreting and utilizing the language of mathematics and logic. Clearly, the earlier students are familiarized with formal notation, the more confident will they be in applying it.

¹Students graduating from high school in Finland take part in a national level exam, the so-called matriculation exam, which includes at least four tests (the test in mathematics is elective). The aim of the exam is to assess students’ knowledge and skills in the topics covered during their high school studies. A student who passes the exam is eligible for university studies.

Provides a clear standard format Structured derivations introduces a well-defined format, which gives students a concrete model for how solutions and proofs are to be written. A familiar format can act as mental support, giving students belief in their own skills to solve a problem. This can be especially important when considering the common “fear” for proof found among students; using structured derivations, proofs are presented in the same well-known way as simple calculational derivations, which may result in students finding proofs less intimidating.

The format also lets students focus on the solution rather than having to spend time thinking about how to put their thoughts down on paper. In addition, the format has potential to make the presentation of mathematics more consistent in textbooks and in the classroom. Finally, as noted in conjunction with the examples above, a structured derivation keeps the entire derivation in one single chain instead of as a collection of fragments written all over the paper.

Offers better communicated solutions and proofs Compared to how mathematics is traditionally presented in the classroom, a structured derivation includes all the information needed to understand the entire solution. Justifying each step in the derivation means that the final product will contain a documentation of the thinking that the student was engaged in while completing the derivation. Consequently, the resulting derivation becomes easier to read and check. As an example, we rewrite the above proof of union distributivity over intersection (Example 3 in section 2.4) as a structured derivation:

$$\begin{aligned}
 & \bullet \quad y \in A \cup (B \cap C) \\
 = & \quad \{ \text{Definition of } \cup \} \\
 & \quad y \in A \vee y \in B \cap C \\
 = & \quad \{ \text{Definition of } \cap \} \\
 & \quad y \in A \vee (y \in B \wedge y \in C) \\
 = & \quad \{ \text{Distribution of } \vee \text{ over } \wedge \} \\
 & \quad (y \in A \vee y \in B) \wedge (y \in A \vee y \in C) \\
 = & \quad \{ \text{Definition of } \cup \} \\
 & \quad (y \in A \cup B) \wedge (y \in A \cup C) \\
 = & \quad \{ \text{Definition of } \cap \} \\
 & \quad y \in (A \cup B) \cap (A \cup C)
 \end{aligned}$$

□

A teacher who presents a solution in this way on the blackboard explicitly documents the explanations for each step. This makes it easier for students to understand the derivation later, for instance when studying it in order to solve home assignments or when trying to understand something that they did not get during the presentation in class.

Aids in problem solving Students should be guided in how to approach proof construction, and many models have been presented in the literature (for instance, [51, 162]). Research [22], however, indicates that students do not clearly work through any of the suggested phases. In particular, students tend to jump straight into solving the problem without first trying to understand the task. Skipping the important opening phase is not possible when solving problems or constructing proofs using structured derivations as the first step involves listing all assumptions. One can thus hypothesize that this thorough opening phase gives students a better base for continuing.

Introduces new instructional approaches and learning activities The fixed format of structured derivations renders the approach suitable for new types of learning activities. For instance, one can create tasks where students are to fill out the blanks (for instance, left out justifications) in a derivation or put the terms and justifications of a derivation in correct order.

The fixed format also makes structured derivations easily parsable by a computer. The format also opens opportunities related to computer support. Compared to many other subjects, mathematics is one of the subjects that have been least digitized. Representing mathematical notation electronically is not straightforward, since mathematical symbols are not available on regular keyboards. One of the most popular standards for representing mathematical notation is \LaTeX ,² a document mark-up language and typesetting system, which makes it possible to render professional quality mathematical text using a computer. The syntax of \LaTeX is, however, not trivial, and one cannot expect students to learn \LaTeX in their mathematics courses. Several \LaTeX editors are, however, available for making it easier to produce mathematical text. One of these is the “What You See Is What You Mean” (WYSIWYM) document processor LyX .³ Currently, we are working on extending LyX to support structured derivations, by making it possible to, for instance, check the syntax of a derivation as well as hide and open subderivations.

The editor can also be used to implement the new activity types mentioned above electronically, for instance as assignments in a web based mathematics course. In addition, the justifications make examples self-explanatory, which is highly appropriate for online self-study material. Online examples need to be unambiguous and easy to follow, as there is no teacher around to explain the details or answer potential questions. The examples taken from the web based high school mathematics course material presented in section 2.4 hardly fulfill these requirements.

²<http://www.latex-project.org/>

³<http://www.lyx.org>

Using subderivations, online examples can be made even more helpful as they can be presented at different levels of detail, hence providing just in time, on the spot assistance to the students. A student who needs to know more about a given step in a derivation can simply open the subderivation, whereas a student who feels confident about the corresponding step can continue going through the example without revealing the more detailed view.

Potential challenges Students at high school and university level have already been doing mathematics in a certain way for over 10 years. Consequently, they have been initiated into an acceptable practice, which may make them reluctant to changes in the ways that they are taught [132]. Every new approach requires an investment of time and energy in acquiring new skills with no certainty of payoff. If students, in addition, are not unhappy with the current situation, there may be little incentive for change [3]. It therefore seems reasonable to assume that students (at least those who are extrinsically motivated, see the discussion in section 2.1) may express at least some resistance when a new format is introduced.

Clearly, structured derivations also requires additional writing compared to the traditional approach, where no explicit justification are required. This may also be considered negative among the students. Finally, considering students' limited (if any) prior training in using logical notation, there is a possibility of a logic-based approach being problematic, in particular at lower levels of education.

Chapter 4

Evaluated Approach II: Python

In this chapter, we briefly introduce the programming language Python and review potential benefits and challenges of introducing it in introductory programming education.

4.1 Overview

Python is a dynamic object-oriented language, which supports software development in different application domains; according to the official web site (<http://www.python.org>), Python is currently used by large companies such as Google, Nokia and NASA. According to the TIOBE Programming Community Index,¹ Python ranks among the 10 most popular languages and has seen an increase in popularity since the early 2000s.

Some of Python's key features listed on the official web site are as follows:

- clear, readable syntax
- strong introspection capabilities
- intuitive object orientation
- natural expression of procedural code
- full modularity, supporting hierarchical packages
- exception-based error handling
- very high level dynamic data types
- extensive standard library (“batteries included”) and third party modules providing functionality for virtually every task
- extensions and modules easily written in C and C++ embeddable within applications as a scripting interface

¹<http://www.tiobe.com/tpci.htm>

- open source that runs on all major operating systems
- well documented and supported, for instance, by various newsgroups

In the following, we will review some of the features by going through three examples.

Example 1 The following code is a Python implementation of the “Hello world” program, which has historically been the first program illustrated in introductory programming courses.

```
print 'Hello World'
```

Written in Java, the corresponding program would look as follows:

```
class Hello {
    public static void main (String[] args) {
        System.out.println("Hello World");
    }
}
```

As the example shows, the code in Python is short and self-explanatory, whereas the Java version needs a class and method definition before getting to the line where the message is actually printed (see the discussion on notation in section 2.3). In Python, statements are terminated by end of line, whereas lines in Java are terminated by semi-colons.

Example 2 The program given below populates a list with positive numbers input by the user. The program ends when a negative number is encountered, whereby the list is printed.

```
print "Input positive numbers, a negative number ends the program."
# read a number from the keyboard
x = input("Number? ")
# initialize an empty list
my_list = []
# add numbers to the list and read new ones as long as they are positive
while x >= 0:
    my_list = my_list + [x]
    x = input("Number? ")
print my_list
```

Since Python is dynamically typed, there is no need to declare a type for the initialized variables (`x` and `my_list`). As illustrated by the `while` loop, block structures are indicated by indentation. The example also illustrates the use of the built-in function `input`, which is used to read a number from the keyboard. In addition, the program shows the use of the built-in list data structure (dynamic array), which is here expanded with new elements using concatenation (+).

Example 3 The modules found in the standard library provide access to additional functionality in a variety of domains. The program below illustrates the `webbrowser` module, which provides an interface to displaying web sites to users. A simple call to the module's `open` function with a url as its argument, opens the corresponding web page in the default browser. In addition, the program demonstrates the use of the built-in dictionary data structure (associative list). Compared to an ordinary list, which is indexed by a range of integers, a dictionary is an unordered set of “key : value” pairs, where the keys can be of any immutable type. The example also illustrates the built-in function for reading textual data from the keyboard (`raw_input`), the `for` loop, user defined functions (`def...`) and exception handling (`try - except`).

```

# module imports
import webbrowser
import string

# declare a new function, which takes one argument
def printmenu(menu):
    # iterate over all keys in the dictionary and print the
    # corresponding key:value pairs, separated by a colon
    for key in menu.keys():
        print key + ' : ' + menu[key]

def openpage(menu):
    choiceOK = False
    # let the user input choices until a valid one is given
    while not choiceOK:
        try:
            choice = raw_input('\nChoose a site: ')
            # try opening the url-value corresponding to the choice-key
            webbrowser.open(menu[ string.upper(choice) ])
            # terminate the loop if the web site was successfully opened
            choiceOK = True
        except KeyError:
            # if choice is not valid, execute the KeyError except block
            print 'You did not pick a valid alternative.'
        # if another error occurs, execute the generic except block
        except:
            print 'Something went wrong.'

def main():
    # initialize a new dictionary with three key-value pairs
    options = {'G' : 'http://www.google.com',
               'Y' : 'http://www.youtube.com',
               'M' : 'http://www.myspace.com'
              }
    printmenu(options)
    openpage(options)

# execute the program by calling the main function
main()

```

For more information on the history and technical details of Python, the reader is referred to the official web site where extensive documentation is available.

4.2 Using Python in Education

Python was originally designed with educational purposes in mind, and the developer, Guido van Rossum, has even suggested that everybody could master programming using Python [209]. However, when conducting our first study on the use of Python in an introductory programming context, the number of previous similar studies were quite few. Reports found on using Python in education [40, 62, 63, 86, 141, 190, 198, 216] mainly presented the language and its benefits, and in a few cases also some preliminary experiences. Today the situation has changed, and one can find a multitude of articles on teaching Python [1, 2, 25, 64, 79, 140, 147, 148, 151, 165, 182], some of which present empirical findings while others can still be considered discussion papers. In any case, it feels safe to say that Python has become a popular instructional language during the last couple of years.

As seen above (section 2.3), learning to program is associated with many types of difficulties. Clearly, our aim is not to address all of these. The main rationale for choosing Python lies in its simple syntax. Research has shown that the learning results in programming classes depend on the time devoted to actual programming [152]; in order to maximize this time, one should avoid having to “waste” valuable time on discussing irrelevant language constructs and syntax errors. It seems reasonable to assume that a programming language with a simpler syntax could address this problem and potentially also bring other benefits.

The reader might have noted that all example programs above are imperative to their nature. As noted in connection with the review of the difficulties in section 2.3, the programming paradigm also plays an important role when making decisions on how to teach programming. Although studies suggest that student learning is not affected [61], the choice of paradigm influences how the course will be taught. In our work we have chosen the imperative paradigm mainly because it was the paradigm used previously in the introductory programming course at one of the high schools where we conducted the studies. We did not want to make any other changes in addition to introducing a new language. Also, as the introductory course at the Department of Information Technologies at Åbo Akademi University followed an imperative approach, doing the same in our studies would make it possible for us to use any potentially positive results as the base for making changes to the university course. It should, however, be noted that Python supports object-orientation and has also been used for teaching objects-first [79].

With this rationale (teach imperative programming using Python) as the starting point, we will now review Python with regard to the list of features characteristic for a programming language suitable for teaching discussed in section 2.3.

Easy to use The Python standard distribution comes with a text editor (*IDLE*), and a large amount of tutorials, books, course material, exercises, assignments and documentation is available on the web. In addition, the extensive standard library provides a range of immediately available data structures, constants and functions, which goes a long way in creating programs in an introductory course. In Java, on the other hand, as the Java Task Force [173] points out, classes and packages need to be imported even to create simple programs. Python provides a large collection of modules offering additional functionality, which can be used later on in the introductory course to introduce interesting topics.

Immediate feedback The interpreter enables fast and interactive demonstration of programming concepts, which makes it possible for the teacher to illustrate constructs in isolation without having to take any surrounding code into account. In addition, the interpreter provides immediate feedback on potential errors, which makes it easy for students to experiment with different constructs and see how they work.

Simple syntax Python has a short syntax and is also dynamically typed, which further reduces the notation. The simple syntax also eliminates troublesome errors for beginning programmers related to, for instance, placement of semi-colons, bracketing and indentation. Compared to languages such as Java or C++, the difference is quite obvious. Although this can be seen already in the first example given above (“Hello World”), the difference becomes even more notable when looking at somewhat longer programs, while still staying at novice level. In order to illustrate this difference, a Java version corresponding to the third Python example is given in Appendix 1 on page 111-112.

Structured in design Python programs resemble pseudo code and use indentation to delimit block structures. The latter helps avoid errors from left out braces (*{code block}*), which can be a problem among novices learning, for instance, C++ and Java. If we, as an example, had the following `while` loop in Java

```
while (x < 0) {
    System.out.println(x + “ is negative”);
    System.out.println(“Input a positive number: “);
    x = in.nextInt();
}
```

but left out the braces, only the first line (`System.out.println(x + “ is negative”);`) would be considered belonging to the `while` block. Consequently, we would get an infinite loop whenever the initial number input by the user was negative. In Python, the corresponding code would execute as expected, as the indentation determines the block. The use of indentation has also been shown to function as an aid to comprehension (for instance, [138, 149]).

Easy I/O and output formatting As noted in section 2.3, I/O has traditionally been rather laborious in Java, especially if not using external wrapper packages. Although the introduction of the `Scanner` class improved the situation, it first needs to be imported and instantiated, again dealing with object oriented concepts that cannot be explained in a course on imperative programming. As we saw in the examples above, Python makes it possible to read from the keyboard using two simple functions (`input` and `raw_input`). Output to the screen is accomplished using the built-in `print` statement, and Python also provides convenient output formatting using format strings. Basic file I/O (reading and writing) is also achieved using a few built-in functions.

Powerful in computing capacity Interpreted languages make it time saving to write and execute short programs, whereas large programs might suffer from a loss in performance due to increased memory consumption and running times. A comparative study of seven programming languages [164] has, however, shown that this is not the case for Python. Rather, Python was found to be a worthwhile alternative to C, C++, Java, Perl, Rexx and Tcl, as it demonstrated an average performance, by no means consuming the most memory or taking the longest to execute. In addition, the results showed that Python programs were only half the length of programs written in traditional languages (C, C++, Java). Similarly, compared to programs written in these languages, Python programs only took half the time to construct.

The comparative study is quite old, and since the performance of the languages may have changed as they have evolved, a corresponding study conducted today might produce other results. A list of comparisons of Python to other languages including more recent results can be found on the Python web site.² These are, however, informal to their nature and we have not been able to find any official study corresponding to the one cited above. Nevertheless, when dealing with introductory programming courses the programs created are rather small, and language performance should not pose a problem.

Potential challenges As noted in section 2.3, industry demands and language popularity are important aspects to both students and teachers. Novices, who do not yet have any background in programming but have heard of languages such as Java and C++, may be disappointed when finding out that they will be taught to program using another language. Student attitudes can thus become an issue despite the fact that Python is extensively used in industry and is not a language of interest merely in academic settings.

One could also argue that by teaching Python first instead of, for instance, Java, students will have to relearn most of the things when transitioning to a commercial language later on. Finally, dynamic typing was mentioned as an advantage, but might also turn out to be a drawback, since the possibility of assigning different types to the same variable might make programs more prone to errors.

²<http://wiki.python.org/moin/LanguageComparisons>

Chapter 5

Evaluated Approach III: Invariant Based Programming

In this chapter, we describe an invariant based approach to teaching programming where focus is on linking theory and practice. We also review potential benefits and challenges of introducing the approach in introductory education.

5.1 Overview

Invariant based programming is a diagrammatic approach to constructing correct programs, where not only pre- and postconditions, but also loop invariants are written before the actual code. The approach is not new, it was studied already in the 1970s by Reynolds [170] and Back [10, 11]. Similar ideas were also proposed by van Emden [207]. In 2004, Back [12] revisited the topic and has since worked on developing invariant based programming into a practical hands-on method.

In invariant based programming, a program is constructed and verified at the same time. The notion of an invariant is generalized to a *situation*: a collection of constraints that describes the set of states that satisfies these constraints. An invariant based program may have several situations and is not restricted to single-entry, single-exit control structures.

Example We will here exemplify the work flow for developing invariant based programs by constructing a program that finds the largest element in an array. We use a cursor to traverse the array from left to right, and for each position we check if the current element is larger than the largest element so far. If this is the case, we update the largest element and advance the cursor. If the current element is smaller than the largest so far, we merely advance the cursor.

We start by drawing figures illustrating the basic data structures involved and how these will change during execution of the algorithm. This is an essential step of the work

flow, as the figures describe the algorithm at work and thus help the programmer identify the situations and get a feeling for the behavior of the algorithm.

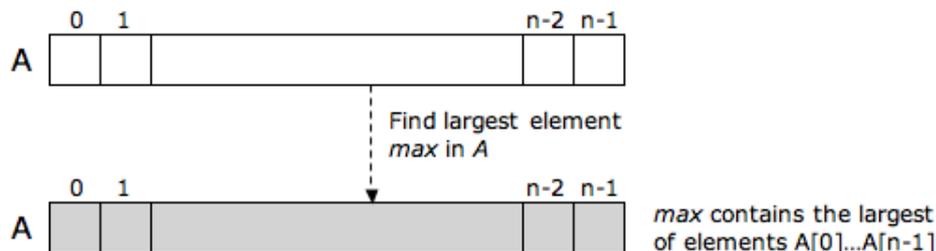


Figure 5.1: Visualization of the specification

The initial figure (Figure 5.1) illustrates the specification (pre- and postcondition) and helps us identify the initial and final situations. The initial situation is quite straightforward; all we know in the beginning is that we have an array A with indices $0..n - 1$. To express the final situation we need to consider what it means that “ max contains the largest of all elements in the array A ”. Clearly, this gives us that $A[i] \leq max$ for all indices i in the array, i.e. $(\forall i : 0 \leq i < n \cdot A[i] \leq max)$. This is, however, not enough. With such a postcondition the program could choose an arbitrary large value and assign it to max ; if this value was larger than all elements in the array, the postcondition would be satisfied regardless if max was one of the elements in the array or not. We thus need to add an additional constraint, which states that max must be one of the elements in the array. As a result, we arrive at the following postcondition:

$$(\forall i : 0 \leq i < n \cdot A[i] \leq max) \wedge (\exists j : 0 \leq j < n \cdot A[j] = max)$$

As situations are sets of states, the final situation is a subset of the initial situation where an additional constraint, “ max contains the largest element in the array A ”, is satisfied. We use an Euler-like diagram, a *nested invariant diagram*, to represent the program and the strengthening of situations. Our first diagram is shown in Figure 5.2 on the facing page. Since situations are nested, all constraints in an outer set also hold in all of its subsets and need therefore not be repeated (for instance, $n : integer$ holds in both the initial and the final situation). Dashed arrows are used to indicate the computation that we want to define and are labeled with a potential guard and the variables that may be changed during the computation.

In the same manner as the final situation was identified as a subset of the initial one, we introduce new situations by adding new constraints to the ones present in the more general situations. We further develop the figure of the algorithm at work by introducing the intermediate situation (Figure 5.3).

As is shown in the corresponding diagram (Figure 5.4), this newly inserted situation is a subset (i.e. a constrained version) of the initial situation. While dashed arrows illustrate

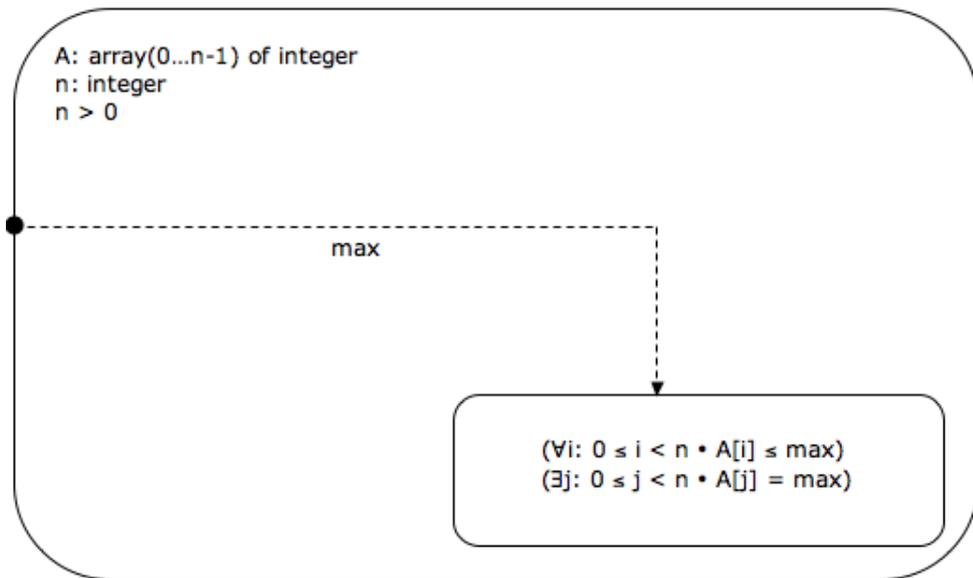


Figure 5.2: Invariant diagram illustrating the initial and final situations

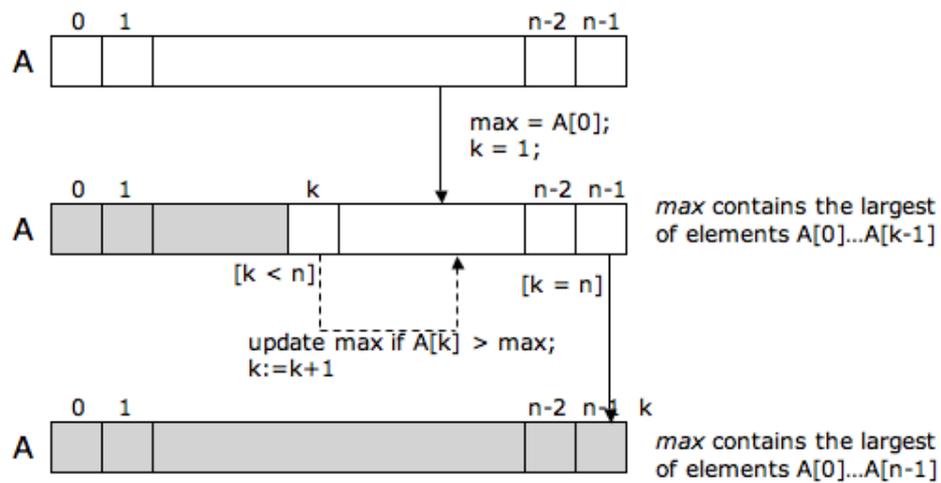


Figure 5.3: The algorithm at work with the intermediate situation inserted

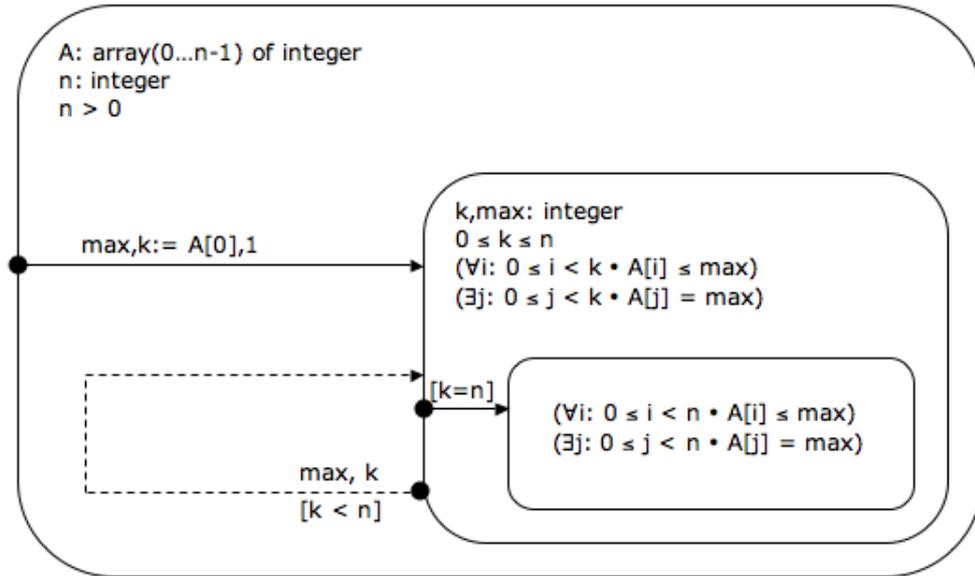


Figure 5.4: Invariant diagram with the intermediate situation inserted

what we want to accomplish, we use solid ones to indicate computations that we have already planned and defined. We call the solid arrows *transitions*. Each transition is labeled with a potential guard (enclosed in brackets) and the program statements to be executed when the transition is carried out.

In order to arrive at our final program (Figure 5.6), we again use a figure to help us get an idea of what should be accomplished during the loop transition (Figure 5.5).

For each transition that we add to the diagram, we need to check that it preserves the situations as follows: assume that we initiate execution in the source situation of a transition and that all the constraints hold for the starting state. Also assume that we reach some target situation after executing the statements for the transition (there may be more than one possible target situation). Then all the constraints of the target situation must hold for the final state. We say that a program is *consistent* if this is true for all transitions in the diagram, i.e. for any situations P and Q in the diagram and for any transition from P to Q .

When all situations and transitions have been added, we still need to check that no infinite loops exist, i.e. that the program *terminates*. We deal with termination by introducing a termination function t for each loop. The integer function t is a termination function for a given situation if 1) it is bounded from below in the indicated situation and 2) its value is decreased before re-entering the situation. The termination function is written in the right upper hand corner of the corresponding situation (Figure 5.6).

Taken together, consistency and termination guarantee that execution proceeds without failures and that each loop terminates. This does, however, not rule out the chance of execution terminating in a non-final situation. If we want to exclude such possibilities

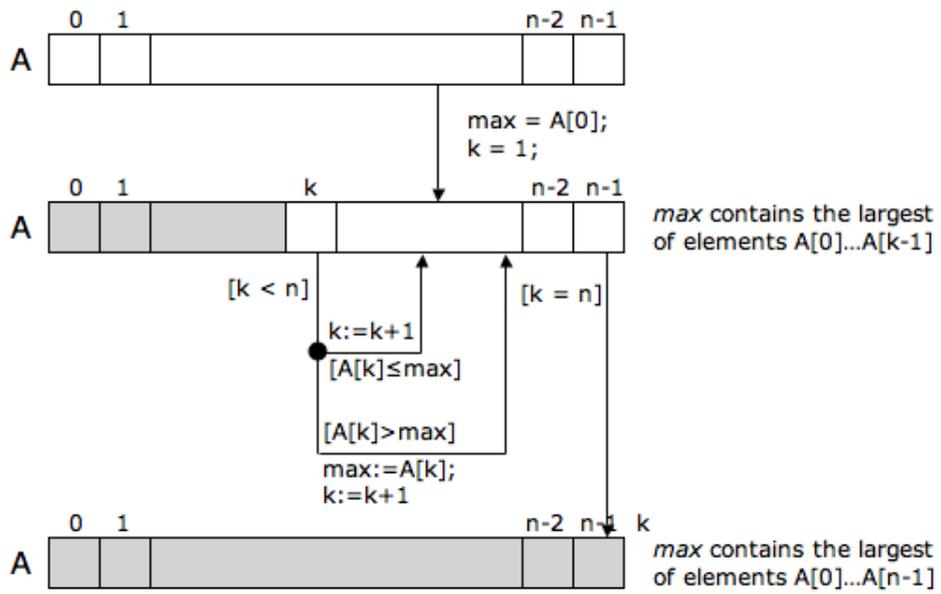


Figure 5.5: Complete algorithm at work

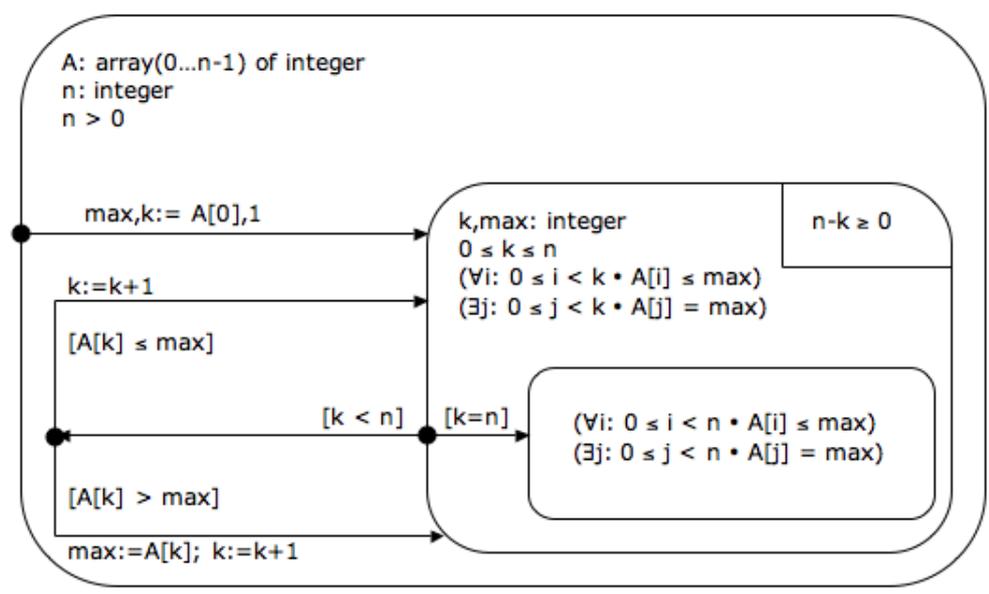


Figure 5.6: Final invariant diagram

we need to check that the program is *live*, i.e. that termination only occurs in final situations. In practice, this means that we must make sure that for all situations (except for final one(s)), there is at least one enabled transition.

An invariant based program is totally correct if it satisfies the three criteria above, i.e. 1) is consistent, 2) terminates and 3) is live. For a more in-depth presentation of invariant based programming as a method, see [12, 13, 14].

Example Proof Although consistency checks can be done informally (for instance, as going through the verification conditions as check lists), writing down formal proofs makes it easier to keep track of all details and make sure that nothing is overlooked.

Structured derivations is well suited for constructing proofs for invariant based programs.¹ In order to shorten the proof, we number each condition, guard and action directly in the diagram (Figure 5.7). Consequently, we avoid the need to explicitly write out all assumptions in text at the beginning of every proof. In the following, we give an example of how the consistency of a transition is verified using a structured derivation. This specific derivation verifies the bolded loop transition (guarded by $k < n \wedge A[k] > \max$).

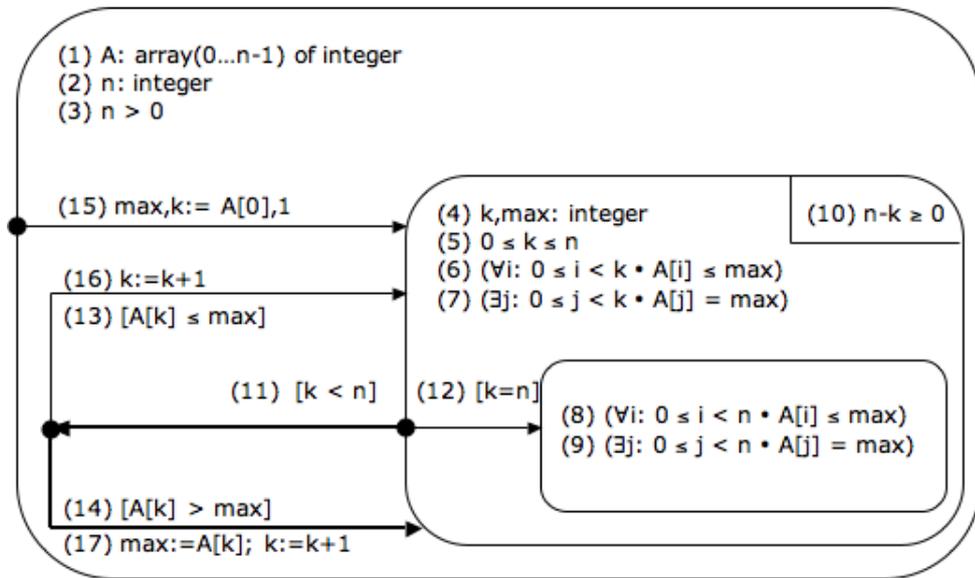


Figure 5.7: Numbered invariant diagram to aid proof construction

To prove that this transition preserves consistency we need to check that all constraints of the target situation hold for the final state if we assume that all constraints of the start situation hold for the initial state. For this specific transition, execution starts

¹In the educational context in which we have introduced invariant based programming, students have already been familiar with structured derivations from a previous course on logic. Thus, using the same proof format was a natural choice.

in the intermediate situation, and we can thus assume that both the precondition and the invariant hold. Execution stops in the same situation, and we therefore need to prove that the precondition and invariant still hold after executing the transition, i.e. for the updated values of max and k . We express the updated values using prime: max' and k' .

We get the precondition P (1-3 in the diagram), invariant I (4-7) and the given loop transition $Loop$ (11, 14, 17) from the diagram. Given these abbreviations, the verification condition for the transition becomes the following:

$$P \wedge I \wedge Loop \Rightarrow (P \wedge I)[max, k := max', k']$$

The resulting structured derivation is shown below.

- Prove $(P \wedge I)[max, k := max', k']$ when
 - $[1] \wedge \dots \wedge [7] \wedge [11] \wedge [14] \wedge [17]$ hold
 - ⊢ { Conjunction introduction rule ($P \wedge I$ corresponds to (1) $\wedge \dots \wedge$ (7) in the diagram)}
- (1' - 3') $A : array(0..n - 1) \text{ of } integer \wedge n : integer \wedge n > 0$
 \Leftrightarrow { [1], [2], [3] }
 T
- (4') $k', max' : integer$
 \Leftrightarrow { [17] }
 $k + 1, A[k] : integer$
 \Leftrightarrow { [1], [4] }
 T
- (5') $0 \leq k' \leq n$
 \Leftrightarrow { [17] }
 $0 \leq k + 1 \leq n$
 \Leftrightarrow { [5], [11] }
 T
- (6') $(\forall i : 0 \leq i < k' \cdot A[i] \leq max')$
 \Leftrightarrow { [17] }
 $(\forall i : 0 \leq i < k + 1 \cdot A[i] \leq A[k])$
 \Leftrightarrow { Range split }
 $(\forall i : 0 \leq i < k \cdot A[i] \leq A[k]) \wedge (\forall i : i = k \cdot A[i] \leq A[k])$
 \Leftrightarrow { One-point rule }

$$\begin{aligned}
& (\forall i : 0 \leq i < k \cdot A[i] \leq A[k]) \wedge A[k] \leq A[k] \\
\Leftrightarrow & \{ A[k] \leq A[k] \equiv T \} \\
& (\forall i : 0 \leq i < k \cdot A[i] \leq A[k]) \wedge T \\
\Leftrightarrow & \{ P \wedge T \equiv P \} \\
& (\forall i : 0 \leq i < k \cdot A[i] \leq A[k]) \\
\Leftarrow & \{ [6], [14] \} \\
& (\forall i : 0 \leq i < k \cdot A[i] \leq \mathit{max}) \\
\Leftrightarrow & \{ [6] \} \\
& T \\
(7') & (\exists j : 0 \leq j < k' \cdot A[j] = \mathit{max}') \\
\Leftrightarrow & \{ [17] \} \\
& (\exists j : 0 \leq j < k + 1 \cdot A[j] = A[k]) \\
\Leftarrow & \{ \text{Witness rule, } j = k \} \\
& T
\end{aligned}$$

□

Using structured derivations, we can also express the entire correctness proof as one single derivation where the different subproofs are kept together using subderivations. This would give us the following proof structure for a program containing one loop (i.e. having one intermediate situation):

- Prove that program X is correct
- ⊨ {Conjunction introduction rule}
 - Prove consistency
 - ⊨ {Conjunction introduction rule}
 - Prove consistency of transition 1
 - Prove consistency of transition 2
 - ⋮
 - Prove consistency of transition n
 - Prove termination
 - ⊨ {Conjunction introduction rule}
 - Prove that the termination function is bounded from below
 - Prove that the value of the termination function decreases in every iteration

- Prove liveness
- ⊢ {Conjunction introduction rule}
 - Prove liveness for the start situation
 - Prove liveness for the intermediate situation

□

If we have several loops, we need to prove that each of these terminates and that the liveness property is established for every intermediate situation.

5.2 Using Invariant Based Programming in Education

Many attempts to introduce a more formal approach to programming in introductory CS education have been made (for instance, [7, 54, 135, 194, 211]), but as noted in section 2.4, convincing students of the value of formal methods is a challenge. If formal methods are not used in industry, it seems reasonable to question why it should be included in education. One hope is that the needs in industry and education could reinforce each other [95]: “If industry is interested in formal methods, students and universities have financial incentives to focus on the area. If typical computer scientists/engineers have formal methods in their toolkit, they may be inspired to apply them in their jobs.” (p. 3)

Formal methods in education hence constitutes an important area open for improvement. The main motivation for introducing invariant based programming in education is to address some of the afore-mentioned issues, aiming at 1) changing the image of formal methods as being difficult, uninteresting and of no use in practice and 2) showing that formal reasoning about program correctness can in fact be done in a practical manner with only basic logic skills.

In the remainder of this section, we will discuss some hypothesized benefits and potential challenges related to the introduction of invariant based programming in education.

Rests on basic logic Invariant based programming differs from many other formal approaches previously used in education in that it rests on basic logic, and does not require any advanced mathematics.

Sense of locality Invariant based programs are built, and verified, in a stepwise manner. As each transition is checked when it is added to the diagram, errors are removed when they are introduced instead of being accumulated to a later point of time. Checking consistency throughout program development requires relatively little effort, compared to doing the same when the diagram has been completed. The continuous checks also emphasize correctness aspects throughout program construction. In addition, when making a change in a program, only the invariants and transitions involved need to be re-checked.

Diagrammatic representation In contrast to, for instance, the approaches presented by Hoare [101] and Dijkstra [55], invariant based programming is diagrammatic. A variety of graphical programming/pseudocode formats have been proposed in the literature [30, 178], and all of these have one common goal: “to provide a clear picture of the structure and semantics of the program through a combination of graphical constructions and some additional textual annotations.” [178, p. 3] To our knowledge these have, however, focused on representing control flow and data flow. In invariant based programming, on the other hand, programs are described from another perspective as the approach emphasizes the invariant properties of the program data structures, and thus makes it possible to reason about the correctness of the constructed program in a rather straightforward manner while also improving readability. This is accomplished without sacrificing either clarity or expressiveness of the diagrams.

Enhanced usability with tool support Invariant based programs can be constructed using only pen and paper, and in many cases this is the best way for initially drafting a program. However, even small programs generate a large number of verification conditions, giving rise to proofs similar to the one presented above. Many of these verification conditions are rather trivial and could be automatically proved or greatly simplified by theorem provers. In addition, the risk of making mistakes in manual proofs and specifications can be diminished with a proper tool. Tool support has also been found to be critical for the successful integration of formal techniques in CS education [214].

*SOCOS*² [15] is a graphical programming environment developed within our research group at Åbo Akademi University for the construction and verification of invariant based programs. It analyzes invariant diagrams semantically and generates correctness conditions, which are sent to external proof tools. *SOCOS* also makes it possible to execute invariant diagrams directly (by compiling them into Python code), without first having to hand translate them into an existing programming language. That is, when having constructed and verified an invariant based program using *SOCOS*, students actually have something they can run; not only a formal derivation of a program that they know should work in theory (since they have verified it), but that they cannot execute in practice.

Links theory and practice An important benefit lies in giving students a concrete link between theory and practice in CS by integrating mathematical proof with programming. As seen above (on page 50) the proof of a complete invariant based program can be expressed as one single structured derivation.

Potential challenges As always when introducing a new approach in education, student resistance may become an issue. This can be particularly challenging if students have prior negative attitudes towards similar approaches. Although one of our aims of introducing invariant based programming in education is to improve students’ attitudes

²<http://mde.abo.fi/confluence/display/SOCOS>

towards the theory of CS and show how it can be used in practice, we cannot be sure whether students will see the same benefits. Another difficulty may arise from identifying the situations in general, and sufficiently strong invariants in particular.

Most novice CS students have no previous background in program verification, but are used to focus on code. These students are thus faced with a new way of thinking when starting to work with invariant based programming, where the specification and the invariants become most important. Similarly, constructing manual proofs in a programming context is new, and may turn out to be challenging.

Chapter 6

Research Framework

In this chapter, we briefly discuss the research context of our work as well as the methods used when conducting the studies.

6.1 Education Research in CS

Education reforms cannot be realized merely in the form of planning. Consequently, scientifically evaluated instruction and educational material have become increasingly important in all disciplines [35, 69]. Unfortunately, the common perspective is to view research and teaching as dialectic [42], and knowledge created through research is usually unavailable to the teaching community. In addition, studies have shown that teachers tend to be negative towards innovations suggested by research, especially if the results have been gained from other than experimental research [107]. Teachers thus seem unaware of the fact that experiments are not necessarily the best way to investigate educational phenomena, as other factors that are difficult to operationalize and take into account, such as student motivation, classroom settings and student background, also affect the results. The resistance among teachers to adapt their tuition according to research results needs to be overcome, because without a connection between research results and teaching practice, curriculum development cannot progress [43].

The nature of CS education research has been described by Fincher as follows [41]:

As educators, we all have a professional interest in the teaching that we deliver and the quality of learning that takes place in our students. We write lectures, devise activities and create assessments. At the same time institutions have quality assurance mechanisms – student evaluation questionnaires, periodic reviews, external examiners – which address the standard of our provision. This is part and parcel of the normal business of being a lecturer.

When interest goes beyond the individual classroom, to examine the efficacy of specific approaches or techniques, to judge the generalisability or transferability of outcomes, to work to understand whether there are a set

of conditions or abilities that pre-dispose for success in CS, then we move towards “CS Education Research”. (p. 336-337)

CS education research is a rather young and still emerging field characterized by a high level of cross-disciplinarity [26]. In addition to CS, the field encompasses aspects from fields such as cognitive psychology, education and sociology, to mention a few. Consequently, there are no clear guidelines for how research should be conducted [70].

Fincher and Petre [70] describe the characteristics of publications that can be considered research in a CS education context as having two dimensions: that of “rationale, argumentation or ‘theory’ ” (p. 2) and that of empirical evidence.

- Many researchers in the field of CS education have a background in CS. For such researchers, the term “theory” may be problematic [70]. Whereas a “theory” in the natural sciences is predictive and causal, this is not the case when working in an educational context. Instead, a “theory” in education and other social sciences deals with the underlying reasons for found effects while the causes may remain unresolved. Such *explanatory theories* aim at explaining observed behaviors, by making explicit factors that affect human behavior in certain contexts. By using an explanatory theory, the researcher brings added value and depth to his/her results. Without the use of explanatory theories, the work risks to be of a mere descriptive nature, answering the question *what* instead of the question *why*.
- In order to assess the success of an educational change, some kind of evidence needs to be gathered; generally this knowledge can only be derived from experience as the “scope of diverse needs is often very wide, the problems to be addressed are usually ill-specified, the effectiveness of proposed interventions is mostly unknown beforehand, and the eventual success is highly dependent on implementation processes in a broad variety of contexts.” [206, p. 2]. This type of educational research is thus typically grounded in empiricism. *Empirical research* is concerned with making observations, interpretations and drawing conclusions in some authentic context [186]. The evidence gathered can be either such that is verifiable by observation and direct experience, or data that strongly confirm a theory or a hypothesis [45].

The *Committee on Scientific Principles for Education Research* has described the process of scientific research in education using the following guiding principles [191, p. 52]:

- Pose significant questions that can be investigated empirically.
- Link research to relevant theory.
- Use methods that permit direct investigation of the question.
- Provide a coherent and explicit chain of reasoning.
- Replicate and generalize across studies.
- Disclose research to encourage professional scrutiny and critique.

The committee points out that the principles should not be seen as an algorithm for research, but rather as offering a code of conduct and general framework indicating how research can be approached. It also notes that an individual study may not fulfill all principles, and Fincher and Petre [70] remark that the principles can be unevenly weighted depending on the nature and quality of the study at hand. They also point out that replication is not necessarily possible in educational research, which usually is conducted in complex contexts that are not easily reproduced. Instead, one can seek to obtain supporting results by, for instance, reproduction of the study by another researcher in similar settings.

6.2 Development as a Research Activity

Richey and Klein [171] have described *developmental research* as an activity in which the goal is to

create knowledge grounded in data systematically derived from practice. It is a pragmatic type of research that offers a way to test “theory” that has been only hypothesized and to validate practice that has been perpetuated essentially through unchallenged tradition. In addition, it is a way to establish new procedures, techniques, and tools based upon a methodical analysis of specific cases. As such, developmental research can have a function of either creating generalizable conclusions or statements of law, or producing context-specific knowledge that serves a problem solving function. (p. 24)

In educational contexts the emphasis of developmental research is usually on aiding the development of a new instructional product or program through a cyclic process of development and research [206]. This can be seen as serving a two-fold purpose. First, the aim is to develop prototypes of instructional products, including an empirical evaluation of their usefulness and effectiveness. As a result, the research efforts also provide methodological suggestions for how such products could be designed and evaluated. Eventually, the cyclic process leads to “theoretically and empirically founded products, learning processes of the developers and (local) instructional theories” (p. 5).

The overall goal can thus be further divided into two separate subgoals: 1) to provide suggestions for how the quality of the material/product/program/etc. can be optimized, and 2) to create, formulate and test design principles (What should the material/product/program/etc. look like?, How should the material/product/program/etc. be developed?). The former of the two subgoals aims at generating practical results in a given context, whereas the latter is more focused on producing scientific knowledge of a generalizable nature.

Richey and Klein [171] stress that the main result strived for in developmental research projects is not knowledge in general, but rather knowledge that is usable by practitioners. Neither should developmental research be seen as a one-way effort where research results are “translated” into a form that can be used in education [206]. The

converse direction is also important: if, for instance, a teacher using a research-based product or program concludes that parts of it cannot be successfully used in a certain context, this is a finding that should be fed back to the researchers. Such results are valuable contributions to a revision of the theory and knowledge on which the product or program is built.

The relevance of this type of research is usually judged based on questions such as “Is the problem addressed a common one? Is it realistic?” Richey and Klein [171] use the traditional stages of planning and conducting research to describe how a developmental research project can be carried out:

- *Problem definition:* At this stage the focus of the problem and the scope of the study are established. An important question to consider is whether the research will address all parts of the instruction development process or only a fraction of these (for instance, the development/evaluation/revision part). As developmental research projects usually focus on emerging procedures, techniques and tools, a solid base to formulate hypotheses on may not be found in the literature. Consequently, developmental research is typically centered around research questions instead of hypotheses.
- *Literature review:* As with any type of research, the goal of the literature review in a developmental research project is to establish the context and the conceptual framework. If no directly relevant literature can be found, the researcher needs to identify literature that can be used to inform the research decisions at an indirect level.
- *Research procedures:* The authentic settings in which developmental research projects are conducted enhance the credibility of the results, but also mean that this kind of studies usually takes more time to complete than others. Whereas simulated situations can be controlled, real-world situations are open for changes and unanticipated events. The authentic settings also give rise to some methodological dilemmas. First, the researcher needs to take external and contextual variables into account. In addition, the participation of the researcher in the study can be problematic; still, this is a common situation, and any potential problems can be accounted for by taking measures to ensure objectivity. The data collection, analysis and synthesis steps are similar to those utilized in other types of research.

6.3 Research Design in This Thesis

Developmental research The developmental research methodology fits the purposes of this thesis¹, as the aim is to evaluate instructional approaches in authentic settings, with the possibility to refine and further develop the approaches based on the evaluation

¹In some of the publications, the term *action research* is used instead. This term can be seen as another label for developmental research [206].

results. Our intention throughout the research has not only been to evaluate the approaches in isolation, but also in connection with each other. For instance, as invariant based programming serves as a concrete link between mathematics and programming, difficulties found in evaluating the invariant based approach can be used to recommend changes in how mathematics or programming is taught. Thus, the research is pragmatic and matches the description of developmental research given above.

As the teaching approaches at hand already exist, the scope of the present research is on the evaluation aspect. This is evident also when reviewing the research questions, which all more or less focus on evaluation. Consequently, this research can be seen as a collection of formative studies aiming at evaluating and thereby improving the way in which the three evaluated approaches are used in education.

Method The methods used in the publications differ from each other to some extent. This is a direct consequence of the nature of the research questions posed in the respective publications. Here, we merely give a brief account of the overall study settings; a more detailed description of what data and methods have been used for evaluating the approaches and the rationale for doing so is given in chapter 7 and in the respective publications.

The data that comprise the basis for the studies have been collected during the years 2004-2007 in four educational contexts: 1) an introductory course on logic at university level, 2) a course on the mathematics of programming at university level, 3) a high school mathematics course, and 4) a high school introductory programming course. The data have been collected through the use of several instruments: questionnaires, exams, assignments and interviews. As the number of students in the groups participating in our studies has been rather low, incorporating qualitative methods has been crucial in order to arrive at new findings and insights. The use of qualitative methods is also common to the developmental research methodology [171]. In addition, only using quantitative methods may not catch all interesting findings, as these methods are fixed around a definite number of variables [43]. For instance, when collecting data using multiple choice questions, each question has a given set of predetermined answers. On the other hand, when using open-ended questions, the respondent will only report on what comes to mind, while multiple choice questions could give a broader coverage. In any case, open-ended questions are important, as they tend to be less leading and hence also more objective than, for instance, multiple choice questions [163].

By using several data sources and analysis methods, the researcher can analyze and describe the same phenomenon from different perspectives. Thus, exams, questionnaires, and interviews can all be used to challenge, confirm, or expand the information gathered [186]. Although individual studies may be problematic when considering trustworthiness and generality, the cumulative weight of evidence derived from several types of data using different methods pointing in the same direction gives the findings a larger degree of trustworthiness. Using several methods and/or sets of data may thus reduce the risk of bias introduced by using only one method and/or set of data. Employing multiple methods also makes it possible to explore diverse research questions; whereas questions looking

to describe a phenomenon are best answered using a qualitative approach, quantitative methods are better at addressing more factual questions [45].

In cases where the data have been quantitative, basic statistical analysis methods (descriptive and inferential) have been used for further investigations. Qualitative data, on the other hand are highly descriptive, and in order to interpret the information, the data first need to be reduced. Qualitative researchers tend to use inductive instead of deductive analysis of data, meaning that findings emerge out of the data rather than as the result of analyzing the data according to an existing framework [153]. In this thesis, we have used content analysis for this purpose. Content analysis is generally used to refer to “any qualitative data reduction and sense-making effort that takes a volume of qualitative material and attempts to identify core consistencies and meanings.” [153, p. 453] Emerging themes can be quantified, and as such, content analysis is suitable for transforming rich data into a form that can be illustrated with numbers and graphs [45].

The basic idea in our studies has been to take rich data collected from students and analyze, reduce and summarize these using categories. Interviews are used in Paper VI and VII, open-ended responses in Paper I - III, V and VII - VIII, and programs or mathematical solutions in Paper II, IV, VI and IX. All of these data types were analyzed in more or less the same manner.² Analysis was initiated by reading through a subset of the data and organizing it into categories, serving as a preliminary coding scheme. Next, all data were analyzed using this scheme. If data that did not fit any of the existing categories were encountered, a new category was created. This, however, only happened a few times; in most cases, the initial set of categories turned out to be sufficient. The categories in our studies have hence been created inductively based on what respondents said (in case of interviews), wrote (in case of open ended questions) or did (in case of errors in programs or mathematical solutions).

The author of this thesis did the initial analysis and the coding in the papers on Python and invariant based programming (Paper VI-IX). The results were discussed and reviewed with the co-authors (or another colleague in case the article had no additional authors). In the articles on structured derivations, one of the co-authors also participated in the creation of the categories and the coding. During the coding phase the two authors discussed any uncertain cases and agreed on a category together.

Coherence with research guidelines The research presented in this thesis fulfills the afore-mentioned two criteria for research discussed by Petre and Fincher [70]: the nine publications build on empirical data, which are analyzed and interpreted in the light of explanatory theories. If, in addition, considering the six guidelines for educational research put forward above, the design of the research presented in this thesis can be summarized as follows:

- *Pose significant questions that can be investigated empirically.* A set of research questions has been developed for each article,³ and each of these has been addressed

²The interviews naturally had to be transcribed prior to analysis.

³Except for some of the background publications as described in section 1.3.

based on results from authentic classroom settings.

- *Link research to relevant theory.* As the studies included in the thesis are separate and aim at answering different questions, each uses its own (explanatory) theory, pertinent to bringing insight into those specific questions. Thus each publication includes the literature review needed in order to help explain and analyze the phenomenon under study.
- *Use methods that permit direct investigation of the question.* As already stated above, the methods used have been carefully chosen based on the questions to be examined.
- *Provide a coherent and explicit chain of reasoning.* Throughout all articles, the aim has been to give a precise description of the research process and provide a thorough review of any background information needed.
- *Replicate and generalize across studies.* As noted above, replication is not easily accomplished in educational settings. Due to the nature and aims of the present work, it was concluded that replication issues were beyond the scope of the research.
- *Disclose research to encourage professional scrutiny and critique.* All articles have been published and are hence available for external judgment. The articles published (or accepted for publication) in international journals or conferences (Paper I-II, Paper IV-IX) have gone through an academic review, whereas Paper III has been submitted for publication and is currently undergoing similar review processes.

Chapter 7

Overview of Publications

In this chapter, we give an overview of the original publications by presenting the research questions investigated, the methods used and the key findings. For a more detailed account of any of these aspects, we refer the reader to the publications at the end of this thesis.

7.1 Structured Derivations in Education

Paper I This publication, *Structured Derivations: A Logic-Based Approach to Teaching Mathematics*, serves as a descriptive background paper on the use of structured derivations in education, setting the stage for the following publications on the topic. The main contribution lies in the presentation of the structured derivations approach, its formal syntax (which has since been updated as a result of the feedback from the other studies) and the rationale for using it in education. Previous findings are summarized and the introduction in the classroom is presented through a detailed description based on our experiences.

In addition, preliminary results on the use of the approach in an introductory CS course on logic are presented. These results indicate that the approach is appreciated by the students, who recognized many of the originally hypothesized benefits. Similarly, some of the potential challenges mentioned in section 3.2 are realized, as the main negative aspects brought up by the students were related to the requirements on justifying every step and an experienced tediousness of using the approach. These initial findings motivated the following studies (Paper II-Paper IV), where more attention is put on the justifications and the benefits and drawbacks experienced by students using the approach.

Paper II The aim of this publication, *Promoting Students' Justification Skills Using Structured Derivations*, is to continue the study on structured derivations in the classroom by focusing on the explicit justifications and student feedback. Two research questions are addressed: “How does the use of structured derivations affect students' justifications?”

and “What advantages and drawbacks do students experience when using structured derivations?”

The data were collected during an elective advanced mathematics course on logic and number theory (about 30 hours in class) at two Finnish high schools during fall 2007. Twenty two (22) students on their final study year participated in the course.

The data analyzed were collected using a pretest, three exams and a mid and post course survey. The pretest included five exercises, which students were to solve while also justifying their reasoning. For each exam, we analyzed three solutions per student, giving us a total of 198 analyzed solutions (22 students * 3 exams * 3 solutions). The findings presented are the result of an analysis of two aspects: the types of justification related errors (JRE) and the frequency of these.

The results from the pretest confirmed that students are not used to justifying their solutions and do not necessarily even know what doing so would entail. Nevertheless, students managed to do well on the justifications in the three exams; a JRE was found in merely 15-20 % of the 66 analyzed assignments for each exam. The students were, in general, positive towards the new approach, and showed particular appreciation for the increase in clarity and understanding. The study also brought some additional light on the nature of the “tediousness” mentioned by students in the pilot study presented in Paper I: apparently, students consider the increased length and time requirements the main drawbacks. The analysis also indicated a lack of completely negative comments, as those starting by pointing out a drawback (“It takes much time,...”, “I don’t like all the writing,...”), all still ended in a positive tone (“... but I understand what I do better”, “... but I make fewer errors”). Student opinions are analyzed to a larger extent in Paper III.

Paper III The aim of this publication, *“It Takes Me Longer But I Understand Better” – Student Feedback on Structured Derivations*, is to further examine students’ reactions to the use of structured derivations. As such, it builds on the preliminary studies presented in Paper I and Paper II. The specific research questions investigated are the following: “What benefits and drawbacks do students experience when having used structured derivations for the first time in a course?” and “Do students at high school and university level experience similar benefits and drawbacks?”

The high school data were collected at the same time as the data for the study presented in Paper II, while the university level data were collected during an introductory CS course on logic in 2007. Hence, the high school students were on their third study year (aged 17-18 years), and had previously taken 9-12 courses in advanced mathematics. The majority of the university students were CS majors in their early twenties and were studying at the university for the first year. Depending on their choice of courses they could have taken one university mathematics course prior to the course on logic. Both the high school and university students were exposed to structured derivations for the first time.

The data analyzed were comprised of answers to four open-ended questions, collected using a post course questionnaire. The focus of the questions was on getting students to

freely express their opinions with regard to the benefits and drawbacks of using structured derivations and the traditional approach respectively. The answers were analyzed in multiple phases in order to obtain high level categories illuminating the benefits and drawbacks of the two approaches. A quantitative analysis of the categorized data made it possible to present the results using charts and to compare high school and university level students' opinions.

In the paper, the benefits and drawbacks of structured derivations are described separately, while those of the traditional approach are covered in the same section. This is a natural consequence of the fact that many of the benefits of the traditional approach were found to be drawbacks of structured derivations and vice versa. Hence, many of the advantages and disadvantages of the traditional approach are already covered in the preceding sections on structured derivations.

The results suggested that students appreciate structured derivations for several reasons, particularly for making solutions clearer, easier to follow and check. In addition, the analysis revealed that the approach has potential to increase students' self-perceived level of understanding. The benefits were all found to be related to the explicit justifications in one way or another. The study also confirmed that students consider the lengthiness and time requirements the main drawbacks. Nevertheless, students find it highly beneficial that it takes the teacher a longer time to go through examples on the blackboard (as they have time to follow along).

The study raised an interesting question for future investigation related to what students mean when they talk about understanding. We were not able to address this question in this study (using, for instance, interviews), as the high school students had already left school to prepare for the matriculation exam at the time when we finished the analysis.¹

Paper IV While working on the previous studies, a question that seemed to come up now and then was “What types of justifications do students give in a solution?” This question is analyzed in this publication, *Student Justifications in High School Mathematics*, in addition to the question “Do the justifications change as the course progresses, and in that case how?”

This study was done in the light of a framework on understanding developed by Skemp [192], where he distinguishes between two types of understanding: relational (“knowing both what to do and why”) and instrumental (“knowing what”, “rules without reason”). People who exhibit an instrumental understanding know how to use a given rule and may think they understand when they actually do not. This was found to be a useful distinction to consider when analyzing students' justifications. Many other frameworks would also have been suitable for this purpose; in the paper we review some of the ones related to understanding in a mathematical context. In addition, there are several

¹The final high school year is shorter in Finland, as students finish school in mid-February in order to start preparing for the matriculation exam, which commences approximately one month later. A few university students also reported on an increase in understanding, but it is questionable whether a couple of interviews would have been sufficient to shed light on the meaning of understanding.

frameworks aiming at classifying learning objectives in general, for instance, Bloom's taxonomy [112] and the SOLO taxonomy [29]. Analyzing student justification in the light of each of these is left as future work.

The data for this study were collected during the same high school course as in Paper II and Paper III. Two assignments from three exams were analyzed, giving us a total of 132 analyzed solutions.

The analysis revealed five main justification types with different characteristics: vague/broad, assumption, rule, procedural description and own explanation. It was concluded that only when students use own explanations can the justification be used "straight off" in evaluating understanding (instrumental or relational). The findings suggested that students mainly use broad and vague justifications in tasks that are familiar (first exam). In the following two exams, new and unfamiliar topics were covered, and hence the frequency of own explanations increased on the expense of more trivial justifications, indicating that the justifications become particularly important as mental tools when adventuring into new terrain.

Similarly to the previous study, this one also raised an interesting question for future work: whether some instrumental justifications could be mapped to relational understanding. This is illustrated by examples given in the article. As in Paper III we were, however, not able to further investigate this question, since the students had already left school to prepare for the matriculation exam when we had completed the analysis.

7.2 Python in Education

Paper V This article, *Why Complicate Things? Introducing Programming in High School Using Python*, serves as a background publication on the use of Python in a high school setting. In addition to giving the rationale for why a simple syntax language like Python could be well suited for educational purposes, the main areas explored are how programming can be introduced at high school level and how suitable Python is for supporting both teachers and students when doing so.

The data were collected during the school year 2004/2005, when 42 high school students completed an introductory programming course using Python. A post-course questionnaire and programs written on the exam were analyzed in order to sketch an image of how high school students managed to learn programming using Python. In addition, a comparison of course grades is presented, where the results of students taking the Python course are compared to those of students taking a corresponding course in Java.

The data analysis revealed that compared to a corresponding course given in Java earlier, less students failed using Python and more students received one of the two highest grades. No problems were found related to the Python syntax; rather, the difficulties reported on were related to abstract topics, such as exception handling and functions. Students with a prior background in programming were positive towards the new language, as they found it more fun and appreciated its simplicity and clarity, rich range of modules and compactness resulting in shorter code.

Paper VI In this publication, *What about a Simple Language? Analyzing the Difficulties in Learning to Program*, the structure and characteristics of programs written on the exam by high school students taking their first course on programming are analyzed. The research questions addressed are the following: “What types of errors can be found in novices’ programs? Do students having learned Python make other types of errors than those who have learned Java, and in that case, in what ways? How does the use of Python as the first language affect the transition to Java?”

The data were collected during two school years: the Java data in 2002/2003 and the Python data in 2004/2005. The only difference between the two groups was the language of instruction; all other factors, for instance, teacher, course contents, environment and time available on the exam, were the same during both years. In total, programs written by 30 Java students and 30 Python students were analyzed. The goal was to find difficulties independent of the language used and such originating from the language. When analyzing the data, the errors were classified as either logical or syntactical. The logical errors were further divided into five subcategories. This categorization was deemed sufficient, since no unclear cases with regard to whether a given error should be considered logical or syntactical were found. Had such cases been encountered, we would have needed to revise the categories.

In addition, eight of the Python students were later interviewed when moving on from Python to Java. The purpose of this follow-up study was to explore the transition from a “simple” language to a more “advanced” one, by investigating whether the Java syntax was problematic for students who had learned to program in Python. Here, focus was thus not on how well students do in learning new topics, but rather a new syntax. The interviews were conducted in the high school facilities, starting with all participants translating a Python program into Java. In addition to the interview questions, the interviewer² also went through the resulting Java program together with the student. The interviews took 25-50 minutes.

The results of the program analysis indicated that students who learn Python make fewer syntax and logic errors than those learning Java. In addition, the interviews revealed no disadvantages from having learned to program using a simple syntax language when moving on to a more complex one. Rather, learning the basic idea and programming concepts using a simple language seems to have potential to make it easier to learn more complex and advanced languages further on.

Paper VII In this publication, *Novices’ Progress in Introductory Programming Courses*, a newly developed instrument called a progress report is used to analyze student progress. A progress report can be seen as a type of learning diary written on paper, aiming at revealing the students’ own opinions and thoughts about their learning. What differentiates it from a traditional diary is that, in addition to calling for self reflection, the report makes it possible for the teacher to evaluate students’ understanding based on how well

²The majority of the interviews were carried out by the author of this thesis. Two interviews were conducted by the course teacher.

they can explain what a given piece of code does. The aim of the publication is to investigate three questions: “How do students understand program code as a whole? How do students understand individual constructs? How do students perceive the difficulty level of different programming topics?”

The progress reports were collected during the school year 2005/2006, when 25 high school students completed an introductory programming course given in Python. We used two progress reports that were handed out after 1/3 and 2/3 of the course respectively. Each report included a piece of code dealing with topics recently covered in the course as well as four questions. First, in the “trace and explain” questions, the students were to trace the code of a short program and in their own words describe 1) what each line of the code does, and 2) what the program as a whole does on a given set of input data. In addition, students were asked to state what they had learned and what had been most difficult so far in the course. The explanations were analyzed and categorized, and the student perceived difficulty level of various programming topics was investigated by looking at both the progress reports and a post course questionnaire.

The main focus of this publication was not on Python, but it still brought interesting light on the use of Python in education. Based on the results from Paper V, where exception handling was considered difficult by the students, changes had been made to the syllabus so that this topic had been moved to the very beginning of the course and was now introduced with variables, output and user input. The results from this study suggested that using Python, topics perceived as difficult, such as exception handling, can be introduced early on thereby resulting in a decrease in the perceived difficulty level.

7.3 Invariant Based Programming in Education

Paper VIII The aim of this article, *Teaching the Construction of Correct Programs Using Invariant Based Programming*, is to serve as a background publication, presenting the approach and initial results from introducing the method in education. The questions explored are the following: “How can invariant based programming be introduced in education? How do students experience learning formal methods using this approach?, How applicable is the use of tool support in the course? What difficulties do students encounter when learning formal methods using this approach?”

The study was conducted in spring 2007, when the invariant based approach was introduced in education as an introductory CS course. Research data were collected using post course questionnaires, assignments, the course exam and interviews.³ In addition to giving a detailed description of the invariant based approach and the course settings, the questionnaire and the interview data were analyzed to provide insight into students’ opinions and the experienced difficulties. Assignments and exam solutions were investigated to find examples of common errors and difficulties.

The results indicated that students learning the invariant based approach find it

³The interviews were not conducted directly after the course exam, but approximately one month later due to practical reasons (vacations, other exams).

interesting, easy to learn and practical. The diagrammatic presentation seems to help students understand how programs work. In addition, students appreciate the locality aspects, which facilitate debugging. While students do not face any difficulties with learning the invariant based syntax, those with prior programming background appear to initially find it challenging not to use a potentially ingrained programming language syntax.

We had expected students to find identifying invariants the most difficult task, but this was not the case. According to student feedback, finding good invariants is not trivial, but writing proofs by hand still seems to be most problematic as doing so requires much time and effort. Moreover, students appear to find it difficult to express conditions correctly using logical notation, which is an essential skill also when constructing proofs.

Paper IX The aim of this publication, *Invariant Based Programming in Education – An Analysis of Student Difficulties*, is to bring further light on the difficulties students face when solving problems using invariant based programming. More specifically, the study focuses on the following research questions: “What kind of errors do novices make when using invariant based programming? Do these errors change as the course progresses, and in that case how? Does the year of study impact on student performance?”

The data for the study consisted of student constructed invariant based programs, which were collected in 2007-2008. One third of the programs was collected during the first half of the course, one third during the second half and the remaining one third was collected from the exam. All in all, 129 programs were analyzed, aiming at finding categories describing the main errors found in students’ programs. In addition, the invariant related errors were examined separately in order to reveal where the main difficulties related to formulating the invariant lie.

The analysis uncovered eight general error types. The frequency of these decreased from the beginning to the end indicating that, in general, students become better at solving problems using the invariant based approach as the course progresses, although the tasks at hand become more difficult.

A further analysis of the invariant related errors revealed nine error types, out of which five were deemed as “severe” and the remaining four as “less severe”. The severe errors were directly related to the invariant, whereas the less severe ones were notational or could be considered careless errors. The findings suggested that while finding the invariant is not trivial, less severe errors still make up a substantial part of the invariant related errors. The main problem can be traced back to a lack in skills to use logical notation, as many students were found to have problems with interpreting, expressing and manipulating quantified expressions. Finally, it was concluded that the invariant based approach is just as suitable for novice students as for students who have studied for a longer period of time as no difference in exam performance was found between the two groups.

Chapter 8

Discussion

In this chapter, the main results from the studies presented above are summarized and discussed. In addition, we discuss the applicability of the chosen research methodology, implications for teaching as well as the quality, limitations and relevance of the research.

8.1 Structured derivations

The key findings from evaluating the use of structured derivations in an educational setting presented in section 7.1 can be summarized as follows:

- High school students are not used to justifying their solutions in mathematics courses.
- The use of structured derivations improves students' justification skills.
- The main perceived benefits of using structured derivations are related to the use of justifications, which leads to increased clarity, understanding and improved error checking possibilities.
- The main perceived drawbacks of using structured derivations are related to time, length and syntax.
- Students choose the level of detail for their justifications mainly based on the difficulty level of the task at hand.

Before moving on to discussing these findings, we feel the need to point out that for illustrative reasons the structured derivations approach has been compared to one type of mathematics teaching (called “traditional”) in this thesis. We acknowledge that there are other approaches, apart from structured derivations, which can give at least some of the same benefits. For instance, the importance of communication can be emphasized in teaching without the use of formalisms and a standard format. Nevertheless, structured derivations do bring other unique advantages, such as a way to keep an entire solution together in one single chain, potential for automatic checking and new exercise types.

Importance of justifications Communication, argumentation and justification skills are recognized as central to the learning of mathematics at all levels [146]. In addition to communicating mathematical ideas orally, students also need to learn how to document their reasoning in writing; research [4] has found that written documentation can be even more efficient for developing understanding compared to a mere oral argumentation. Unfortunately, as noted in section 2.4, students are not used to justifying their solutions, since the traditional way of presenting mathematical solutions does not require them to do so.

Using structured derivations, however, justifications become a natural part of solving problems, and our studies have shown that students learn to justify their solutions quite quickly during one single course. The justifications force students to start writing mathematical text and as they get used to writing justifications continuously, it also seems reasonable to assume that their mathematical language and corresponding communication skills will improve. In most cases, the benefits reported on in our studies (increased clarity, understanding and facilitated error checking) were directly or indirectly linked to the justifications.

The structured format and the documentation available in the justifications make it easier to spot mistakes. Compare this to the traditional format, where one often has to start checking everything from the beginning in order to find a mistake. Research has shown that students, in general, are not keen on checking their solutions in mathematics, although their problem-solving performance could be improved by doing so [143]. Checking a solution in the traditional format is, however, time consuming. For instance, while analyzing the difficulties students encounter when solving two mathematical tasks, Lithner [122] found that it took a first-year undergraduate student seven minutes to find an error in his own solution.¹ Without justifications, one has to once again reconstruct the thought processes that took place when solving the problem in the first place in order to be able to follow the chain of reasoning. Using structured derivations, on the other hand, examples and solutions become self-explanatory.

As a teacher, one should, however, not expect to get long and advanced justifications for simple and familiar steps. Our findings in Paper IV show that students choose the level of detail for their justifications mainly based on the difficulty level of the task at hand: in tasks that are familiar, students tend to opt for broad and vague justifications, whereas justifications which say more come into play when the topics covered are new and/or the assignments become more difficult. Justifications written using students' own words are of particular importance to the teacher for understanding a solution and students' thinking; this is not necessarily the case for vague and broad justifications.

The issue of time Students found that the time required for writing proofs and solutions was increased using structured derivations. This is understandable, as solving a problem does no longer only involve writing down the actual mathematical formulas

¹The task involved constructing a profit function based on existing functions for calculating production cost and sale price, and then finding its maxima in a given interval.

and expressions, but also the justifications. Clearly, the increased writing will lead to an increase in time spent on formulating the solution to a given task. We, however, believe that spending more time on a mathematical problem has several benefits, for instance by helping avoid careless errors. This was also recognized by the students in our studies.

In addition, the very nature of mathematical problems requires a certain amount of time; mathematics is not something to rush through if one really wants to understand. This is, however, not necessarily obvious to students. Research has shown that there is a wide spread belief among students that all problems can be solved quickly [78]. Schoenfeld [184] found that high school students think 12 minutes is a reasonable amount of time to spend on a problem before concluding that it is impossible. Students thus appear inclined to give up if they do not succeed in solving a problem in a short amount of time.

A related issue is wait time in the mathematics classroom, i.e. how long the teacher waits after asking a question before picking a student to respond or before answering it him/herself. Studies have shown that teachers wait less than three seconds for simple questions, whereas the time is extended for more challenging problems [96, 177, 202]. The short wait time for simple questions may give students the impression that these should be answered rapidly and briefly. On such premises, it is understandable that students are not tempted to spend much time on, for instance, justifying simple steps.

The reluctance to spend time on a certain task can also be explained to some degree by looking at the generation that today's students belong to [150]. High school and first year university students have grown up with the Internet and are thus used to immediate feedback, expecting quick responses and no waiting time. These students may even value the speed at which tasks are completed more than the level of accuracy of the result.

Although students considered the time consumption a drawback when solving problems themselves, they still appreciated the teacher taking more time when going through examples on the blackboard. Skemp [192] highlights the importance of the teacher going through examples slowly. He notes that while mathematicians are used to handling condensed and concentrated ideas, students are not, and consequently they have a hard time following.

Increase in length Similarly to the time aspect, the increased length of solutions is a natural consequence of using structured derivations; the derivations simply contain more information, making them longer but also clearer.

Nevertheless, the length issue is important to address, and there are ways in which the length can be decreased while still following the format. The perhaps most obvious way to decrease the amount of writing needed is to take larger steps in a derivation. Another possibility is to use abbreviations in the justifications. These "short cuts" should, however, never be used to such an extent that the readability of the derivation suffers. Also, when either introducing a new topic or the structured derivations format, we find it important that teachers and students write detailed derivations and justifications in order to become used to the justifications and learning all the aspects involved when solving a certain type of problem.

Structure and syntax As discussed in section 2.4, mathematics is usually taught and done in a flexible manner. As a result, solutions, proofs and derivations are written in a way that can be considered a matter of individual taste. For a mathematician this is an important aspect of the “inherent beauty” of mathematics, but for a student such freedom can be problematic as he or she needs to spend time on figuring out how to write down, for instance, a proof.

Structured derivations provides a clear syntax, which goes hand in hand with the structure. The use of a given syntax was appreciated by some students for the increase in clarity, whereas others considered it a source of confusion. In general, the university students were more positive towards the syntax than those at high school. This can be explained by the fact that the university students were CS majors and consequently were used to following a given syntax, for instance, when programming.

8.2 Python

The key findings from evaluating the use of Python in an educational setting presented above (section 7.2) can be summarized as the following:

- Students find Python easy to learn.
- Students who learn Python make fewer syntax errors and logic errors than those learning Java.
- Using Python, topics perceived as difficult, such as exception handling, can be introduced early on. This results in a decrease in perceived difficulty level.
- Learning the idea of programming using a “simple” language, such as Python, has potential for facilitating learning more complex and advanced languages later.

Many results put forward in related research support our findings.² For instance, Enbody et al. [64] found that a Python based introductory course prepares students for a follow up course in C++ as well as an introductory course in C++ does, and concluded that Python could be used as an introductory language without the risk for problems when transitioning to another language. This is well in line with our interview findings in Paper VI. Radenski [165] and Sanders and Langford [182] found that students appreciate Python and think it is a good language for introducing programming. This result is similar to our findings presented in Paper V.

In addition, our results (Paper VI) suggest that students who learn to program using a language with a simple syntax make fewer errors than those learning a more complex language. A verbose syntax might result in novices finding it necessary to focus on getting the syntax correct to such an extent that the algorithm becomes a secondary

²Most of the related work is, however, based on studies conducted in a university context, whereas our Python research has taken place in a high school classroom.

concern only. In that case, it seems reasonable to assume that a simple syntax leads to fewer logic errors, which is a hypothesis supported by our findings.

Although not a main result, Paper V also touches on the use of Python from a teacher’s perspective and concludes that the switch from Java to Python brought many benefits. A compact syntax leaves more time for going through examples and for student coding in class, as there is no need to spend time on “extra” writing, merely required as part of the syntax. The indentation forces students to write well structured programs, which makes the code easier to check. In addition, the teacher can illustrate new concepts separately in the interactive mode, instead of having to write a complete program where the new concept may drown in surrounding code. Finally, the built-in constructs and modules make it possible to introduce interesting programs early on.

Based on the results from Paper V, where exception handling was considered difficult by the students, changes were made to the syllabus so that this topic was moved to the very beginning of the course. This change was possible as the syntax of exception handling is straight forward in Python (see Example 3 in section 4.1). As a result, the topic was no longer considered difficult by students (Paper VII). The order in which topics are introduced thus seems to have an impact on the perceived difficulty level. This has also been suggested by Petre et al. [158], who found indications of topics being introduced early in a course to be perceived as “easy” by students, whereas later topics usually are considered more difficult. This can naturally be considered an expected result, since the earlier a topic is introduced, the more the time students have to work on it. When using a language with a simple syntax it is possible to take advantage of this by moving more difficult topics to the beginning of the course; this is, however, not necessarily doable if using a language with a complex syntax.

8.3 Invariant Based Programming

The main findings from evaluating invariant based programming in education presented in section 7.3 can be summarized as follows:

- Invariant based programming can be used to introduce a more formal approach to programming already at novice level.
- Students find the approach interesting and practical, and have no problem learning the syntax.
- Students have difficulty using logical notation and finding strong enough invariants.
- Students find manual proofs tedious, but without proper checks “careless errors” become a problem.

Introducing a formal approach to programming as an introductory course was not completely uncontroversial. As discussed in chapter 2, this is usually avoided due to the view that students do not possess the skills and knowledge needed to deal with the concepts

involved. However, as was shown in the example in section 5.2, invariant based programming does not require any advanced mathematics or logic. Based on our findings, students find no difficulties with the syntax of the approach; on the contrary, they find the method easy to learn. The results also indicate that the approach can be adopted among novices just as well as among students who have studied for a longer time. This suggests that students do not necessarily become more mathematically mature during their CS studies, which is well in line with the claims made by Gries [83] suggesting that traditional mathematics courses have only little or no impact on students' reasoning skills and attitudes.

In invariant based programming, students are able to check if their programs are correct in a local manner, for instance, by considering one transition at a time. The empirical results presented in Paper VIII indicated that students appreciate the approach for being a practical method for verifying programs and actually “seeing” how a program works. The traditional approaches to program correctness building on the “Hoare-Dijkstra” methodology for writing code with built-in proofs rely on static reasoning, where program variables are treated as entities that remain unchanged at any point of execution. At that point their values are frozen and can therefore be treated as pure mathematical entities. Operational reasoning, on the other hand, treats program variables as objects, whose values vary during execution. The latter may seem more natural to a person who is used to practical programming [210]. The diagrammatic notation used in invariant based programming combines operational (control flow is visible) and static (invariants are represented as mathematical entities) reasoning, and may consequently be easier to grasp than an approach based on static reasoning only.

Our studies (Paper VIII-Paper IX) have also pointed out some problematic aspects of using the approach. In the following, we will briefly discuss the two main ones: difficulties related to 1) the construction of manual proofs and 2) the invariant.

Constructing manual proofs Our studies show that most students consider the manual proving process tedious and time demanding, which has also been pointed out in earlier research [67]. This is an expected, and quite understandable, result as even trivial proofs need to be constructed, requiring time and effort. Hence, students do not necessarily understand the point of creating such proofs, and would not do so on their own initiative expecting it to improve their code.

This also brings up the issue of scale, which has been considered a problem with formal methods in general, and one of the reasons why they are not widely used. When arguing that formal methods do not scale up to extensive projects, one should, however, remember that large programs are made up of smaller parts; these “subparts” (modules) can be formally verified even using methods with limited scalability. Still, this is an issue making people reluctant to use the techniques. The introduction of useful tools has been seen as one solution [105, 214]: “[g]iven the right computer-based tools, the use of formal methods could become widespread and transform software engineering.” [105, p. 93] This statement is included in the verification grand challenge, in which the CS community has committed itself to “making verified software a reality within the next

15 to 20 years” (p. 93). In invariant based programming, the *SOCOS* tool, which was described in section 5.2, can be used to, for instance, automatically prove or simplify trivial verification conditions. A beta version of *SOCOS* was, in fact, used in the course described in Paper VIII and IX; based on the initial teaching experiences and student feedback, the tool is currently being further developed to better suit the needs and skill levels of novice CS students.

Nevertheless, we argue that scalability is not a relevant issue when considering introductory education. For instance, the goal of the course on invariant based programming is to let students become familiar with correctness concepts, learn the mathematical basis of programs and get some feeling for how programs can be verified. Clearly, scale does not pose a problem in such a context.³

Despite the obvious benefits of a tool, we also believe that hands-on construction is essential and constitutes the way in which the approach should be introduced. First of all, learning to build invariant based programs and the corresponding proofs manually is important in order for students to become familiar with the approach. The hands-on experience also makes explicit the link between mathematics and programming as each transition becomes a separate theorem that needs to be proven. Without first going through the process of creating and verifying programs using only pen and paper, students would not get to know the effort that goes into manual proofs and might therefore not see how helpful tool support actually is. The lack of a tool does, however, put a natural limit on the difficulty level of the assignments given to students; programs including several nested loops can easily take a very long time to construct and prove manually. Given that students take other courses simultaneously, the assignments for one single course cannot be allowed to require too much time.

Invariant related difficulties As mentioned in section 5.2, one of the hypothesized challenges when starting to teach invariant based programming was that students would have difficulty in finding the invariant. Invariant related errors were, in fact, found to be most common when analyzing students’ solutions. A detailed analysis showed that inventing the invariant, especially when dealing with nested loops, is not trivial, but that there are other problematic aspects as well that do not directly originate in the invariant property.

According to the results, a substantial part of the invariant difficulties were related to expressing the invariant; many students had problems with interpreting, formulating and manipulating quantified expressions. This was also recognized as one of the main problems by the students themselves, and similar indications can be found in the literature [183, 188]. This finding is comparable to our Python results presented in Paper VI, which suggest that a complex syntax may get in the way of learning programming concepts and algorithms. Similarly, in invariant based programming, a student who does not feel confident using logical notation to formalize expressions cannot focus completely

³The scalability of invariant based programming is currently the topic of another project in our research group. A detailed discussion on this is, however, outside the scope of this thesis.

on deriving the correct invariant as syntactical issues get in the way. Consequently, the correctness of the invariant may suffer.

Difficulties related to logical notation and the use of quantifiers also serve as a partial explanation for why students find proofs difficult. Constructing correct proofs naturally becomes difficult if one does not possess the skills to express and manipulate logical statements. Part of the challenges associated with teaching program correctness thus seems to result from the afore-discussed shortages in mathematics education. This result underlines the need for proper and sufficient training in formalizing statements and, conversely, interpreting expressions written using formal notation.

Another noteworthy type of invariant related errors were those coined as careless ones. Although rigorous and formal checking of the verification conditions can reveal all errors, there is not always time for this type of checks during a course with limited hours available. Our results show that it is easy to, for instance, miss out on bounds in guards and leave out variable declarations when there is no interpreter or compiler available that would check the program. Under pressure, these are “smaller details” that may be easily overlooked. This is especially true in exam situations, where students usually do not have the time to do proper proofs.

Although careless errors may not be considered as serious as, for instance, weak invariants, their occurrence accentuates a need to further stress the importance of going through and proving each transition separately, if not by writing a formal proof, at least informally by checking that all properties hold. Without explicit checks, formal or informal, the invariant approach is no more “safe” than a “trial-and-error” approach without systematic testing.

8.4 Applicability of Developmental Research

The developmental research methodology and the analysis procedures used throughout this thesis have proven to be useful as tools for conducting this specific type of research. All results presented are based on data that have been systematically derived from practice. Through the cyclic process between development and educational practice it has been possible to improve and refine the approaches and the teaching material formatively both during the courses and in between. For instance, as a result of the student feedback on structured derivations (Paper I - Paper III) the syntax has been simplified and the issues of time and length have been discussed. In Python, topics originally considered difficult (for instance, exception handling and functions) have been moved to the beginning of the course, thus giving students more time to practice these concepts. In invariant based programming, a new way to split proofs into smaller chunks helped make the proofs shorter. The length was also further decreased by numbering the conditions in the diagrams; previously, the proofs contained quite a lot of repetition as the assumptions had to be explicitly written out separately for each verification condition.

The cyclic improvements have not only taken place for the separate approaches in isolation. On the contrary, results from studies on one approach have pointed out places

for improvement in how the other approaches are taught. For instance, the studies on invariant based programming brought light on issues that need to be changed; the most critical one being the need for more training in predicate logic in general and using quantifiers in particular. As a result, during the current academic year (2008/2009) more focus was put on quantifiers in both the prior course on logic and the invariant based course itself. Preliminary results and observations from this “updated” version of the course indicate that students do no longer exhibit the same kind of uncertainty when working with quantifiers.

Studying the three approaches simultaneously has thus made it possible to make suitable changes in order for the approaches to work as well as possible both in isolation and together. An important premise for this to be possible in practice is naturally a close collaboration between method developers, teachers and researchers. With such collaboration in place, changes and modifications can be made quickly as a joint effort where the perspectives of all parties can be taken into account.

8.5 Implications for Teaching

In this section we consider the implications that the results presented above may have for teaching. In addition, different ways in which the approaches can be introduced in education are discussed.

Formalism and novices Contrary to commonly held preconceptions, the studies presented in Paper I - IV and Paper VIII - IX have indicated that formalism and exactness can be introduced in introductory mathematics and programming education without being “too advanced”. Previous research [17, 156, 157] and the studies presented in this thesis have shown that structured derivations and elementary logic can be introduced at least as early as at high school level. Similarly, invariant based programming can be taught with propositional and predicate logic as the only pre-requisites.

Our results have nevertheless brought light on an accentuated need for giving students sufficient practice in moving between informal and formal representations. In order to successfully use the invariant based approach, more attention needs to be put on appropriate training in formalization as early as possible. Unfortunately, as we have seen previously in this thesis, the current state of mathematics education at high school level does not support such preparation.

Despite the current lack of training in programming, logic and formal mathematics at lower levels of education, this should not be seen as a reason for postponing the introduction of these topics at university level. Carter [39] points out that “[w]hilst we accept that many students do not arrive complete with the skill sets that we desire, we tend to forget that this does not mean they are incapable of attaining them – they just have not yet been given the opportunity to try. Spending some time teaching them the skills and techniques they require must be better than simply trying to avoid teaching the theory behind CS.” (p. 2)

Mathematics and programming at high school level Despite the high quality of secondary education in Finland, the discussion in chapter 2 indicated some apparent places for improvement.

Considering the mathematical maturity required in many university disciplines, high school students would certainly benefit from getting more training in formal reasoning and proof earlier than what is possible with the current high school curriculum. As a natural consequence of teaching mathematics using structured derivations, students become familiar with formalisms, accustomed to using logical notation and used to communicating mathematical ideas in writing. Consequently, the introduction of structured derivations seems as a worthwhile alternative for introducing more rigor throughout high school mathematics without the need to introduce new courses.

Since early experience with CS and programming contributes to important meta skill development and increases the chance of succeeding in university CS courses (see the discussion in section 2.3), it appears important to reintroduce these topics in the high school curriculum. Our studies on Python were conducted in a high school setting, and the results show no reason why programming should not be introduced at high school level. Programming at an early stage could improve the prior knowledge and skills of future university students and also help students get used to following a given structure and syntax. The latter is beneficial in other areas of study as well; for instance, one could assume that familiarity with following a syntax would aid when learning to present mathematical solutions using structured derivations.

As high school students study several subjects in parallel, one cannot expect them to spend an abundance of their spare time on one single course. It is thus essential that a programming course focuses on the important aspects of programming and avoids wasting time on irrelevant issues, such as going through and memorizing a verbose syntax. Based on our results, this could be achieved using a language with a simple syntax.

Consequences for the teacher The three evaluated approaches are, in essence, only new ways of presenting “old” topics. Nevertheless, as is the case with all changes, instructors should expect and be prepared for some initial reluctance when introducing new methods. This has been especially evident for structured derivations and invariant based programming, which can be seen as rather different from the traditional ways of presenting mathematics and programming. The resistance may be particularly pronounced in mathematics education, as students have already studied the subject without having to follow any fixed format since first grade.

Solving problems using structured derivations and invariant based programming may take more time than when using traditional approaches to mathematics and programming. In structured derivations this is due to the additional writing involved and in invariant based programming the increase in time is related to the need for proving the program.⁴ As argued in chapter 8, the time requirements should however not be seen

⁴The time needed to thoroughly test a program would hardly be any less than the time it takes to prove it, but rigorous testing is not an activity that most students invest time or effort in (unless

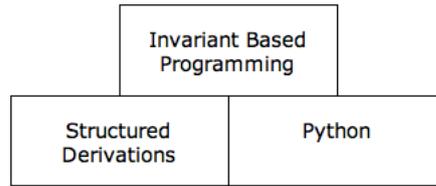


Figure 8.1: Course model where invariant based programming builds on the two lower courses

as a drawback; rather, they should be considered a factor guaranteeing that the student does a thorough and good job when solving a problem. This does have implications for the teacher, who can no longer give tens of assignments as homework or go through a handful of examples in an hour. Rather, one needs to be able to carefully pick out and focus on the most essential assignments.

Alternative course models The three approaches work well together and can be used to build a continuum of courses covering two of the fundamental concepts of CS: programming and mathematics. The approaches can naturally also be introduced separately. Structured derivations can be used as a way for introducing justifications and more formalism in mathematics education, and invariant based programming can be introduced provided that students have background knowledge in some proof format. Whether practical programming experience is a prerequisite for invariant based programming is still unclear. With students having no programming background, there would be no need to overcome an ingrained way of thinking or previously learnt programming language syntax. On the other hand, without any programming background, all programming concepts would be unfamiliar. In such a situation, the invariant based approach could be introduced, for instance, as a course on algorithms starting from the concept of mathematical induction.

Partly based on the results presented in this thesis, the model presented in Figure 8.1 has been adopted in the introductory CS curriculum at Åbo Akademi University. Structured derivations and Python are taught during the first semester of the first study year, followed by a course on invariant based programming at the beginning of the second semester. This model fits international recommendations for introductory CS instruction [99, 104] as it integrates mathematics and programming.

Two of the approaches, structured derivations and Python, have been evaluated at high school level with positive results. Invariant based programming, on the other hand, has so far only been introduced at introductory university level. However, given that high

they are also graded on how well they test their own programs, which is the case when, for instance, using Web-CAT (<http://web-cat.org/>) for automatic assessment of programming exercises). Thus, the time needed for proving seems as an additional overhead that is not required when programming in the traditional sense.

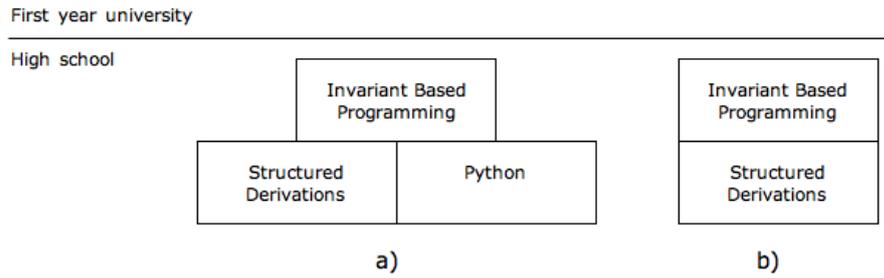


Figure 8.2: Alternative high school course models

school students on their last year are close to first year university students with regard to age and attitude [6], one could argue for the introduction of invariant based programming at high school level as well. This could be arranged, for instance, by offering invariant based programming as an elective course in advanced mathematics, just as structured derivations was introduced in the elective course on logic and number theory in the studies reported on in Paper II-IV. Depending on whether a course on programming is deemed as a necessary precursor, we could arrive at two different high school models. One (alternative a) in Figure 8.2) would require students to have a background in structured derivations as well as in a programming language, whereas the other (alternative b) in Figure 8.2) would only require knowledge of the former.

Regardless of the choice of model, the bigger challenge, as we see it, lies in making sure that students have enough practical skills in using logical notation, both to interpret formal statements and to formalize expressions, before taking the course. If the lack of logic and formalism in high school mathematics prevails, this may become a problem; the single course on logic and number theory is hardly sufficient for giving students the skills needed. After all, the course involves a limited number of hours (approximately 28 lectures a la 45 minutes in class).

We see three solutions to this problem. The first, and maybe the most obvious one, would be to introduce more courses on logic at high school level. Given the discussion in section 2.4 on the current situation of mathematics in Finnish high schools, it seems rather unlikely that this option would be realized. Alternatively, one could consider revealing the logic that is already present, but hidden, in high school mathematics, for instance by starting to use logical connectives and quantifiers instead of informal notations and descriptions. This would clearly require much effort, as all text books and examples would need to be rewritten. Using structured derivations this step would, however, be an obvious one to take. Yet another alternative would be to adapt the invariant based approach to the level of the students so that, for instance, quantifiers would be expressed using natural language and “proofs” would be performed informally only; this would, however, mean that we would “accept” that logic is hidden in mathematics education and therefore we see this at the least desired alternative.

8.6 Evaluating the Results

Evaluation criteria In all research it is important to consider how we can be sure that the findings are worth paying attention to. Lincoln and Guba [118] list four questions, which researchers have found useful for this purpose (p. 290):

1. How can we be certain that the findings of a study are “true” for the subjects and context involved?
2. How can we determine how applicable the results of a study are with other subjects or in other contexts?
3. How can we decide if the results of a study would be repeated if the study was replicated under similar or the same conditions?
4. How can we determine to what extent the results are based on the subjects and the given context and not on the researcher’s biases, motivations, interests or perspectives?

The standards for how the questions are to be answered for quantitative and qualitative research respectively are different, as the two research paradigms “proceed from different assumptions, attend to different phenomena, and are appropriate for different purposes” [77, p. 14]. Quantitative research focuses on discovering existing facts according to a belief that the reality can be isolated and is objective. The goal is to discover results which are generalizable, i.e. applicable universally regardless of factors such as time, place and culture [94]. Qualitative research, on the other hand, builds on researcher’s interpretations, and it is hence impossible to separate the researcher from the issue under study [94]. Focus is therefore not on objectivity in the traditional sense, but rather on the level of honesty and credibility of the researcher [153]. When judging the quality of qualitative research, the goal is not to show that the findings are true, rather that they are trustworthy. Moreover, the qualitative researcher cannot make any claims of a study being replicable [126].

Most of the results presented in this thesis are based on qualitative data in the form of student answers to open questions, programs or mathematical solutions. Where quantitative methods have been used to illustrate results from analyzing these data, the trustworthiness is mainly dependent on the quality of the qualitative analysis. Some quantitative data have also been collected, for instance, when using Likert scales (Paper II, V, VII and VIII) and comparing course grades (Paper V). We will begin by discussing the trustworthiness of the quantitative parts of our work and then move on to a corresponding discussion with regard to the qualitative parts.

Rigor of the quantitative studies For quantitative research, the questions listed above are answered through the criteria *internal validity*, *external validity*, *reliability* and *objectivity* [45, 118, 203].

The *internal validity* of a study refers to the extent to which the results can be attributed to the variables measured and not to other possible causes. It is hence typically only relevant in research that aims at establishing a causal relationship. There are several threats to internal validity, i.e. reasons why the results in a study could be attributed to other factors than the one under study: history, maturation, testing, instrumentation, mortality, and regression. In addition, there are threats related to group selection and social interaction. For an explanation of what these entail, the reader is referred to the Research Methods Knowledge Base [203].

We have identified some threats to internal validity in our studies. In Paper V and VI, the course grades and student errors were compared to those of a group of students from two years earlier. Consequently, the students and assignments were not identical. This can be seen as a selection threat, since the results of the comparison may be argued to be the result of alternative causes, and not of the programming language used. As the Java course had taken place earlier, we were not able to collect any further data (for instance, related to motivation level, programming background and mathematical skills) from the students that could have made it possible to make more conclusive statements about the comparability of the two groups. One set of data that was available was the final grade in mathematics from comprehensive school, but using this for comparing the groups would have been questionable, since the students had come to high school from different schools, hence having had different teachers with their own grading principles. In the respective papers, we argue for why we have reasons to believe that the two groups are, nevertheless, comparable.

Where the findings are based on data collected at the end of the course, students who have dropped out are naturally not heard. This is an example of a mortality threat to internal validity. Although the dropout rate in our courses has been quite low (by dropouts we here refer to students who have handed in at least one assignment before dropping out), not hearing these students can be considered a weakness in our research; these students could have had valuable opinions, for instance with regard to the drawbacks of the methods, which were now left out. However, as most students who dropped out did so early on in the course, they had unlikely had enough time to “get the idea” of the methods and be able to give constructive feedback.

A Likert scale was used in Paper V, VII, VIII for one of the following reasons: 1) to let students rank the difficulty level of the topics of the course (from very easy to very difficult), or 2) to gain insight into students’ attitudes towards a certain issue under study, for instance the appropriateness of Python as the first language. When using Likert scales to reveal attitudes, it is important to remember that the scale does not measure attitudes directly but rather the extent to which a respondent agrees with a number of written statements [113]. They do, however, aid in illustrating how strong the views of the respondents are in relation to each other. In all cases the scales were quite brief, thus minimizing the risk of pattern answering which can arise if students become tired or bored [32]. The statements were carefully worded to avoid ambiguity, so that all respondents would have the same thing in mind when answering. In addition, the statements were made to reflect attitudes both for and against the phenomenon at hand.

External validity refers to the extent to which the results of a study can be generalized beyond the sample to other contexts. Our studies have been conducted in course settings with quite small student groups. Hence, we have not been able to do any actual sampling; instead, the sample has been made up by all students in class in order for us to obtain a reasonable number of subjects. This is a direct result of the size of student groups at Finnish high schools and the Department of Information Technologies at Åbo Akademi University, and hence outside our control. This type of sampling may produce samples that are not representative of the population as the whole [47]. This should, however, not be a problem in our context, as we do not claim that the results are highly generalizable (see “transferability” below).

Reliability can be seen as a “synonym for dependability, consistency and replicability over time, over instruments and over groups of respondents.” [45, p. 146] For research to be reliable, it needs to demonstrate that the results would be similar if it was repeated under similar conditions. In order to determine whether the results are reliable, a study thus needs to be replicated. As was already mentioned in section 6.1, educational studies are not easily replicated, since the context is complex and changing to its nature. Throughout all studies, we have, however, aimed at making the research process transparent to make it possible for the reader to evaluate the validity and reliability of the results.

Objectivity is commonly evaluated based on the notion that the researcher is seen as external to the research process. The criterion refers to the extent to which findings are grounded in the data and not the result of, for instance, research bias. When analyzing quantitative data, our focus has been on using statistical tests and measurements, and not on our own interpretations. Hence, the results should be objective.

Trustworthiness of the qualitative studies Due to the differences between quantitative and qualitative research discussed above, the criteria used to evaluate the quantitative parts of our research are not considered suitable for judging the qualitative ones. Several alternative ways describing how qualitative studies should be evaluated have therefore been proposed. Some researchers [76] have argued that it is “pointless to attempt to predetermine a definitive set of criteria against which all qualitative research should be judged” (p. 515), whereas others believe that the usual criteria used for quantitative research “have value but require redefinition to fit the realities of qualitative research” [199, p. 266]. The idea of the former is that eventually, the evaluation will still be made in the mind of the reader in the light of his or her background and interpretations. According to this view, it is crucial that the researcher describes and justifies both the research process and the ways in which the trustworthiness of the research has been strived for and evaluated.

Among those researchers who believe in the usefulness of explicit criteria are, for instance, Lincoln and Guba [118], who have identified an alternative set of criteria corresponding to those commonly used to judge quantitative research. Their concept of “trustworthiness” includes four criteria parallel to those of quantitative research: *credibility*, *transferability*, *dependability* and *confirmability*.

Credibility (cf. internal validity) refers to how well the interpretation the researcher has created corresponds to the reality [118]. In our work, we have aimed at improving credibility in five ways: 1) use of authentic quotations, 2) cultural competence of the researcher, 3) prolonged engagement, 4) triangulation, and 5) data collection in authentic settings. In the following, we will briefly discuss these in more detail.

The use of *authentic quotations* is a common way of increasing credibility in qualitative research, as it makes it possible for the reader to determine how well the categories fit the data. In all studies we have described the categories in detail, and where possible (the studies based on student feedback) we have illustrated the categories with quotations taken from as many different respondents as possible to avoid overquoting the same persons.

The researcher makes interpretations based on her own background and knowledge. For her to make trustworthy interpretations it is essential that she is *culturally competent* [118], that is, that she is familiar with the study domain and speaks the same language as the respondents. All researchers involved in the studies presented in this work are insiders in the field of mathematics and CS, and hence possess the competence needed to interpret the collected data.

Prolonged engagement at the research site also increases credibility [118]. As all studies in this thesis are based on entire courses, the researcher (or the teacher, with whom the researchers have collaborated) should have had time enough to demonstrate a prolonged period of engagement and observation for learning the context, building trust etc.

Triangulation is important, as the credibility of a given result is increased if supported by, for instance, different data sources or collection modes [118]. Our aim has been to base our findings on several types of data, for instance, from questionnaires, interviews and student constructed material (solutions, proofs, programs). In the studies on structured derivations, we have used data from two educational levels (university and high school).

All data have been *collected in authentic settings*, i.e. in actual classroom situations. In our opinion, this also increases credibility, as the students have not been put into an artificial environment.

The lack of formal tests conducted to verify the credibility of the categories that have emerged in the studies can be argued to be a weakness. In our opinion, the way in which the categories were created and discussed (see section 6.3) should, however, have revealed any potential discrepancies. During the coding process it became clear that the data were rather easy to categorize; only a few unclear cases were encountered, and these were categorized only after two or more researchers had discussed the piece of data and agreed on how it should be interpreted.

Transferability is the qualitative correspondence to the external validity (generalizability) criterion in quantitative research. Lincoln and Guba [118] point out that “if there is to be transferability, the burden of proof lies less with the original investigator than with the person seeking to make an application elsewhere. The original inquirer cannot know the sites to which transferability might be sought, but the applicers can and do.” (p. 298) The original researcher is therefore merely responsible for providing the reader

with sufficient descriptive data, making it possible for him or her to judge whether the studied approach or phenomenon could be applicable in his or her settings. To attend to this criterion, we have provided detailed descriptions of the different studies, for instance with regard to subjects, educational settings, application of methods, data collection and analysis. We do not, however, claim that the results are directly transferable to other contexts; rather, we aim at providing an understanding for how the methods can be used in education and what benefits and difficulties to expect in contexts similar to ours. This can, in turn, provide the reader with insight into what the results could bring to his or her own situation. Furthermore, as the goal of the studies has been to evaluate the use of new approaches in education, the results should be interesting from a method developer's and curricula designer's point of view regardless of the degree of transferability.

In quantitative research, reliability is based on replicability. Qualitative studies, however, are commonly unique and not easily repeated, which makes it inappropriate to talk about the reliability of the results. Instead, Lincoln and Guba [118] introduced the term *dependability*, which also takes into account the stability and consistency of the processes involved in a study over time. One way of improving dependability is to make it possible for others to conduct an "inquiry audit" [118]. In order to do so, the researcher needs to store raw data, coding schemes, notes and other documents pertinent to the analysis. Based on this material, an external auditor can then evaluate the process (i.e. how the results have been arrived at) and determine whether dependability can be established. Although we have not conducted such an audit in this thesis, we have stored all material, emails etc. that have been used during the analysis. Hence, the material to conduct an audit is available.

In quantitative studies, the objectivity criterion takes as a starting point that the researcher is external to the study process. Similarly to the reliability criterion, objectivity can be considered irrelevant for qualitative studies where the researcher makes interpretations and is hence "inside" the research. The focus of *confirmability* is therefore on how well the findings are grounded in the data. In our opinion, many of the measures taken to improve credibility also improve confirmability. For instance, the use of authentic quotations shows the connection between the collected data and the produced interpretation. A high level of transparency with regard to the research and its phases also aids in helping the reader determine whether any biases may have affected the results. In addition, the inquiry audit described above can also be used to establish confirmability; the auditor can evaluate the product (i.e. the results) to determine whether it is based on the data. Finally, reference to previous research findings that confirm the interpretations can improve confirmability. As two of the methods studied in this work have not been used extensively in education before, there are no previous research to refer to. For the work on Python, however, some connections to related literature have been made.

Other limitations One could argue that the studies presented in this thesis are not based on a joint theoretical framework. The initial studies on the three approaches were explorative in nature, whereas the following ones investigated various aspects of the educational use of the approaches. The lack of a single theoretical framework is thus a

natural consequence of the somewhat different foci of the studies. Rather than try to force all research to fit one single theory, we therefore chose to provide a context for the respective studies separately using relevant literature.

Tests of statistical significance can be used to find out the probability with which a difference between, for instance, two groups have occurred by pure chance. In this work such tests have been conducted in several papers, whereas some articles lack this kind of analysis. This can be seen as a limitation in our work. As the articles have already been published, this can, however, no longer be amended.

The empirical studies are interrelated and build upon each other in the sense that the first study inspired the second one, the second study raised questions that were investigated in the third one, and so on. Since several studies use the same data, there are therefore some cases of overlap. As we see it, these overlaps are, however, not redundant, but rather a natural result of using a developmental research approach, where the aim of each study has been to give insight into different aspects of using the approaches in education and point out interesting issues for further investigation.

8.7 Relevance

The overall goal of any research is to address relevant problems with potential for impact. As already pointed out previously, the present work has been valuable for the further development of both the approaches and the ways in which they are used in education.

Considering the current state of mathematics education and the call for a mathematically more rigorous CS education discussed in section 2.4, empirical data as those presented in this thesis are relevant. Although attempts at, for instance, introducing more formalism in the mathematics and CS classroom have been made before, only few have provided empirical results. In our opinion, sketching new ideas for teaching without providing any authentic evaluation is not enough. The ideas could naturally be worthwhile, but one cannot know for sure before they have been tried out in practice.

Results as the ones presented in this thesis are also useful in discussions on curricula development. As an example, the research has been important for the development of the new introductory course sequence in CS at Åbo Akademi University. To address the serious gap when comparing the mathematics and CS education given at high school on one hand and the expectations at university level on the other, there is a need for increased collaboration and joint efforts between high school curricula designers and university representatives. In such discussions, empirical results are of great importance to inform decision making.

Chapter 9

Final Words

In this concluding chapter, we will summarize the main contributions and give some suggestions for future work.

9.1 Conclusions

The main contribution of this thesis lies in the presentation and evaluation of new methods for teaching mathematics and programming. The choice of the three approaches was made based on the research conducted at the Department of Information Technologies at Åbo Akademi University. Two of the evaluated approaches have been developed within our research group: 1) structured derivations starting from the book on refinement calculus by Back and von Wright [18] and 2) invariant based programming as a correct by construction approach to programming. In addition, Python has been extensively used in the Gaudi Software Factory [142] when working out software processes with students as programmers. As the three approaches respectively seemed to work well for 1) making the process of “doing mathematics” more explicit and logic-based, 2) constructing correct programs relying only on elementary propositional and predicate logic, and 3) getting students started with programming quickly using a new language, we found it interesting to investigate whether these approaches could be used in our introductory courses.

The results of our empirical studies suggest that the methods are appropriate alternatives for teaching mathematics and programming to novices while also providing several benefits. Structured derivations can be used to introduce preciseness and rigor early on in mathematics, and as a result, students become used to formalisms and better at communicating mathematical ideas. Novices learning to program using Python make less errors than those learning a more complex language, which suggests that using Python, focus can be put on learning essential concepts rather than on sorting out issues originating from a complex syntax. Invariant based programming is found easy to learn by students, and the approach can be used to introduce beginners to the more formal aspects of programming and help them develop a deeper understanding for how, and why, a program works. The invariant based approach does, however, require sufficient

training in applying the underlying logic. Taken together these approaches can be used to create an educational setting that provides a concrete link between the theory and practice of computing. The results are indicative, but not necessarily generalizable to other contexts. Nevertheless, we believe that the results are applicable and important when considering 1) the design of mathematics and programming education, and 2) the development of the evaluated teaching approaches.

The developmental research methodology has proven suitable for the type of research presented in this thesis, as it makes it possible to design and develop teaching methods based on practice-based results from authentic settings. By following this methodology, it has been possible to use the results put forward in Paper I - IX for developing both the approaches and the ways in which they are used in education.

Although the research is based on specific approaches, the results are of general nature. Applying “rules without reason” when solving mathematical problems may work in the short run, but in order to develop mathematical reasoning skills, one needs to be able to justify steps in the solution and communicate mathematical ideas efficiently. The findings in this thesis point out many benefits of students doing so. In addition, hiding exact formalisms behind informal notations is prone to result in ambiguity and confusion. Thus, despite the teaching approach chosen for teaching programming and mathematics, there is a need for investing time in going through how mathematical ideas can be communicated (using explicit justifications) and formalized (using logical notation).

Even though the discussion on teaching practical programming has focused on Python, our intention is not to point out that language as the one and only alternative for introductory programming education; rather, we have presented the benefits of using a language with a simple syntax. Hence, the results could just as well apply to any other language that shares the same features and characteristics as those of Python.

In order to truly understand the underlying concepts and workings of a program, it is, however, not enough to be proficient at constructing code in a given programming language; one also needs to understand and work with concepts such as preconditions, invariants and postconditions, which in turn requires good skills in formalizing statements and interpreting logical expressions.

9.2 Future Research

Currently, our plans for future work focus mainly on the use of structured derivations and invariant based programming in education. Compared to evaluating an existing programming language in education, we feel that these new approaches provide us with several, yet unexplored, areas of interest.

The future work can be divided into three main categories based on their respective focus: 1) evaluation, 2) tool development and 3) educational practice.

The goal of the *evaluation* focused plans is to continue empirically evaluating the approaches in new contexts. An interesting question is if, and in that case how, structured

derivations could be introduced earlier, for instance, at the final year of comprehensive school. As noted earlier, we are interested in investigating whether invariant based programming could be used to introduce algorithms at high school level in order to give students a more theoretical view of CS. How this could be accomplished in practice is, however, yet undecided. In addition, as noted in chapter 7, the studies presented in this thesis have raised interesting questions for further investigation, for instance, with regard to structured derivations and the perceived increase in understanding.

Another interesting research agenda is in the area of *tool development*. We are currently creating tools for structured derivations and invariant based programming, and an additional challenge here is how to build the tools so that they are usable by both teachers and students. The vision for the structured derivations tool is, for instance, not only to serve as an editor for writing derivations on a computer, but also to provide syntax checking and verification features. By checking the correctness of every step in a derivation separately, a tool could conclude whether the entire solution is correct or not. Thus, the tool could analyze any solution written as a structured derivation without the need for template solutions. In cases where the correctness of a certain step could not be automatically decided, the tool would leave a mark for the teacher so that he or she would know to manually check that specific point of the solution. The availability of tools suitable for use in a classroom setting naturally gives rise to many new and interesting research questions. How is the tool used by the teacher? By the students? How does it affect learning? What are the benefits and drawbacks of tool support in the classroom? How can automatic verification be used in an educational context?

From the perspective of *educational practice*, there are a couple of important issues to address that may not be of academic nature but still of great importance in order for the approaches to become more widespread. First, teachers need to be educated in the use of the approaches. Second, text material needs to be created as there currently are no textbooks available for use in curricula employing structured derivations or invariant based programming. Although this may be seen as an obstacle for a more large scale spread, this can also be a blessing in disguise as having waited for empirical results makes it possible to build the material on an evidence-based foundation. Finally, at university level, the curriculum beyond the introductory courses needs to be redesigned so that the efforts to integrate theory and practice are not left at the introductory level.

Bibliography

- [1] Krishna K. Agarwal and Achla Agarwal. Python for CS1 CS2 and Beyond. *Journal of Computing in Small Colleges*, 20(4):262–270, 2005.
- [2] Krishna K. Agarwal, Achla Agarwal, and M. Emre Celebi. Python Puts a Squeeze on Java for CS0 and Beyond. *Journal of Computing Sciences in Colleges*, 23(6):49–57, 2008.
- [3] Gerlese S. Akerlind and A. Chris Trevitt. Enhancing Self-Directed Learning through Educational Technology: When Students Resist the Change. *Innovations in Education and Training International*, 6(2):96–105, 1999.
- [4] Lillie R. Albert. Outside-In - Inside-Out: Seventh-Grade Students' Mathematical Thought Processes. *Educational Studies in Mathematics*, 41(2):109–141, 1995.
- [5] Sylvia Alexander, Martyn Clark, Ken Loose, June Amillo, Mats Daniels, Roger Boyle, Cary Laxer, and Dermot Shinnars-Kennedy. Case Studies in Admissions to and Early Performance in Computer Science Degrees. In *ITiCSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 137–147, New York, NY, USA, 2003. ACM.
- [6] Vicki L. Almstrum. *Limitations in the Understanding of Mathematical Logic by Novice Computer Science Students*. PhD thesis, Department of Computer Science, University of Texas, 1994.
- [7] Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for Teaching Formal Methods. *SIGCSE Bulletin*, 33(2):71–88, 2001.
- [8] Eric M. Anderman and Christopher A. Wolters. Goals, values, and affect: Influences on student motivation. In Patricia A. Alexander and Philip H. Winne, editors, *Handbook of Educational Psychology*, pages 369–389. Routledge, 2 edition, 2006.
- [9] Owen Astrachan. Pictures as Invariants. In *Proceedings of the 22nd SIGCSE symposium*, pages 112–118, New York, NY, USA, 1991. ACM Press.

- [10] Ralph-Johan Back. Program Construction by Situation Analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.
- [11] Ralph-Johan Back. Invariant Based Programs and Their Correctness. In W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.
- [12] Ralph-Johan Back. Invariant Based Programming Revisited. Technical Report 661, Turku Centre for Computer Science, 2005.
- [13] Ralph-Johan Back. Invariant Based Programming. In S. Donatelli and P.S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency*, 2006.
- [14] Ralph-Johan Back. Invariant Based Programming: Basic Approach and Teaching Experiences. *Formal Aspects of Computing*, (3):227–244, 2009.
- [15] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Verifying Invariant Based Programs in the SOCOS Environment. In *BCS-FACS Workshop on Teaching Formal Methods: Practice and Learning Experience*, London, UK, December 2006.
- [16] Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured Calculation Proof. *Formal Aspects of Computing*, 9:469–483, 1997.
- [17] Ralph-Johan Back, Mia Peltomäki, Tapio Salakoski, and Joakim von Wright. Structured Derivations Supporting High-School Mathematics. In A. Laine, J. Lavonen, and V. Meisalo, editors, *Current Research on Mathematics and Science Education*. Department of Applied Sciences of Education, University of Helsinki, 2004.
- [18] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.
- [19] Ralph-Johan Back and Joakim von Wright. A Method for Teaching Rigorous Mathematical Reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, Aug 1999.
- [20] Ralph-Johan Back and Joakim von Wright. Structured Derivations: a Method for Doing High-School Mathematics Carefully. Technical Report 246, Turku Centre for Computer Science, 1999.
- [21] Ralph-Johan Back and Joakim von Wright. *Mathematics with a Little Bit of Logic: Structured Derivations in High-School Mathematics*. Manuscript, 2006.
- [22] Diane Baker and Connie Campbell. Fostering the Development of Mathematical Thinking: Observations from a Proofs Course. *PRIMUS: Problems, Resources and Issues in Mathematics Undergraduate Studies*, 14(4):345–353, 2004.

- [23] Doug Baldwin and Peter B. Henderson. The Importance of Mathematics to the Software Practitioner. *IEEE Software*, pages 22–24, 2002.
- [24] Mordechai Ben-Ari. Computer Science Education in High School. *Computer Science Education*, 14(1):1–2, 2004.
- [25] Tom Bennet. Python Image Processing in a Computer Literacy Course. *Journal of Computing in Small Colleges*, 23(4):20–27, 2008.
- [26] Anders Berglund. *Learning Computer Systems in a Distributed Project Course: The What, Why, How and Where*. Doctoral dissertation, Uppsala University, 2005.
- [27] Robert Biddle and Ewan Tempero. Java Pitfalls for Beginners. *SIGCSE Bulletin*, 30(2):48–52, 1998.
- [28] John. B Biggs. Student approaches to learning and studying. Research Monograph. ISBN-0-85563-416-2, Australian Council for Educational Research, Hawthorn, 1987.
- [29] John Burville Biggs and Kevin Francis Collis. *Evaluating the Quality of Learning: the SOLO Taxonomy*. New York: Academic Press, 1982.
- [30] Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive Factors in Programming with Diagrams. *Artificial Intelligence Review*, (15):95–114, 2001.
- [31] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Threshold concepts in computer science: do they exist and are they useful? In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 504–508, New York, NY, USA, 2007. ACM.
- [32] Ian Brace. *Questionnaire Design: How to Plan, Structure, and Write Survey Material for Effective Market Research*. Kogan Page Publishers, 2004.
- [33] John M. Bridgeland, Jr John J. DiIulio, and Karen Burke Morison. The silent epidemic. perspectives of high school dropouts. Civic Enterprises report, March 2006. Available at <http://www.civicerprises.net/pdfs/thesilentepidemic3-06.pdf>. Accessed on September 11, 2009.
- [34] Kim B. Bruce. Controversy on how to teach cs 1: a discussion on the sigcse-members mailing list. *SIGCSE Bulletin*, 37(2):111–117, 2005.
- [35] Hugh Burkhardt and Alan H. Schoenfeld. Improving Educational Research: Toward a More Useful, More Influential, and Better-Funded Enterprise. *Educational Researcher*, 32(9):3–14, December 2003.

- [36] Matthew Butler and Michael Morgan. Learning Challenges Faced by Novice Programming Students Studying High Level and Low Feedback Concepts. In *ICT: Providing choices for learners and learning. Proceedings of ascilite Singapore 2007*, pages 99–107, Singapore, December 2007.
- [37] Pat Byrne and Gerry Lyons. The Effect of Student Attributes on Success in Programming. *SIGCSE Bulletin*, 33(3):49–52, 2001.
- [38] Jinfa Cai, Mary S. Jacobsin, and Suzanne Lane. Assessing Students' Mathematical Communication. *Schools Science and Mathematics*, 96(5):238–246, 1996.
- [39] Janet Carter. Students May Be Capable of Formal Reasoning - But They Have Yet to Try! Online.
- [40] Vern Ceder and Nathan Yergler. Teaching Programming with Python and PyGame. In *PyCon 2003*, March 2003.
- [41] Mike Clancy, John Stasko, Mark Guzdial, Sally Fincher, and Nell Dale. Models and Areas for CS Education Research. *Computer Science Education*, 11(4):323–341, December 2001.
- [42] Tony Clear. Valuing Computer Science Education Research? In *Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research*, pages 8–18, New York, NY, USA, 2006. ACM.
- [43] Douglas H. Clements. Curriculum Research: Toward a Framework for "Research-Based Curricula". *Journal for Research in Mathematics Education*, 38(1):35–70, 2007.
- [44] Richard Close, Danny Kopec, and Jim Aman. CS1: Perspectives on Programming Languages and the Breadth-First Approach. In *Journal of Computing in Small Colleges*, pages 228–234, USA, 2000.
- [45] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research Methods in Education*. Routledge: New York, 6th edition, 2007.
- [46] Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a Discipline. *Communications of the ACM*, 32(1):9–23, 1989.
- [47] Paul Connolly. *Quantitative Data Analysis in Education: A Critical Introduction Using SPSS*. Taylor & Francis, 2007.
- [48] Cynthia L. Corritore and Susan Wiedenbeck. What Do Novices Learn During Program Comprehension. *International Journal of Human-Computer Interaction*, 3(2):199–208, 1991.
- [49] Barbara Gross Davis. *Tools for Teaching*. John Wiley and Sons, 2 edition, 2009.

- [50] Michael de Raadt, Richard Watson, and Mark Toleman. Language Trends in Introductory Programming Courses. In *Proceedings of Informing Science and IT Education Conference*, pages 329–337, Cork, Ireland, June 2002.
- [51] Euda E. Dean. Teaching the Proof Process: A Model for Discovery Learning. *College Teaching*, 44:52–55, 1996.
- [52] Edward L. Deci. Effects of externally mediated rewards on intrinsic motivation. *Journal of Personality and Social Psychology*, 18(1):105–115, 1971.
- [53] Adrienne Decker. A tale of two paradigms. *Journal of Computing in Small Colleges*, 19(2):238–246, 2003.
- [54] Richard Denman, David A. Naumann, Walter Potter, and Gary Richter. Derivation of Programs for Freshmen. In *Proceedings of the 25th SIGCSE symposium*, pages 116–120, New York, NY, USA, 1994. ACM Press.
- [55] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [56] Edsger W. Dijkstra. The Notational Conventions I Adopted, and Why. *Formal Aspects of Computing*, 14(2):99–107, 2002.
- [57] Edsger W. Dijkstra and Carel S. Scholten. Predicate Calculus and Program Semantics. *Texts and Monographs in Computer Science*, pages 21–29, 1990.
- [58] Edsger W. Dijkstra. Notes on Structured Programming. Technical report, Technological University Eindhoven, 1969.
- [59] Tommy Dreyfus. Why Johnny Can't Prove. *Educational Studies in Mathematics*, 38:85–109, 1999.
- [60] Anna Eckerdal and Michael Thuné. Novice java programmers' conceptions of "object" and "class", and variation theory. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 89–93, New York, NY, USA, 2005. ACM.
- [61] Albrecht Ehlert and Carsten Schulte. Empirical comparison of objects-first and objects-later. In *ICER '09: Proceedings of the fifth international workshop on Computing education research workshop*, pages 15–26, New York, NY, USA, 2009. ACM.
- [62] Jeffrey Elkner. Using Python in a High School Computer Science Program. In *9th International Python Conference*, March 2001.
- [63] Jeffrey Elkner. Using Python in a High School Computer Science Program - Year 2. In *10th International Python Conference*, February 2002.

- [64] Richard J. Enbody, William F. Punch, and Mark McCullen. Python CS1 as Preparation for C++ CS2. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 116–120, New York, NY, USA, 2009. ACM.
- [65] Noel Entwistle. Learning and studying: Contrasts and influences. In Dee Dickinson, editor, *Creating the Future: Perspectives on Educational Change*. New Horizons for Learning, 1991.
- [66] Noel Entwistle. Contrasting Perspectives on Learning. In F. Marton, D. Hounsell, and N. Entwistle, editors, *The Experience of Learning: Implications for teaching and studying in higher education.*, pages 3–22. Edinburgh: University of Edinburgh, Centre for Teaching, Learning and Assessment, 2005.
- [67] David Evans and Michael Peck. Inculcating Invariants in Introductory Courses. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 673–678, New York, NY, USA, 2006. ACM Press.
- [68] Wim H.J. Feijen. Exercises in Formula Manipulation. *Formal development programs and proofs*, pages 139–158, 1990.
- [69] Michael J. Feuer, Lisa Towne, and Richard J. Shavelson. Scientific Culture and Educational Research. *Educational Researcher*, 31(8):4–14, November 2002.
- [70] Sally Fincher and Marian Petre, editors. *Computer Science Education Research*. Taylor & Francis, 2004.
- [71] Finnish National Board of Education. National Core Curriculum for Upper Secondary Schools 2003, 2003.
- [72] Ann E. Fleury. Parameter Passing: the Rules the Students Construct. In *SIGCSE '91: Proceedings of the 22nd SIGCSE technical symposium on CS education*, pages 283–286, New York, NY, USA, 1991. ACM Press.
- [73] Robert Floyd. Assigning Meanings to Programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
- [74] Judith Gal-Ezer, Catriel Beeri, David Harel, and Amiram Yehudai. A High School Program in Computer Science. *Computer*, 28(10):73–80, 1995.
- [75] Tony Gardiner. Learning to Prove: Using Structured Templates for Multi-Step Calculations as an Introduction to Local Deduction. *ZDM: The International Journal on Mathematics Education*, 36(2):67–76, 2004.
- [76] Dean Garratt and Phil Hodkinson. Can there be criteria for selecting research criteria? - a hermeneutical analysis of an inescapable dilemma. *Qualitative Inquiry*, 4(4):515–539, 1998.

- [77] David Geelan. But how good is it? In *Weaving Narrative Nets to Capture Classrooms*, volume 21, pages 14–27. Springer Netherlands, 2004.
- [78] Barbara Glass and Carolyn A. Maher. Students’ Problem Solving and Justifications. In *Proceedings of the 28th Conference of the International Group for the Psychology of Mathematics Educational*, volume 2, pages 463–470, 2004.
- [79] Michael H. Goldwasser and David Letscher. Teaching an Object-Oriented CS1 - with Python. In *ITiCSE ’08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 42–46, New York, NY, USA, 2008. ACM.
- [80] Tina Götschi, Ian Sanders, and Vashti Galpin. Mental models of recursion. *SIGCSE Bulletin*, 35(1):346–350, 2003.
- [81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [82] David Gries. Teaching Calculation and Discrimination: a More Effective Curriculum. *Communications of the ACM*, 34(3):44–55, 1991.
- [83] David Gries. Equational Logic as a Tool. In *AMAST ’95: Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology*, pages 1–17, London, UK, 1995. Springer-Verlag.
- [84] David Gries. Improving the curriculum through the teaching of calculation and discrimination. In C. Neville Dean and Michael G. Hinchey, editors, *Teaching and Learning Formal Methods*, pages 181–196. Academic Press, London, 1996.
- [85] David Gries and Fred B. Schneider. Teaching Math More Effectively, Through Computational Proofs. *The American Mathematical Monthly*, 102(8):691–697, Oct 1995.
- [86] Mark Guzdial. A Media Computation Course for Non-Majors. In *ITiCSE ’03: Proceedings of the 8th annual ITiCSE conference*, pages 104–108, New York, NY, USA, 2003. ACM Press.
- [87] Bruria Haberman and Yifat Ben-David Kolikant. Activating ”black boxes” instead of opening ”zipper” - a method of teaching novices basic cs concepts. In *ITiCSE ’01: Proceedings of the 6th annual ITiCSE conference*, pages 41–44, New York, NY, USA, 2001. ACM Press.
- [88] Said Hadjerrouit. Java as First Programming Language: a Critical Evaluation. *SIGCSE Bulletin*, 30(2):43–47, 1998.
- [89] Dianne Hagan and Selby Markham. Does It Help to Have Some Programming Experience before Beginning a Computing Degree Program? In *ITiCSE ’00: Proceedings of the 5th annual ITiCSE conference*, pages 25–28. ACM Press, 2000.

- [90] Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.
- [91] Anthony Hall. Realising the Benefits of Formal Methods. *Formal Methods and Software Engineering*, 3785:1–4, 2005.
- [92] Anthony Hall, David L. Dill, John Rushby, C. Michael Holloway, Ricky W. Butler, and Pamela Zave. Industrial Practice. *Computer*, 29(4):22–27, 1996.
- [93] Gila Hanna. Challenges to the Importance of Proof. *For the Learning of Mathematics*, 15(3):42–49, November 1995.
- [94] Katsuko Hara. Quantitative and qualitative research approaches in education. *Education*, 1995, 1995.
- [95] John Harrison. Formal Methods in Education and Industry. Presentation at the FM Outreach Meeting. Online, June 2008. Available at <http://www.cl.cam.ac.uk/~jrh13/slides/sri-10jun08/slides.pdf>. Accessed on September 20, 2009.
- [96] Aiso Heinz and Markus Erhard. How Much Time Do Students Have to Think about Teachers’ Questions? An Investigation of the Quick Succession of Teacher Questions and Student Responses in the German Classroom. *ZDM: The International Journal on Mathematics Education*, 38:388–398, 2006.
- [97] Kirsti Hemmi. *Approaching Proof in a Community of Mathematical Practice*. PhD thesis, Department of Mathematics, Stockholm University, 2006.
- [98] Peter B. Henderson. Mathematical Reasoning in Software Engineering Education. *Communications of the ACM*, 46(9):45–50, 2003.
- [99] Peter B. Henderson, Doug Baldwin, Venu Dasigi, Marcel Dupras, Jane Fritz, David Ginat, Don Goelman, John Hamer, Lew Hitchner, Will Lloyd, Bill Marion, Jr., Charles Riedesel, and Henry Walker. Striving for Mathematical Thinking. In *ITiCSE-WGR ’01: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 114–124, New York, NY, USA, 2001. ACM.
- [100] Reuben Hersch. Proving Is Convincing and Explaining. *Educational Studies in Mathematics*, 24:389–399, 1993.
- [101] Charles Anthony R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [102] Chenglie Hu. Rethinking of Teaching Objects-First. *Education and Information Technologies*, 9(3):209–218, 2004.
- [103] Gunn Imsen. *Elevens värld. Introduktion till pedagogisk psykologi*. Studentlitteratur, 2006.

- [104] Joint Task Force on Computing Curricula. Computing Curricula 2001: Computer Science, December 2001.
- [105] Cliff Jones, Peter O’Hearn, and Jim Woodcock. Verified software: a grand challenge. *Computer*, 39(4):93–95, April 2006.
- [106] Cliff B. Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.
- [107] Kalle Juuti and Jari Lavonen. Design-Based Research in Science Education: One Step Towards Methodology. *NORDINA*, 4:54–68, 2006.
- [108] Tanja Kavander and Tapio Salakoski. Where Have All the Flowers Gone? Computer Science Education in General Upper Secondary Schools. In *Proceedings of the Fourth Finnish / Baltic Sea Conference on Computer Science Education*, 2004.
- [109] Joseph T. Khalife. Threshold for the Introduction of Programming: Providing Learners with a Simple Computer Model. In P. Romero, J. Good, E. Acosta Chaparro, and S. Bryant, editors, *Proceedings of the 18th PPIG*, pages 244–254, 2006.
- [110] Christine Knipping. A Method for Revealing Structures of Argumentations in Classroom Proving Processes. *ZDM: The International Journal on Mathematics Education*, 40:427–441, 2008.
- [111] Yifat Ben-David Kolikant. Students’ Alternative Standards for Correctness. In *ICER ’05: Proceedings of the 2005 international workshop on Computing education research*, pages 37–43, New York, NY, USA, 2005. ACM Press.
- [112] David R. Krathwohl. A Revision of Bloom’s Taxonomy: An Overview. *Theory into Practice*, 41(4):212–218, 2002.
- [113] Ranjit Kumar. *Research Methodology: A Step-by-Step Guide for Beginners*. SAGE, 2 edition, 2005.
- [114] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A Study of the Difficulties of Novice Programmers. In *ITiCSE ’05: Proceedings of the 10th annual ITiCSE conference*, pages 14–18, New York, NY, USA, 2005. ACM Press.
- [115] Matti Lehtinen. Pitkän matematiikan opetussuunnitelmat kriittisessä tarkastelussa. *Solmu*, 2:1–2, 2008.
- [116] Uri Leron. Structuring Mathematical Proofs. *The American Mathematical Monthly*, 90(3):174–185, March 1983.
- [117] Timothy C. Lethbridge. Priorities for the Education and Training of Software Engineers. *Journal of Systems and Software*, 53(1):53–71, 2000.

- [118] Yvonna S. Lincoln and Egon G. Guba. *Naturalistic Inquiry*. SAGE, 1985.
- [119] Raymond Lister. The Randolph Thesis: CSEd Research at the Crossroads. *SIGCSE Bulletin*, 39(4):16–18, 2007.
- [120] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 119–150, New York, NY, USA, 2004. ACM.
- [121] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bulletin*, 38(3):118–122, 2006.
- [122] Johan Lithner. Mathematical Reasoning and Familiar Procedures. *International Journal of Mathematics Education in Science and Technology*, 31(1):83–95, 2000.
- [123] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bulletin*, 39(1):499–503, 2007.
- [124] Sandra Madison and James Gifford. Parameter Passing. The Conceptions Novices Construct. Paper presented at the Annual Meeting of the American Educational Research Association, March 1997.
- [125] Martin L. Maehr and Heather A. Meyer. Understanding motivation and schooling: Where we’ve been, where we are, and where we need to go. *Educational Psychology Review*, 9(4):371–409, December 1997.
- [126] Catherine Marshall and Gretchen B. Rossman. *Designing Qualitative Research*. SAGE, 4 edition, 2006.
- [127] Olli Martio. Mitä vikaa on matematiikan opetuksessa? *Arkimedes*, 2:14–17, 1998.
- [128] Olli Martio. PISA-tutkimus, matematiikan oppisisällöt ja opettajat. *Solmu*, 1, 2005.
- [129] Olli Martio. Mathematics Curriculum Development in Finland - Unexpected Effects. Online, August 2008. Available at <http://www.jem-thematic.net/en/node/1210>. Accessed on April 2, 2009.
- [130] Ference Marton and Roger Säljö. Approaches to learning. In F. Marton, D. Hounsell, and N. Entwistle, editors, *The Experience of Learning: Implications for teaching and studying in higher education.*, pages 39–58. Edinburgh: University of Edinburgh, Centre for Teaching, Learning and Assessment, 2005.

- [131] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *SIGCSE Bulletin*, 33(4):125–180, 2001.
- [132] Richard A. McCray, Robert L. DeHaan, and Julie Anne Schuck. Improving Undergraduate Instruction in Science, Technology, Engineering, and Mathematics: Report of a Workshop. Technical report, The National Academies Press, Washington, D.C, 2003.
- [133] Andrew McGettrick, Roger Boyle, Roland Ibbett, John Loyd, Gillian Lovegrove, and Keith Mander. Grand Challenges in Computing Education. Technical report, BCS - The British Computer Society, 2004.
- [134] Linda McIver and Damian Conway. Seven Deadly Sins of Introductory Programming Language Design. In *Proceedings of the 1996 International Conference on Software Engineering: EDucation and Practice (SE:EP '96)*, pages 309–316, Dunedin, New Zealand, Jan 1996.
- [135] Henry McLoughlin and Kevin Hely. Teaching Formal Programming to First Year Computer Science Students. *SIGCSE Bulletin*, 28(1):155–159, 1996.
- [136] Kirby McMaster, Nicole Anderson, and Brian Rague. Discrete Math with Programming: Better Together. In *Proceedings of the 38th SIGCSE symposium*, pages 100–104. ACM Press, 2007.
- [137] Susan M. Merritt, Charles J. Bruen, J. Philip East, Darlene Grantham, Charles Rice, Viera K. Proulx, Gerry Segal, and Carol E. Wolf. ACM Model High School Computer Science Curriculum. *Communications of the ACM*, 36(5):87–90, 1993.
- [138] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, 1983.
- [139] George Milbrandt. Using Problem Solving to Teach a Programming Language in Computer Studies. *Journal of Computer Science Education*, 8(2):14–19, 1993.
- [140] Bradley Miller and David Ranum. Freedom to Succeed: a Three Course Introductory Sequence using Python and Java. *Journal of Computing in Small Colleges*, 22(1):106–116, 2006.
- [141] John A. Miller. *Promoting Computer Literacy Through Programming Python*. PhD thesis, University of Michigan, 2004.
- [142] Luka Milovanov. *Agile Software Development in an Academic Environment*. PhD thesis, Åbo Akademi University, 2006. TUCS Dissertations No 81.

- [143] Tracey Muir and Kim Beswick. Where Did I Go Wrong? Students' Success at Various Stages of the Problem-Solving Process. In *MERGA 28. Building Connections: Theory, Research and Practice. Vol. 1 and 2.*, pages 561–568, 2005.
- [144] Peter Naur. Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4):310–316, July 1966.
- [145] Liisa Näveri. Lasketun ymmärtäminen. *Dimensio*, 3:49–52, 2005.
- [146] NCTM. Standards for School Mathematics. Online. Available at <http://standards.nctm.org>. Accessed on June 12, 2008.
- [147] Rance D. Necaie. Transitioning from Java to Python in CS2. *Journal of Computing in Small Colleges*, 24(2):92–97, 2008.
- [148] Uolevi Nikula, Jorma Sajaniemi, Matti Tedre, and Stuart Wray. Python and Roles of Variables in Introductory Programming: Experiences from Three Educational Institutions. *Journal of Information Technology Education*, 6:199–214, 2007.
- [149] Anthony F. Norcio. Indentation, documentation and programmer comprehension. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 118–120, New York, NY, USA, 1982. ACM.
- [150] Diana G. Oblinger and James L. Oblinger. Is It Age or IT: First Steps Toward Understanding the Net Generation. In Diana G. Oblinger and James L. Oblinger, editors, *Educating the Net Generation*. EDUCAUSE, 2005.
- [151] Joseph D. Oldham. What Happens after Python in CS1? *Journal of Computing in Small Colleges*, 20(6):7–13, 2005.
- [152] David B. Palumbo. Programming Language/Problem-Solving Research: a Review of Relevant Issues. *Review of Educational Research*, 60(1):65–89, 1990.
- [153] Michael Quinn Patton. *Qualitative Research and Evaluation Methods*. SAGE, 3 edition, 2002.
- [154] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A Survey of Literature on the Teaching of Introductory Programming. In *ITiCSE-WGR '07: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 204–223, New York, NY, USA, 2007. ACM.
- [155] Peda.net. Opintojen läpäiseminen ja keskeyttäminen korkeakoulutuksessa. Online, March 2009. Available at <http://www.peda.net/veraaja/tuvako/verkkojulkaisu/maaliskuu2009>. Accessed on September 11, 2009.

- [156] Mia Peltomäki and Ralph-Johan Back. An Empirical Evaluation of Structured Derivations in High School Mathematics. In *ICMI 19: 19th ICMI Study Conference on Proof and Proving in Mathematics Education*, 2009.
- [157] Mia Peltomäki and Tapio Salakoski. Strict Logical Notation is Not a Part of the Problem but a Part of the Solution for Teaching High-School Mathematics. In *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education - Koli Calling*, pages 116–120, 2004.
- [158] Marian Petre, Sally Fincher, Josh Tenenberg, et al. My Criterion is: Is it a Boolean?: A card-sort elicitation of students' knowledge of programming constructs. Technical Report 6-03, Computing Laboratory, University of Kent, Canterbury, Kent, UK, June 2003.
- [159] Paul R. Pintrich. Motivation and Classroom Learning. In Gloria E. Miller William M. Reynolds, editor, *Handbook of Psychology; Educational Psychology*, pages 103–122. John Wiley and Sons, 2003.
- [160] Susan Pirie and Thomas Kieren. Creating Constructivist Environments and Constructing Creative Mathematics. *Educational Studies in Mathematics*, 23(5):505–528, 1992.
- [161] Wolfgang Polak. Formal Methods in Practice. *Electronic Notes in Theoretical Computer Science*, 25:62–72, 1999.
- [162] George Polya. *How to Solve It*. Princeton University Press, 1957.
- [163] Roel Popping. Analyzing Open-Ended Questions by Means of Text Analysis Procedures. In *European Survey Research Association*, Warsaw, Poland., July 2009.
- [164] Lutz Prechelt. An Empirical Comparison of Seven Programming Languages. *Computer*, 33(10):23–29, 2000.
- [165] Atanas Radenski. "Python First": a Lab-Based Digital Introduction to Computer Science. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 197–201, New York, NY, USA, 2006. ACM.
- [166] Anthony Ralston. Do We Need ANY Mathematics in Computer Science Curricula? *SIGCSE Bulletin*, 37(2):6–9, 2005.
- [167] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. Self-efficacy and mental models in learning to program. *SIGCSE Bulletin*, 36(3):171–175, 2004.
- [168] Paul Ramsden. *Learning to Teach in Higher Education*. Routledge, 2 edition, 2003.
- [169] Joy N. Reed and Jane E. Sinclair. Motivating Study of Formal methods in the Classroom. In *TFM 2004*, pages 32–46. Springer-Verlag Berlin Heidelberg, 2004.

- [170] John C. Reynolds. Programming with Transition Diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.
- [171] Rita C. Richey and James D. Klein. Developmental Research Methods: Creating Knowledge from Instructional Design and Development Practice. *Journal of Computing in Higher Education*, 16(2):23–38, 2005.
- [172] Eric Roberts. The dream of a common language: the search for simplicity and stability in computer science education. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 115–119, New York, NY, USA, 2004. ACM.
- [173] Eric Roberts, Kim Bruce, Robb Cutler, James Cross, Scott Grissom, Karl Klee, Susan Rodger, Fran Trees, Ian Utting, and Frank Yellin. The ACM Java Task Force. Project Rationale, August 2006.
- [174] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.
- [175] Michael Rost. Generating student motivation. Online. Available at <http://www.pearsonlongman.com/ae/worldview/motivation.pdf>. Accessed on September 11, 2009.
- [176] Janet Rountree and Nathan Rountree. Issues Regarding Threshold Concepts in Computer Science. In Margaret Hamilton and Tony Clear, editors, *Conferences in Research and Practice in Information Technology (CRPIT)*, volume 95, 2009.
- [177] Mary Budd Row. Wait Time: Slowing Down My Be a Way of Speeding Up! *Journal of Teacher Education*, 37:43–50, 1986.
- [178] Geoffrey G Roy. Designing and Explaining Programs with a Literate Pseudocode. *Journal on Educational Resources in Computing*, 6(1):1, 2006.
- [179] Abhik Roychoudhury. Introducing Model Checking to Undergraduates. In *Formal Methods Education Workshop*, pages 9–15, 2006.
- [180] Jorma Sajaniemi and Chenglie Hu. Teaching Programming: Going beyond "Objects First". In *Proceedings of the 18th PPIG*, 2006.
- [181] Renan Samurcay. The Concept of Variable in Programming: Its Meaning and Use in Problem-Solving by Novice Programmers. In James C. Spohrer and Elliot I. Soloway, editors, *Studying the Novice Programmer*, pages 161–178. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
- [182] Ian D. Sanders and Sasha Langford. Students' Perceptions of Python as a First Programming Language at Wits. In *ITiCSE '08: Proceedings of the 13th annual*

- conference on *Innovation and technology in computer science education*, pages 365–365, New York, NY, USA, 2008. ACM.
- [183] Christelle Scharff. From Design by Contract to Static Analysis of Java Programs: A Teaching Approach. In *FM-Ed 2006*, 2006.
- [184] Alan H. Schoenfeld. Exploration of Students’ Mathematical Beliefs and Behavior. *Journal for Research in Mathematics Education*, pages 338–351, 1989.
- [185] Alan H. Schoenfeld. What Do We Know about Mathematics Curricula? *Journal of Mathematical Behavior*, 13(1):55–80, 1994.
- [186] Alan H. Schoenfeld. Method. In Frank K. Lester, editor, *Second Handbook of Research on Mathematics Teaching and Learning*, pages 69–110. New York: MacMillan, 2007.
- [187] Martina Schollmeyer. Computer Programming in High School vs. College. In *SIGCSE ’96: Proceedings of the 27th SIGCSE technical symposium on CS education*, pages 378–382. ACM Press, 1996.
- [188] John Selden and Annie Selden. Unpacking the Logic of Mathematical Statements. *Educational Studies in Mathematics*, 29:123–151, 1995.
- [189] Dale Shaffer. The Use of Logo in an Introductory Computer Science Course. *SIGCSE Bulletin*, 18(4):28–31, 1986.
- [190] Christine Shannon. Another Breadth-First Approach to CS I Using Python. In *SIGCSE ’03: Proceedings of the 34th SIGCSE technical symposium on CS education*, pages 248–251, New York, NY, USA, 2003. ACM Press.
- [191] Richard J. Shavelson and Lisa Towne. *Scientific Research in Education*. National Academies Press, 2002.
- [192] Richard R. Skemp. Relational Understanding and Instrumental Understanding. *Mathematics Teaching*, 77:20–26, 1976.
- [193] D. Sleeman, Ralph T. Putnam, Juliet A. Baxter, and Laiani K. Kuspa. A Summary of Misconceptions of High School Basic Programmers. In Elliot I. Soloway and James C. Spohrer, editors, *Studying the Novice Programmer*, pages 237–257. Hillsdale, NJ: Lawrence Erlbaum., 1989.
- [194] Ann E. Kelley Sobel. Empirical Results of a Software Engineering Curriculum Incorporating Formal Methods. In *Proceedings of the 31st SIGCSE symposium*, pages 157–161, New York, NY, USA, 2000. ACM Press.
- [195] Tuomas Sorvali. Miten matematiikka taipuu opettajankoulutuksen tarpeisiin? In Jorma Enkenberg, Erkki Savolainen, and Pertti Väisänen, editors, *Tutkiva opettajankoulutus - taitava opettaja*, pages 108–117. Savonlinnan opettajakoulutuslaitos 2004, 2004.

- [196] James C. Spohrer and Elliot Soloway. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632, 1986.
- [197] James C. Spohrer and Elliot I. Soloway, editors. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
- [198] Frank Stajano. Python in Education: Raising a Generation of Native Speakers. In *8th International Python Conference*, 2000.
- [199] Anselm L. Strauss and Juliet M. Corbin. *Basics of qualitative research: techniques and procedures for developing grounded theory*. SAGE, 2 edition, 1998.
- [200] Wing C. Tam. Teaching Loop Invariants to Beginners by Examples. In *Proceedings of the 23rd SIGCSE symposium*, pages 92–96, New York, NY, USA, 1992. ACM Press.
- [201] Harriet G. Taylor and Luegina C. Mounfield. An Analysis of Success Factors in College Computer Science: High School Methodology is a Key Element. *Journal of Research in Computing Education*, 24(2):230–239, 1991.
- [202] Kenneth Tobin. The Role of Wait Time in Higher Cognitive Learning. *Review of Educational Research*, 57(1):69–95, 1987.
- [203] William M. K. Trochim. Research Methods Knowledge Base. Available at <http://www.socialresearchmethods.net/kb>. Accessed on August 7, 2009.
- [204] Allen Tucker, Fadi Deek, Jill Jones, Dennis McCowan, Chris Stephenson, and Anita Verno. A Model Curriculum for K-12 Computer Science: Final Report of the ACM K-12 Task Force Curriculum Committee, October 2003.
- [205] Allen B. Tucker, Charles F. Kelemen, and Kim B. Bruce. Our Curriculum Has Become Math-Phobic! In *Proceedings of the 32nd SIGCSE symposium*, pages 243–247, New York, NY, USA, 2001. ACM Press.
- [206] Jan van den Akker. Principles and Methods of Development Research. In Jan van den Akker, Robert Maribe Branch, Kent Gustafson, Nienke Nieveen, and Tjeerd Plomp, editors, *Design Approaches and Tools in Education and Training*. Kluwer Academic Publishers, 1999.
- [207] Maarten H. van Emden. Programming with Verification Conditions. *IEEE Transactions on Software Engineering*, SE-5(2):148–159, 1979.
- [208] Antonetta J.M. van Gasteren. On the Shape of Mathematical Arguments. *Lecture Notes in Computer Science*, pages 90–120, 1990.
- [209] Guido van Rossum. Computer Programming for Everybody. CNRI: Corporation for National Research Initiatives, 1999.

- [210] J. R. Jefferson Wadkins. Rigorous Proofs of Program Correctness without Formal Logic. *SIGCSE Bulletin*, 27(1):307–311, 1995.
- [211] J. Stanley Warford. An Experience Teaching Formal Methods in Discrete Mathematics. *SIGCSE Bulletin*, 27(3):60–64, 1995.
- [212] Peter Warren. Teaching Programming Using Scripting Languages. *Journal of Computing in Small Colleges*, 17(2):205–216, 2001.
- [213] Brenda Cantwell Wilson and Sharon Shrock. Contributing to Success in an Introductory Computer Science Course: a Study of Twelve Factors. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 184–188, New York, NY, USA, 2001. ACM.
- [214] Jeannette M. Wing. Weaving Formal Methods into the Undergraduate curriculum. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 2–7, May 2000.
- [215] Cecile Yehezkel, Mordechai Ben-Ari, and Tommy Dreyfus. Computer architecture and mental models. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 101–105, New York, NY, USA, 2005. ACM.
- [216] John M. Zelle. Python as a First Language. Retrieved November 21, 2003 from <http://mcsp.wartburg.edu/zelle/python/python-first.html>, 2003.

Appendix 1

```
// class imports
import java.util.HashMap;
import java.util.Iterator;
import java.util.Scanner;
import java.net.URI;
import java.awt.Desktop;

public class OpenURL {
    // declare a new class method taking one argument
    static void printmenu(HashMap menu) {
        String key, value;
        // create an iterator object for iterating over the hashmap data
        Iterator itr = menu.keySet().iterator();
        // iterate over the hashmap
        while(itr.hasNext()){
            // get and print the key and the corresponding value
            key = (String) itr.next();
            value = (String) menu.get(key);
            System.out.println(key + " : " + value);
        }
    }

    static void openpage(HashMap menu) {
        boolean choiceOK = false;
        String choice, url;
        // initialize a Scanner object for reading data from the keyboard
        Scanner in = new Scanner(System.in);
        // let the user input choices until a valid one is given
        while (!choiceOK){
            try {
                System.out.print("Choose a site: ");
                choice = in.nextLine();
                // convert choice to uppercase
                choice = choice.toUpperCase();
                // retrieve the value corresponding to the choice key from the hashmap
                url = (String) menu.get(choice);
                // try opening the url
                Desktop.getDesktop().browse(new URI(url));
                // terminate the loop if the browse-method was successfully executed
                choiceOK = true;
            }
        }
    }
}
```

```

        // if choice is not valid, catch a NullPointerException
        catch (NullPointerException e) {
            System.out.println("You did not pick a valid alternative.");
        }
        // if another error occurs, execute the generic catch-block
        catch (Exception e) {
            System.out.println("Something went wrong.");
        }
    }
}

public static void main(String[] args) {
    // create and initialize a hashmap with three key-value pairs
    HashMap options = new HashMap(3);
    options.put("G", "http://www.google.com");
    options.put("Y", "http://www.youtube.com");
    options.put("M", "http://www.myspace.com");
    printmenu(options);
    openpage(options);
}
}

```

As the purpose of this thesis is not to review the Java syntax, we will not make any comments in addition to the ones written inside the program. The main point is to give an example of what the code for the somewhat longer Python program could look like in Java.

Appendix 2

Mathematics, the advanced syllabus

Compulsory courses

1. Functions and equations
2. Polynomial functions
3. Geometry
4. Analytical geometry
5. Vectors
6. Probability and statistics
7. The derivative
8. Radical and logarithmic functions
9. Trigonometric functions and number sequences
10. Integral calculus

Elective courses

1. Number theory and logic
2. Numerical and algebraic methods
3. Advanced differential and integral calculus

Mathematics, basic syllabus

Compulsory courses

1. Expressions and equations
2. Geometry
3. Mathematical models I
4. Mathematical analysis
5. Statistics and probability
6. Mathematical models II

Elective courses

1. Commercial mathematics
2. Mathematical models III

Part II

Original Publications

Paper I

Structured Derivations: a Logic Based Approach to Teaching Mathematics

R-J. Back, L. Mannila, M. Peltomäki & P. Sibelius

Originally published in *FORMED 2008: Formal Methods in Computer Science Education*. Budapest, Hungary, March 2008.

Structured Derivations: a Logic Based Approach to Teaching Mathematics

Ralph-Johan Back and Linda Mannila and Patrick Sibelius^{1,2,3}

Department of IT, Åbo Akademi University, Turku, Finland

Mia Peltomäki⁴

Department of IT, University of Turku, Turku, Finland

Abstract

Being able to reason rigorously and comfortably in mathematics plays an essential role in computer science, particularly when working with formal methods. Unfortunately, the reasoning abilities of first year university students' are commonly rather poor due to lack of training in exact formalism and logic during prior education. In this paper we present *structured derivations*, a logic based approach to teaching mathematics, which promotes preciseness of expression and offers a systematic presentation of mathematical reasoning. The approach has been extensively evaluated at different levels of education with encouraging results, indicating that structured derivations provide many benefits both for students and teachers.

Keywords: Structured derivations, teaching mathematics, mathematics for formal methods

1 Introduction

Being able to reason rigorously and comfortably in mathematics is an essential prerequisite for studies in computer science (CS), especially when working with formal methods. Nevertheless, many CS students unfortunately show little understanding for and interest in mathematics in general and formal notation, logic and proofs in particular. For instance, Gries [10] notes that “students’ reasoning abilities are poor, even after several math courses. Many students still fear math and notation, and the development of proofs remains a mystery to most.” (p. 2) Almstrum [3] found that novice CS students experience more difficulty with the concepts of mathematical logic than with other CS concepts.

¹ Email: backrj@abo.fi

² Email: linda.mannila@abo.fi

³ Email: patrick.sibelius@abo.fi

⁴ Email: mia.peltomaki@utu.fi

One reason for students' low level of skills in formal reasoning and proofs can be traced back to their prior education: exact formalism and proof are perceived as difficult and consequently avoided at e.g. high school level (e.g. [5,12,13,18]). For instance in Finland, high school students are offered the choice of two different mathematics syllabi, including a total of 21 courses (sixteen compulsory, five elective) [15]. Despite the large number of courses, proof and formal reasoning is only mentioned in the learning objectives for one of them, the elective course "Logic and Number Theory" in the advanced syllabus. This is also the only course that introduces logical notation and truth values. How could we expect first year university students to expose high levels of proficiency in topics such as logic, exact formalisms and constructing proofs, when their only prior chance to study these topics is in one (elective) course throughout their entire general education? Proofs should be considered a way of thinking that can be applied to any mathematical topic, instead of being viewed as a distinct topic [11,18].

In addition to the lack of training in formal reasoning and proofs, studies have indicated problems in the way proofs are approached and presented in education. For instance, Dreyfus [9] claims that students often receive mixed messages. As an example he notes that many mathematical textbooks offer intuitive explanations in one solution, use examples to clarify another, and give a rigorous proof for yet another. The differences between these justifications are however not explicated, but leave students with three different views of what *could* constitute a proof. As a result, students do not know what counts as an acceptable mathematical justification.

Moreover, students are likely to engage in activities that feel worth while and relevant for their studies as a whole. However, the prevalent curriculum strategy at CS departments is to divide courses "into areas of 'theory' and 'practice'... [which] causes both faculty and students to view the theory of computing as separate and distinct from the practice of computing." [2, p. 73] In order for mathematics to be considered useful by CS students, it should thus be presented in a way that clearly links it to the computing practice.

In this paper, we present *structured derivations* [4,6,7], a logic based approach to teaching mathematics, which we argue can be used to address all the aforementioned problems. Structured derivations promote preciseness of expression and offer a systematic and straightforward presentation of mathematical reasoning, without restricting the application area. Using structured derivations, logic becomes a tool for doing mathematics, rather than a object of mathematical study.

We begin with a brief description of the structured derivations approach to constructing proofs in Section 2. The approach has been extensively evaluated since 2001 and currently the evaluation involves five institutions at high school and university level. We summarize the high school experience in Section 3 and give a more detailed account of our experience from using the approach in a first year CS course in Section 4. We conclude with a discussion section including ideas for future work.

2 Structured Derivations

Structured derivations [4,6,7] is a further development of Dijkstra's *calculational proof style*, where we have added a mechanism for doing subderivations and for

handling assumptions in proofs. With these extensions, structured derivations can be seen as an alternative notation for Gentzen like proofs in predicate calculus or higher order logic [6]. A structured derivation has the following general syntax:

```

derivation ::= " • "[ task " : "]" [assumptionList] [(⊢ | ⊢-)] [proofSteps]
assumptionList ::= (postulate | lemma)+
postulate ::= "[ " identification "]" formula
lemma ::= "(" " identification ")" formula derivation
proofSteps ::= term (basicStep | subderivation)+
basicStep ::= relation "{ " motivation "}" term
subderivation ::= relation "{ " motivation "}" derivation+ " ... " term
    
```

Terminals are given inside quotes and nonterminals in roman font. The layout of a structured derivation is fixed. The general proof layout is as follows (“task” is a short informal explanation of what we want to do):

derivation:

<ul style="list-style-type: none"> • task: 	assumptionList
⊢	proofSteps

Below to the left, we show the layout for a derivation where all assumptions are postulates and where there are only basic proof steps. The middle box shows the layout of a lemma, where the proof of the lemma (the formula) is a derivation written directly after the formula but is indented one step to the right. On the right we show a proof step with a subderivation. A subderivation justifies the proof step and corresponds to the application of an inference rule in a Gentzen like proof system. One proof step may require one or more subderivations. The subderivations follow immediately after the motivation for the proof step and are indented one step.

simple derivation:

<ul style="list-style-type: none"> • task: 	id] formula
⋮	
	id] formula
⊢	term
	rel {motivation}
	term
⋮	
	rel {motivation}
	term

lemma with proof:

(id) formula	
<ul style="list-style-type: none"> • task: 	assumptionList
⊢	proofSteps

subderivation proof step:

rel	term	
	{motivation}	
	<ul style="list-style-type: none"> • task: 	assumptionList
	⊢	proofSteps
...	term	

This proof format fixes the overall structure and layout of a derivation (hence the name *structured derivations*) but it does not fix the syntax of basic entities such

as task, formula, term, relation, motivation, or identification. Thus, we can use structured derivations for proofs on different domains, and with different levels of rigor and detail (from a completely intuitive argumentation to an axiomatic proof in a logical theory).

We illustrate structured derivations with two mathematical problems. The first one illustrates the use of logical rules in standard mathematical reasoning, while the second illustrates subderivations and the way in which we combine formal and informal reasoning in a structured derivation (the second problem is taken from the Finnish high school matriculation exam 2006).

Our first problem is to solve the equation $(x - 1)(x^2 + 1) = 0$. We have the following solution

-
- Solve the equation $(x - 1)(x^2 + 1) = 0$:
 $(x - 1)(x^2 + 1) = 0$
 - \equiv {zero product rule: $ab = 0 \equiv a = 0 \vee b = 0$ }
 $x - 1 = 0 \vee x^2 + 1 = 0$
 - \equiv {add 1 to both sides in left disjunct and -1 to both sides in right disjunct}
 $x = 1 \vee x^2 = -1$
 - \equiv {a square is always non-negative}
 $x = 1 \vee F$
 - \equiv {disjunction rule}
 $x = 1$
-

The second problem is to determine the values of a for which the function $f(x) = -x^2 + ax + a - 3$ is always negative.

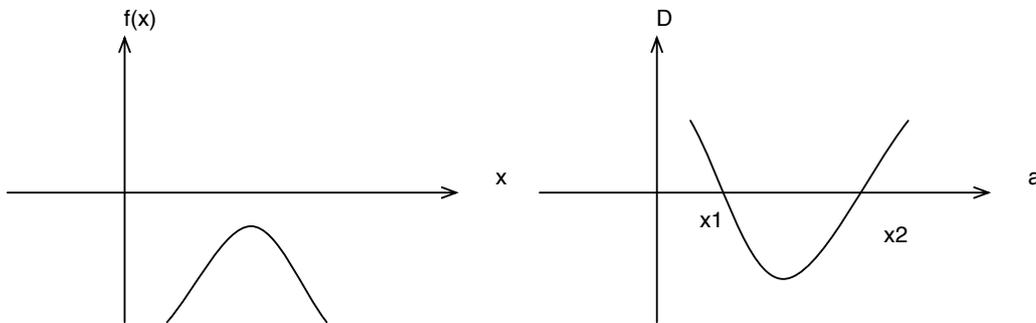


Figure 1. Downward and upward opening parabolas

The following structured derivation shows how we determine the value of a . Figure 1 illustrates the arguments used in the proof.

-
- Determine the values of a for which $-x^2 + ax + a - 3$ is always negative:
 $(\forall x \cdot -x^2 + ax + a - 3 < 0)$
 - \equiv {the function is a parabola that opens downwards (the coefficient for x^2 is negative); such a function is always negative if it does not intersect the x -axis, i.e. has no roots (figure on the left)}
 $(\forall x \cdot -x^2 + ax + a - 3 \neq 0)$

$$\begin{aligned}
&\equiv \{\text{this condition holds iff the discriminant } D \text{ for the function is negative}\} \\
&\quad D < 0 \\
&\equiv \{\text{substitute value of } D\} \\
&\quad \bullet \text{ Determine the discriminant } D: \\
&\quad \quad D \\
&= \{\text{the discriminant for the equation } Ax^2 + Bx + C = 0 \text{ is } B^2 - 4AC\} \\
&\quad \quad a^2 - 4(-1)(a - 3) \\
&= \{\text{simplify}\} \\
&\quad \quad a^2 + 4a - 12 \\
&\dots \quad a^2 + 4a - 12 < 0 \\
&\equiv \{\text{the function } a^2 + 4a - 12 \text{ is a parabola that opens upwards (the coefficient} \\
&\quad \text{for } a^2 \text{ is positive); such a function is negative between its roots (figure on the} \\
&\quad \text{right)}\} \\
&\quad \bullet \text{ Solve the equation } a^2 + 4a - 12: \\
&\quad \quad a^2 + 4a - 12 = 0 \\
&\equiv \{\text{square root formula}\} \\
&\quad \quad a = \frac{-4 \pm \sqrt{4^2 - 4 \cdot 1 \cdot (-12)}}{2 \cdot 1} \\
&\equiv \{\text{simplify the expression}\} \\
&\quad \quad a = 2 \vee a = -6 \\
&\dots \quad -6 < a < 2
\end{aligned}$$

This proves that

$$(\forall x \cdot -x^2 + ax + a - 3 < 0) \equiv -6 < a < 2$$

In other words, the function is always negative if and only if $-6 < a < 2$.

If we are using a computer supported tool with outlining features (like the \TeX macs plug-in mentioned below), we can choose to hide the two subderivations. Omitting the more detailed steps will give us a better view of the overall structure of the proof.

Traditional approaches to teaching and presenting mathematics contain much implicit information [9,14]. Using structured derivations, all steps in the derivation are explicitly motivated and the final product thus contains a documentation of the thinking the student was engaged in while completing the derivation. This facilitates reading and debugging both for students and teachers.

Moreover, as stated in the introduction, traditional approaches to teaching proofs leave students uncertain about what rigor is required for a particular proof in a certain situation [9]. Structured derivations provide a well-defined proof format, which gives students a concrete “model” for what constitutes a proof and which can guide them in how to carry out rigorous proofs in practice. A clear and familiar format functions as a mental support that gives students belief in their own skills to construct the proof. A defined format also lets students focus on the solution rather than spending time thinking about how to put their thoughts down on paper. Furthermore, our approach provides a structure that can be used to make the presentation of mathematics more consistent in textbooks and classrooms. Due to

the well-defined syntax and simple structure, structured derivations are also well suited for presentation on the web.

3 Structured Derivations at High School Level

As stated in the introduction, Finnish students can graduate from high school without being exposed to logic or proof in their mathematics courses. This is alarming not only from a computing perspective, but from a science perspective in general. Thus, although our main concern as CS educators is to ensure that our students possess sufficient mathematical skills in order to be able to progress successfully in their studies, we feel that attention also should be put on mathematics education at lower levels. We summarize our experiences from introducing structured derivations at high school in this section, and describe our experiences from introducing the same method into a CS syllabus in the next section.

The two mathematics syllabi offered in Finnish high schools have different foci: the general syllabus focuses on developing the capabilities needed “to use mathematics in different situations in life and in further studies” [15, p. 119], whereas the advanced syllabus focuses on learning to “understand the nature of mathematical knowledge” [15, p. 122]. The advanced syllabus is practically the norm for students seeking admission to universities for further studies in, for instance, mathematics, CS, engineering, medicine and physics. Considering the need for mathematical maturity in these fields, the students would most certainly benefit from getting more training in formal reasoning and proof already at high school level (as mentioned before, these topics are currently only mentioned in one advanced elective course). This does, however, not necessarily imply that more specific courses on logic should be introduced, but rather that logic should be integrated in other courses [16].

In 2001, a longitudinal study was initiated in a high school in Turku, Finland [5,17]. The aim of the study was to investigate whether structured derivations could be used to integrate logic, proof and formal reasoning throughout high school mathematics education without the need for additional courses on logic. The research setting involved a test group and a control group, which were followed up during their entire high school period (three years). The students chose which group to belong to, but care was taken to ensure that the entry level of the students was as similar as possible in both groups. The groups had different teachers who taught the exact same material, only using different approaches; the test group teacher rewrote and taught all ten compulsory mathematics courses using structured derivations, whereas the control group teacher gave the courses in his usual presentation style. Moreover, the test group teacher spent a few hours at the beginning of the first course introducing basic notions of elementary logic and giving students formal and informal practice in working with logical connectives.

The results from the study were positive, indicating that logical notation and structured derivations can be successfully used in a high school setting. Students in the test group consistently outperformed the control group in all ten courses [5] as well as in the matriculation exam [17].⁵

⁵ Students take the matriculation exam at the end of their high school studies. The matriculations examination board approves of the use of structured derivations in the matriculation exam.

In addition to this longitudinal study, the approach has also been introduced in single courses at three other high schools. Clearly, this renders a completely different situation than when all courses are given using the same format. Despite the limited time available for the teacher to present the approach and for the students to get familiar with it and use it, the results from these courses have also been positive. Surveys and observations have shown that despite a somewhat negative initial reaction to the new strict format requiring additional writing, most students learned to use and appreciate the structured approach during one single course.

We are now in the process of developing more systematic teaching material to support the use of structured derivations in mathematics education. Back and von Wright have written “Mathematics with a Little Bit of Logic” [8], a text book that introduces the approach and that can also be used as a teachers manual. Moreover, two ordinary high school mathematics text books have been “translated” into structured derivations. In addition, all assignments in ten complete mathematics matriculation exams have been solved using structured derivations (altogether 150 solved problems). This collection of solutions is important not only as an example base, but also as a confirmation that the approach can in fact be applied on a wide variety of problems.

4 Structured Derivations for First Year CS Students

The need for practical skills in proving mathematical theorems becomes evident to our CS students already during their first year courses. A compulsory course on logic was introduced in the basic studies in the CS curriculum at our department already in the 1990s, but as it was rather theoretical, students did not see the connection to the real world and felt that the course did not give them any skills that could be useful in practice. The course was totally redesigned in fall 2006, when structured derivations were introduced to put more focus on enhancing students’ logical reasoning and proof-writing abilities in practice.

The course was attended by 47 students and included 36 lectures (of 45 minutes), six exercise sessions (of 90 minutes) and a final exam. A pre- and postcourse survey as well as observations were used to evaluate the course and students’ opinions about the approach. The idea of the course was to apply structured derivations to high school mathematics. We thought that applying the rigorous derivational format on familiar problems would make it easier for students to learn the methodology, as they would not have to learn any new mathematics at the same time. This, however, did not work as intended. Instead, the familiar domain hindered the students from seeing the purpose of the course, thinking that the course was just a repetition of high school mathematics, failing to understand the importance of the format used. The initial confusion was partly a result of miscommunication, as the teaching assistant did not enforce the use of the structured derivations format in the exercise sessions. This gave the students a message that was not consistent with the one they received during the lectures.

Resistance to relearn familiar material using a new format is understandable; why should one start using a new approach for doing something one has already been doing successfully in another way for 12 years. However, as the students

realized that the topics covered in the course were indeed intended to be familiar, and that structured derivations was the new thing that they were supposed to learn, the resistance faded away. The results from the final exam were good (70% of the students passed, 30% with the highest grade) and the final feedback was in general positive. In the following, we list some of the positive and negative aspects brought up by students in the open questions of the post course survey.⁶

Students identified many of the same benefits of using structured derivations as was originally hypothesized when the approach was developed:

- *“When you write out everything, careless mistakes disappear”*
- *“Writing better mathematical derivations: same motivations as earlier, but more logically constructed”*
- *“Learning a systematic way of working”*
- *“I learned to think deeper on mathematical solutions”*
- *“Useful to practice problem solving, structure and divide problems”*
- *“The different proof strategies will be useful in math courses. But the logical motivations were also important to learn considering how to prove one’s programs”*

Some students found writing the derivations a bit tedious, but nevertheless found the approach interesting and useful.

- *“They feel a bit unnecessary sometimes, but on the other hand you see much more clearly what you’ve meant when you look at the solution again later”*
- *“Structured derivations feels like unnecessary work, but the format does make the calculations clearer”*
- *“In many cases I feel that structured derivations is a way of complicating simple things. Sure, you should be able to motivate what you’re doing, but there’s no need to exaggerate. On a suitable level of abstraction, this is, however, an interesting way of thinking”*
- *“The derivations were important. Even if you don’t like them, you may appreciate them more during further studies”*

Students also appreciated the structured derivations simply because it was a new format that appealed to them.

- *“The approach was pretty difficult and therefore interesting”*
- *“Interesting to learn how to write them and understand why you should write them”*
- *“The approach was interesting. I’ve always had problems proving things”*

The negative aspects brought up by the students were mainly related to the motivations and assumptions in proofs. One student also mentioned difficulties remembering the syntax of the format.

- *“Feels somewhat unnecessary to motivate everything”*
- *“I think it’s difficult to write down the motivations as many of them are obvious”*
- *“Difficult to know when a derivation is correct. What can you assume and what can’t you assume?”*
- *“Difficult to remember how to write them correctly”*

Finally, some students seemed to have a negative attitude towards mathematics in general.

- *“Derivations are always uninteresting”*
- *“I’m not interested in derivations and I don’t think I’ll need it in future CS studies”*

We were pleased to see that only a couple of students expressed negative opinions towards mathematics after the course. Most students stated that they appreciated the approach and the benefits it provides (clarifies solutions, facilitates debugging, makes relations explicit, enforces a systematic way of working, etc). As was found in the feedback on the high school courses, students initially felt frustrated with the extra writing needed in the form of motivations, but still found the format useful. We feel that the students’ feedback indicates that the majority of them did no longer

⁶ The quotations have been freely translated into English by one of the authors.

fear notation after this particular course. In our opinion, this is an encouraging finding.

Based on the experience from the first version of the course, some revisions were made before giving the course again starting in fall 2007. The main changes made were that instead of applying structured derivations on high school topics, the approach is now exemplified with problems in areas that the students are not familiar with from before: propositional and predicate logic, discrete mathematics, elementary algebra, lattice theory and boolean algebra. The focus is still on using the structured derivations framework and on emphasizing the need to develop skills in constructing mathematical proofs in practice. This course is going on as we write this, and the final evaluation is thus yet to be done. However, our preliminary observations indicate that students now “got the point” of the course from the very beginning, have used the structured derivations format in their own solutions and appreciate learning new mathematical topics.

5 Discussion

The experience from using structured derivations in education has been encouraging both at secondary and tertiary level. Although the results of introducing the approach in individual high school courses have been positive, we nevertheless believe that integrating logic as a tool in all mathematics courses is to be preferred. The effects of one single course are easily canceled out by the remaining courses not mentioning logic or proof at all.

Our experiences from teaching the structured derivations course as an introductory CS course has also been very encouraging, so much that structured derivations is now the standard approach used in the logic course. We are also planning a new course for first year university students in natural sciences and engineering. The course will be specifically designed as a bridging course between high school and university, focusing on improving students’ proficiency in doing mathematical proofs with structured derivations.

We are presently working on tool support for making structured derivation proofs, both on a personal computer and on the web. We use T_EXmacs [1], a wysiwyg L^AT_EX editor, as the basic framework for writing mathematical documents. We have constructed a plug-in for T_EXmacs that understands structured derivations and makes it easy to construct and browse derivations. In particular, T_EXmacs now supports selective hiding and revealing of subderivations and lemma proofs. This has made it straightforward for both teachers and students to work with derivations and proofs in electronic format.

Moreover, we are also currently working on providing just-in-time on-the-spot assistance for students reading a mathematical proof. For instance, consider a step in a derivation that calls for solving an equation, like in the second example given in Section 2. In that example, the equation was solved in a subderivation. If all subderivations are initially hidden, then students who feel confident about how to solve an equation do not need to open the subderivation, while students who are uncertain can do that. Thus, one single example can be used for students at different skill levels. This feature also renders structured derivations suitable for self-study

material, as examples can be made self-explanatory on different levels, providing the reader the choice of different levels of detail.

References

- [1] Texmacs homepage. Available online: <http://www.texmacs.org>. Retrieved on December 2, 2007.
- [2] Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, 2001.
- [3] Vicki Lynn Almstrum. *Limitations in the Understanding of Mathematical Logic by Novice Computer Science Students*. PhD thesis, Department of Computer Science, University of Texas, 1994.
- [4] Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculational proofs. *Formal Aspects of Computing*, 9:469–483, 1998.
- [5] Ralph-Johan Back, Mia Peltomäki, Tapio Salakoski, and Joakim von Wright. Structured derivations supporting high-school mathematics. In A. Laine, J. Lavonen, and V. Meisalo, editors, *Current Research on Mathematics and Science Education*. Department of Applied Sciences of Education, University of Helsinki, 2004.
- [6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.
- [7] Ralph-Johan Back and Joakim von Wright. A method for teaching rigorous mathematical reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, Aug 1999.
- [8] Ralph-Johan Back and Joakim von Wright. *Mathematics with a Little Bit of Logic: Structured Derivations in High-School Mathematics*. Manuscript, 2006.
- [9] Tommy Dreyfus. Why Johnny Can't Prove. *Educational Studies in Mathematics*, 38:85–109, 1999.
- [10] David Gries. Equational logic as a tool. In *AMAST '95: Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology*, pages 1–17, London, UK, 1995. Springer-Verlag.
- [11] David Gries. *Teaching and Learning Formal Methods*, chapter Improving the curriculum through the teaching of calculation and discrimination, pages 181–196. Academic Press, London, 1996.
- [12] Gila Hanna. Challenges to the importance of proof. *For the Learning of Mathematics*, 15(3):42–49, November 1995.
- [13] Kirsti Hemmi. *Approaching Proof in a Community of Mathematical Practice*. PhD thesis, Department of Mathematics, Stockholm University, 2006.
- [14] Uri Leron. Structuring mathematical proofs. *American Mathematical Monthly*, 90(3):174–185, 1983.
- [15] Finnish National Board of Education. National core curriculum for upper secondary schools 2003, 2003.
- [16] The ASL Committee on Logic and Education. Guidelines for logic education. *The Bulletin of Symbolic Logic*, 1(1):4–7, 1995.
- [17] Mia Peltomäki and Tapio Salakoski. Strict logical notation is not a part of the problem but a part of the solution for teaching high-school mathematics. In *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education - Koli Calling*, pages 116–120, 2004.
- [18] A. H. Schoenfeld. What do we know about mathematics curricula? *Journal of Mathematical Behavior*, 13(1):55–80, 1994.

Paper II

Promoting Students' Justification Skills Using Structured Derivations

L. Mannila & S. Wallin

Originally published in *ICMI 19: 19th ICMI Study Conference on Proof and Proving in Mathematics Education*. Taipei, Taiwan, May 2009.

PROMOTING STUDENTS' JUSTIFICATION SKILLS USING STRUCTURED DERIVATIONS

Linda Mannila and Solveig Wallin

Åbo Akademi University, Finland

Being able to explain the process of solving a mathematical problem is essential to learning mathematics. Unfortunately, students are not used to justifying their solutions as emphasis in the classroom is usually put on the final answer. In this paper, we describe how students can become used to explicate their thinking while solving a problem or writing a proof in a structured and standard format using structured derivations. We also present the results from an analysis of upper secondary school students' argumentation skills from using this approach in a course on logic and number theory. Our findings suggest that the structured derivations format is appreciated by the students and can help promote their justification skills.

BACKGROUND

“Mathematics is not just about identifying the truth but also about proving that this is the case” (Almeida, 1995, p. 171). Learning to argue about mathematical ideas and justifying solutions is fundamental to truly understanding mathematics and learning to think mathematically.

The National Council of Teaching Mathematics (NCTM) issues recommendations for school mathematics at different levels. In the current documents (NCTM, 2008), communication, argumentation and justification skills are recognized as central to the learning of mathematics at all levels.

According to Sfard (Sfard, 2001), thinking can be seen as a special case of intrapersonal communication: “[o]ur thinking is clearly a dialogical endeavor where we inform ourselves, we argue, we ask questions, and we wait for our own response [...] becoming a participant in mathematical discourse is tantamount to learning to *think* in a mathematical way” (p.5). Although it is important to be able to communicate mathematical ideas orally, documenting the thinking in writing can be even more efficient for developing understanding (Albert, 2000).

Justifications are not only important to the student, but also to the teacher, as the explanations (not the final answer) make it possible for the teacher to study the growth of mathematical understanding (Pirie & Kieren, 1992). Using arguments such as “Because my teacher said so” or “I can see it” is insufficient to reveal their reasoning (Dreyfus, 1999). A brief answer such as “ $26/65=2/5$ ” does not tell the reader anything about the student’s understanding. What if he or she has “seen” that this is the result after simply removing the number six (6)?

Nevertheless, quick and correct answers are often valued more in the classroom than the thinking that resulted in those answers. It is common for students to be

required to justify their solution and explain their thinking only when they have made an error – the need to justify correctly solved problems is usually de-emphasized (Glass & Maher, 2004). As a result, students rarely provide explanations in mathematics class and are not used to justify their answers (Cai et al., 1996). Consequently, the reasoning that drives the solution forward remains implicit (Dreyfus, 1999; Leron, 1983).

In this paper, we will present an approach for doing mathematics carefully, which aids students in documenting their solutions and their thinking process. We will also present the results from the analysis of students' justifications from a course using this approach. The aim is to investigate the following questions:

- How does the use of structured derivations affect students' justifications?
- What advantages and drawbacks do students experience when using structured derivations?

STRUCTURED DERIVATIONS

Structured derivations (Back et al., 1998; Back & von Wright, 1999; Back et al., 2008) is a further development of Dijkstra's calculational proof style, where Back and von Wright have added a mechanism for doing subderivations and for handling assumptions in proofs. With this extension, structured derivations can be seen as an alternative notation for Gentzen like proofs.

In the following, we illustrate the format by briefly discussing an example where we want to prove that $x^2 > x$ when $x > 1$.

- Prove that $x^2 > x$, when
 - $x > 1$
 - ||- $x^2 > x$
 - ≡ { Add $-x$ to both sides }
 $x^2 - x > 0$
 - ≡ { Factorize }
 $x(x - 1) > 0$
 - ≡ { Both x and $x-1$ are positive according to assumption. Therefore their product is also positive. }

T

The derivation starts with a description of the problem (“Prove that $x^2 > x$ ”), followed by a list of assumptions (here we have only one: $x > 1$). The turnstile (||-) indicates the beginning of the derivation and is followed by the start term ($x^2 > x$). In this example, the solution is reached by reducing the original term step by step. Each step in the derivation consists of two terms, a relation and an explicit justification for why the first term is transformed to the second one. Justifications are written inside curly brackets.

Another key feature of this format is the possibility to present derivations at different levels of detail using subderivations, but as these are not the focus of this paper, we have chosen not to present them here. For information on subderivations and a more detailed introduction to the format, please see the book and articles by Back et al.

Why Use in Education?

As each step in the solution is justified, the final product contains a documentation of the thinking that the student was engaged in while completing the derivation, as opposed to the implicit reasoning mentioned by Dreyfus (1999) and Leron (1983). The explicated thinking facilitates reading and debugging both for students and teachers.

Moreover, the defined format gives students a standardized model for how solutions and proofs are to be written. This can aid in removing the confusion that has commonly been the result of teachers and books presenting different formats for the same thing (Dreyfus, 1999). A clear and familiar format has the potential to function as mental support, giving students belief in their own skills to solve the problem. As solutions and proofs look the same way using structured derivations, the traditional “fear” of proof might be eased. Furthermore, the use of subderivations renders the format suitable for new types of assignments and self-study material, as examples can be made self-explanatory at different detail levels.

STUDY SETTINGS

The data were collected during an elective advanced mathematics course on logic and number theory (about 30 hours) at two upper secondary schools in Turku, Finland during fall 2007. Twenty two (22) students participated in the course (32 % girls, 68 % boys). The students were on their final study year.

For this study, we have used a pre course survey including a pretest, three course exams and a mid and post course survey. The pretest included five exercises, which students were to solve. They were also asked explicitly to justify their results. The surveys included both multiple choice questions and open-ended questions for students to express their opinions in their own words.

For each course exam, we have manually gone through and analyzed three assignment solutions per student, giving us a total of 198 analyzed solutions (22 students * 3 exams * 3 solutions). In the analysis, we focused on two things: the types of justification related errors (JRE) and the frequency of these.

RESULTS AND DISCUSSION

Justification related errors in the exams

The analysis revealed the following three JRE types:

- *Missing justification.* A justification between two terms in the derivation is missing.

- *Insufficient or incorrect justification.* E.g. using the wrong name of a rule or not being precise enough, for instance, writing “logic” as the justification, when a more detailed explanation would have been needed.
- *Errors related to the use of mathematical language.* Characterized by the student not being familiar with the mathematical terminology. For instance, one student wrote “solve the equation” when actually multiplying two binomials or simplifying an inequality.

The pre course survey indicated that the students had quite varied justification skills. Over half of the students disagreed with the statement “I usually justify my solutions carefully” and an analysis of the pretests showed that many students did do quite poorly on the justification part, especially for the two most difficult exercises (over 50 % of the students gave an incorrect or no explanation). Also, the nature of the justifications was rather mixed: whereas some gave detailed explanations, some only wrote a couple of words giving an indication of what they had done.

The exam assignments included surprisingly few JREs taking into account the skills exhibited by students in the pretest. The overall frequency of JREs stayed rather constant throughout the course: a JRE was found in 15-20 % of the 66 assignments analyzed for each exam. Most students who made a JRE of a specific type, made only one such error in the nine assignments. Note that this is one erroneous justification comment throughout all three exams. Only six students made more than one JRE of a specific type.

Missing justifications were the most common JRE in the second exam (11 % of students), whereas students did mainly insufficient/ incorrect justifications in the first and third exam (9-12 %). Errors related to mathematical language stayed fairly constant in all exams (3-6 %).

The low number of missing justifications in the first exam is understandable given the character of the assignments (short, familiar topics). In the second exam, new topics had been introduced, resulting in a larger number of missing justifications. This however decreased in the third exam, suggesting that students had got used to always justifying each step. The slightly increased number of insufficient/ incorrect justifications in the third exam can be explained by the third exam being the most difficult one. The main point here is to note that the overall frequency of JREs was low.

Survey results

The mid and post course surveys revealed students’ perceived benefits and drawbacks of using structured derivations. Our analysis showed that 77 % of the students stated that the solutions were much clearer than before. Further another 77 % suggested an increased understanding for doing mathematics.¹

¹ The quotations have been freely translated from Swedish by one of the authors.

“At first I found it completely unnecessary to write this way, but now I think it is a very good way, because now I understand exactly how all assignments are done.”

“I actually liked this course (rare when it comes to mathematics), structured derivations made everything much clearer. Earlier, I basically just wrote something except real justifications. Sometimes I haven’t known what I’ve been doing.”

The main drawbacks, according to the students, were that the format made solutions longer (32 % of students) and more time consuming (55 % of students). This is understandable, as the explicit justifications do increase the length of the solutions and also take some time to write down. The justifications, however, were considered a source of increasing understanding, thus the time consumption might be regarded something positive after all. In fact, we believe it is a large benefit, as it helps promote quality instead of quantity.

The students also noted that structured derivations required more thinking. Moreover, they recognized that the format helped them make fewer errors partly because they had to let it take time to write down the solutions.

“In this course the calculations become more careful since you take the time to think every step through.”

“[Using the traditional format, you] can more easily make mistakes when you calculate so fast.”

Another interesting finding was that students seemed to believe that justifications were not part of the solutions when doing mathematics in the traditional format. Describing the traditional way they do mathematics, they e.g. noted:

“You don’t have to explain what you do!... It’s enough to get a reasonable answer.”

“You lack explanations for why you do things the way you do.”

A final remarkable observation was the lack of completely negative comments. Comments starting out in a negative tone (“It takes much time”, “I don’t like all the writing”), all ended up positive (“... but I understand what I do better”, “...but I make fewer errors”). In our opinion, this is a promising finding.

CONCLUDING REMARKS

The format and results presented in this paper suggest that it is possible to get students to start justifying their solutions better. If you want to do something carefully, it will take some time and effort. “Quality before quantity” is something that, in our opinion, should be emphasized also in mathematics education.

The focus on also explaining solutions raises a new challenge - how do we get students to choose an appropriate level of detail for their justifications. While talented students may feel comfortable using “simplify” as a justification, this might not be sufficient for weaker students. A certain level of detail thus needs

to be enforced at least at the beginning of a new topic, in order to ensure that students truly are learning the topic at hand.

Another question raised that merits further investigations is what type of justification should be preferred (name of a mathematical rule, natural language description of the process, i.e. what is done in the step)? The impact of the type of justification (“simplify” compared to a longer description) on the quality/correctness of a solution also deserves attention.

REFERENCES

- Albert, L.R. (2000). Outside-In – Inside-Out: Seventh-Grade Students’ Mathematical Thought Processes. *Educational Studies in Mathematics*, 41(2), pp. 109-141.
- Almeida, D. (1995). Mathematics Undergraduates’ Perceptions of Proof. *Teaching Mathematics and Its Applications*, 14(4), p. 171-177.
- Back, R-J., Mannila, L., Peltomäki, M. & Sibelius P. (2008). Structured Derivations: a Logic Based Approach to Teaching Mathematics. FORMED08, Budapest, March.
- Back, R-J. & von Wright, J. (1998). *Refinement Calculus – A Systematic Introduction*. Springer.
- Back, R-J. & von Wright, J. (1999). A Method for Teaching Rigorous Mathematical Reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, August 1999.
- Cai, J., Jakabcsin, M.S. & Lane, S. (1996). Assessing Students’ Mathematical Communication. *School Science and Mathematics*, 96(5), p. 238-246.
- Dreyfus, T. (1999). Why Johnny Can’t Prove. *Educational Studies in Mathematics*, 38(1-3), pp. 85-109.
- Glass, B. & Maher, C.A. (2004). Students Problem Solving and Justification. *Proceedings of the 28th Conference of the International Group for the Psychology of Mathematics Education*, vol 2, p. 463-470.
- Leron, U. (1983). Structuring Mathematical Proofs. *American Mathematical Monthly*, 90(3), pp. 174-185.
- NCTM (2008). *Standards for School Mathematics*. Available online: <http://standards.nctm.org>. Accessed on June 12, 2008.
- Pirie, & Kieren (1992). Creating Constructivist Environments and Constructing Creative Mathematics. *Educational Studies in Mathematics*, 23, pp. 505-528.
- Sfard, A. (2001). Learning Mathematics as Developing a Discourse. In R. Speiser, C. Maher, C. Walter (Eds), *Proceedings of the 21st Conference on PME-NA*. Columbus, Ohio.

Paper III

“It Takes Me Longer, But I Understand Better” – Student Feedback
on Structured Derivations

R-J. Back, L. Mannila & S. Wallin

Originally published in *TUCS Technical Reports*, number 943. Turku Centre for
Computer Science, April 2009.



“It Takes Me Longer, but I Understand Better” - Student Feedback on Structured Derivations

Ralph-Johan Back

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
backrj@abo.fi

Linda Mannila

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
linda.mannila@abo.fi

Solveig Wallin

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
solveig.wallin@abo.fi

Abstract

In this paper, we present the results from a qualitative study on students' initial reactions to the use of structured derivations in mathematics education. Our findings suggest that the approach increases the clarity of solutions and facilitates debugging of proofs. It also has potential to increase students' self-perceived level of understanding. Our findings indicate that the main drawbacks experienced by the students are related to time and length. Nevertheless, the overall feedback on the approach was found to be positive, thus encouraging further use of structured derivations in mathematics education.

Keywords: structured derivations, mathematics education, student feedback

TUCS Laboratory
Learning and Reasoning

1 Introduction

This paper is based on an alternative approach to teaching mathematics, which is founded on a systematic notation for proofs and derivations as well as the explicit use of logical notation and inference rules. We refer to this approach as *structured derivations*.

The rationale for introducing structured derivations in education is grounded on two common issues with the traditional way in which mathematics is taught. First, proof and logic are important foundations of mathematics, but are typically considered difficult and are therefore often avoided in mathematics education at, for instance, secondary level [5, 17, 19, 32]. Second, mathematical solutions have traditionally been presented using various formats by teachers and in text books, resulting in confusion among students [12]. Consequently, strong arguments have been presented in favor of more training in proof, logic and rigorous reasoning [16, 21] and different ways to deal with the lack of a common structure have been proposed [14, 22, 23]. Using structured derivations, it is possible to address both these issues.

Structured derivations has been evaluated in a classroom setting at both high school¹ and university level with promising results [3, 4, 5, 25, 28, 29]. The largest study so far was initiated in 2001 at a Finnish high school [5, 28, 29], when all ten compulsory courses in the advanced mathematics syllabus were rewritten and given using structured derivations. The study was conducted as a quasi experiment, involving a test group taught using structured derivations and a control group taught in the same way as earlier. Based on course and exam results, students in the test group consistently outperformed the control group in all ten courses as well as in the matriculation exam.² In addition, the drop out rate was lower in the test group.

The quasi experiment focused on student performance. Another important aspect when introducing a new approach is students' attitudes towards it; after all, a method that yields good result but is unappealing to students will not necessarily be successful in the long run. The aim of the present article is to bring light on students' opinions regarding structured derivations by investigating the following questions:

- What benefits and drawbacks do students experience when having used structured derivations for the first time in a course?
- Do students at high school and university level experience similar benefits and drawbacks?

¹In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16-19. The principal objective of these schools is to offer general education preparing students for future studies.

²Finnish high school students take the matriculation exam at the end of their high school studies.

The paper is arranged as follows. First we introduce structured derivations and discuss some examples to illustrate the approach. Thereafter, we describe the study settings and the methods used. Next we present and discuss the results. We conclude the paper with a brief summary and ideas for future work.

2 Structured Derivations

Back and von Wright first developed structured derivations [2, 3, 6, 7] as a way to present proofs in programming logic. Later they adapted the method to provide a practical approach to presenting proofs and derivations in an easily readable and well-structured format in high school mathematics. The approach is a further development of the *calculational style* [10, 11, 13, 34], where mechanisms for doing subderivations and for handling assumptions in proofs have been added. In essence, structured derivations can be seen as combining the calculational style with natural deduction.

A structured derivation has a general syntax that fixes the overall structure and layout of a proof/derivation. The syntax and the layout are easily parsable by a computer and thus render the format suitable for presenting mathematics digitally, for instance, on the web. The use of subderivations also makes the format suitable for new types of assignments and self-study material, as examples can be made self-explanatory at different detail levels.

In the following, we will illustrate structured derivations by solving three mathematical problems.

2.1 Example 1

We start by proving the inequality $(1 + a)(1 + b)(1 + c) \geq 1 + a + b + c$, when $a, b, c \geq 0$. The structured derivation that proves this theorem is shown below. The derivation starts by introducing the problem on a bulleted line. Next we state the assumptions that we are allowed to make, on lines starting with a dash. In this case, the assumption is that a, b and c are all non-negative. The proof starts with the “ \Vdash ” line.

- Show that $(1 + a)(1 + b)(1 + c) \geq 1 + a + b + c$

- when $a, b, c \geq 0$

\Vdash {combining = and \geq gives \geq }

$$(1 + a)(1 + b)(1 + c)$$

= {multiply the second and the third parentheses}

$$(1 + a)(1 + b + c + bc)$$

$$\begin{aligned}
&= && \{\text{multiply the remaining parentheses}\} \\
&&& 1 + b + c + bc + a + ab + ac + abc \\
\geq &&& \{\text{subtract the non-negative expression } ab+ac+bc+abc. \text{ The expression} \\
&&& \text{is non-negative as } a, b, c \text{ are positive}\} \\
&&& 1 + a + b + c
\end{aligned}$$

The proof transforms an initial expression to some desired form, where each step is a relation between two terms. The first step states that $(1 + a)(1 + b)(1 + c) = (1 + a)(1 + b + c + bc)$ and justifies this by a mathematical operation that is known to preserve equality. The proof proceeds in this way step by step, until we finally have shown that the expression that we started with is greater or equal to the last expression, $1 + a + b + c$.

The justification of a derivation step should explain why the indicated relationship holds between the terms. There is plenty of space for writing the justifications, as these are written on separate lines between the terms. This underlines the importance of justifying each step, since equal amount of space is provided for arithmetic expressions as for the justifications. The relation between the terms is written in a separate column to make it clearly visible.

2.2 Example 2

The standardized proof format is one of the central ideas of structured derivations. A second important aspect is the use of logical notation and logical reasoning in high school mathematics. Although logic is avoided in education at that level, it still abounds in the theory, exercises and examples covered.

Consider as an example the following problem, where we are to solve the equation $(x - 1)(x^2 + 1) = 0$.

$$\begin{aligned}
\bullet &&& (x - 1)(x^2 + 1) = 0 \\
\equiv &&& \{\text{zero product rule: } ab = 0 \equiv a = 0 \vee b = 0\} \\
&&& x - 1 = 0 \vee x^2 + 1 = 0 \\
\equiv &&& \{\text{add 1 to both sides in the left disjunct}\} \\
&&& x = 1 \vee x^2 + 1 = 0 \\
\equiv &&& \{\text{add } -1 \text{ to both sides in the right disjunct}\} \\
&&& x = 1 \vee x^2 = -1 \\
\equiv &&& \{\text{a square is never negative: } a^2 < 0 \equiv F\} \\
&&& x = 1 \vee F
\end{aligned}$$

\equiv {disjunction rule: $p \vee F \equiv p$ }

$$x = 1$$

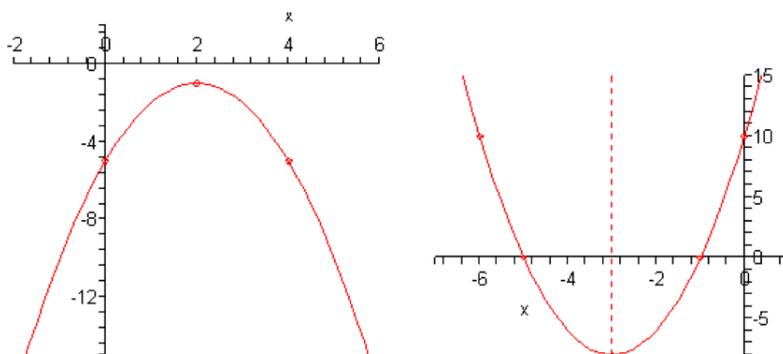
In traditional mathematics teaching, informal notations and descriptions are commonly used to hide the underlying logic. The structured derivation above, however, shows how explicit logical notation and logical reasoning can be used both to structure the derivation and to explain the individual reasoning steps. In this specific derivation, disjunction is used to keep two different branches of the derivation together, and in the final step, a specific logical inference rule is applied.

The explicit use of logical notation also has another notable advantage: the traditional mathematical notation has evolved over many centuries, and often provides different notations for things that are logically the same. Logical notation, on the other hand, is standardized and thus cuts down on the different things that students have to learn.

2.3 Example 3

Whereas the two examples above illustrate shorter derivations, the following one is longer and demonstrates the use of subderivations. A subderivation is typically used when we need to establish an auxiliary fact, but do not want to interrupt the main line of reasoning. A subderivation is indented one step to the right and starts with a bullet. To indicate where the main derivation continues we use an ellipsis (...) at the beginning of the corresponding line.

In this example, the task is to determine the values of constant a such that the function $f(x) = -x^2 + ax + a - 3$ is always negative. The problem is expressed using a universally quantified formula and the reasoning relies on two figures; this example thus also shows how graphs and other illustrations can be used in a structured derivation.



- Determine the values of constant a such that the function $f(x) = -x^2 + ax + a - 3$ is always negative.:

$$(\forall x \cdot -x^2 + ax + a - 3 < 0)$$

≡ {the function is a parabola that opens downwards as the coefficient for x^2 is negative; such a function is always negative if it does not intersect the x - axis (figure on the left)}

$$(\forall x \cdot -x^2 + ax + a - 3 \neq 0)$$

≡ {a second degree equation lacks solutions when its discriminant D is less than zero}

$$D < 0$$

≡ {substitute values for D }

- Compute the discriminant D :

$$D$$

= {the discriminant for the equation $Ax^2 + Bx + C = 0$ is $B^2 - 4AC$ }

$$a^2 - 4(-1)(a - 3)$$

= {simplify}

$$a^2 + 4a - 12$$

... $a^2 + 4a - 12 < 0$

≡ {the function $a^2 + 4a - 12$ is a parabola that opens upwards as the coefficient for a^2 is positive; such a function is negative between the intersection points with the x - axis (figure on the right)}

- Compute the places where $a^2 + 4a - 12$ intersects the x - axis:

$$a^2 + 4a - 12 = 0$$

≡ {square root formula}

$$a = \frac{-4 \pm \sqrt{4^2 - 4 \cdot 1 \cdot (-12)}}{2 \cdot 1}$$

≡ {simplify}

$$a = 2 \vee a = -6$$

... $-6 < a < 2$

The original proposition is reduced step by step to a form that explicitly gives the answer to the problem ($-6 < a < 2$). The main derivation includes two subderivations, one for determining the value of the discriminant, and one for solving the second-degree equation that arises.

Hiding and showing subderivations

A computer based editor for structured derivations makes it possible to display and hide the subderivations at will, so that a derivation can be constructed and studied at different levels of detail. Hiding the subderivations lets us see the overall structure of the proof, whereas showing them allows us to study and edit the proof at a more detailed level.

The following derivation shows the above derivation with both subderivations hidden.

- Determine the values of constant a such that the function $f(x) = -x^2 + ax + a - 3$ is always negative.:
 $(\forall x \cdot -x^2 + ax + a - 3 < 0)$
- \equiv {the function is a parabola that opens downwards as the coefficient for x^2 is negative; such a function is always negative if it does not intersect the x - axis (figure on the left)}
 $(\forall x \cdot -x^2 + ax + a - 3 \neq 0)$
- \equiv {a second degree equation lacks solutions when its discriminant D is less than zero}
 $D < 0$
- \equiv {substitute values for D }
- ... $a^2 + 4a - 12 < 0$
- \equiv {the function $a^2 + 4a - 12$ is a parabola that opens upwards as the coefficient for a^2 is positive; such a function is negative between the intersection points with the x - axis (figure on the right)}
- ... $-6 < a < 2$

3 Study Settings

3.1 Data Collection

The data for the study were collected in fall 2007, when the approach was used in individual courses at both high school and university level. The high school course was an elective advanced course on logic and number theory, whereas the university course was a first year introductory logic course for computer science students. Both courses included approximately 30 lessons (45 minutes) in class. In total, 22 high school students (32% girls and 68% boys) and 24 first year university students (8% girls and 92% boys) participated in the study.

A questionnaire including multiple choice questions and open-ended items was distributed to the students at the end of the courses. In this study we focus on the following open ended questions:

- What do you feel are the *benefits* of solving problems using structured derivations?
- What do you feel are the *drawbacks* of solving problems using structured derivations?

We also asked the same questions for the traditional methods – i.e. what are the benefits and drawbacks of solving problems using the traditional approach. Doing so, we hoped to catch several dimensions of students’ opinions, by allowing them to address the same questions from different perspectives.

3.2 Method

As the questions asked were open-ended, the data collected were of qualitative nature. Qualitative data are highly descriptive, and in order to interpret the information, the data need to be reduced. Content analysis was chosen for this purpose.

The basic idea of content analysis is to take textual material and analyze, reduce and summarize it using emergent themes [9]. These themes can then be quantified and hence content analysis is suitable for transforming rich data into a form which can be statistically analyzed.

The content analysis was done independently by two of the authors in order to ensure internal reliability. These authors reviewed half the questionnaires each, while at the same time listing the drawbacks and benefits for structured derivations and the traditional approach respectively. Most students mentioned several drawbacks and/or benefits, and these comments were split accordingly. This initial coding resulted in a detailed view of the students’ opinions. In order to further organize and reduce the data, the authors together reviewed the results of the analyses, discussed the findings and combined detailed comments into higher level categories. When the authors agreed on the categorization, all questionnaires were analyzed again using the resulting categorization as the coding scheme.

When the second round analysis had been completed, a quantitative approach was taken in order to make it possible to present the data in a diagrammatic form and to analyze them statistically. Percentages were calculated and presented both separately for the two courses and in total. Categories that were represented in 10% of student answers or less were combined into a single “other” category in order for the results not to become cluttered. The χ^2 - test was used to reveal any differences between high school students’ and university students’ opinions. Effect sizes were also calculated in order to give an indication of the practical importance of statistically significant findings.

4 Results

4.1 Benefits

The analysis revealed five main categories describing the benefits of structured derivations (Figure 1).

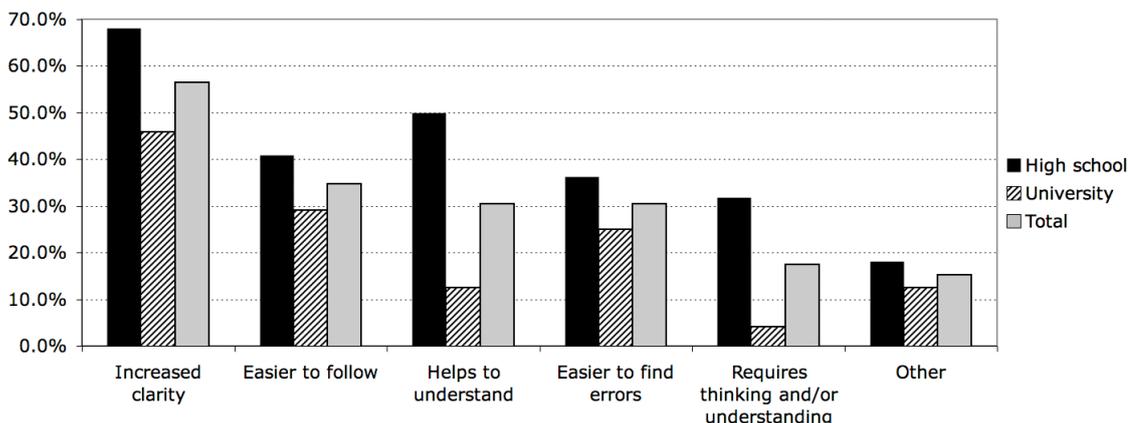


Figure 1: Experienced benefits of structured derivations (% of students)

As can be seen from the diagram above, there is a notable difference between high school and university level for the categories related to understanding. A statistical analysis using the χ^2 -test confirmed a significant difference between the two education levels on these categories (“Helps understand” and “Requires thinking and/or understanding”).

A larger number of high school students (50%) found that structured derivations helped them understand (“I know what I do”, “It helps me understand”) compared to the university students (13%) ($p=0.006$, $\chi^2=7.624$, $df=1$). The effect size indicated a relatively strong association ($\Phi=0.407$).

In a similar manner, high school students (32%) were more likely to appreciate structured derivations for the explicit requirements on thorough thinking and understanding compared to the university students (4%) ($p=0.013$, $\chi^2=6.109$, $df=1$). Here the effect size indicated a moderate association ($\Phi=0.364$).

No difference between education levels was found for the other categories, indicating that the opinions related to these categories were rather similar regardless of education level.

In the following, we will briefly introduce each of these categories separately. Quotations from the questionnaires, freely translated to English by two of the authors, are used for illustration purposes. Quotations marked with “HS” indicate that these are the words of high school students (HS1-HS22), whereas those marked with a “U” are excerpts from university students’ (U1-U24) comments.

Increased clarity

Nearly 60% of all students found that the approach increased the clarity of solutions and examples, making this the largest benefit category. The opinions in this category were of three main types:

1) explicitly talking about clarity,

“I actually liked this course (rare when it comes to mathematics). Structured derivations helped make everything clearer, earlier I’ve basically just written something down without any justifications and sometimes I haven’t known what I’ve been doing.” (HS10)

“It makes the solution clear so that you truly know what you do.” (HS13)

2) clarity originating from the justifications, and

“It’s clearer as each step is properly explained.” (U18)

3) clarity originating from the structured stepwise format.

“You clearly see step by step what you have done.” (HS21)

“You get a nice division (structure), easier to see the main thread.” (U15)

Easier to follow

Almost 35% of all students found that it was easy to follow a structured derivation, for instance when reviewing a solution or listening to the teacher’s explanations.

“Anyone can understand the solution.” (U8)

“It’s easier to follow along.” (U14)

Helps to understand

Nearly a third of all students reported that the format helped them understand both when solving a problem and looking at a solution later.

“At first I found it completely unnecessary to write this way, but now I think it is a very good way, because now I understand exactly how all assignments are done.” (HS7)

“You learn more about what you’re actually doing.” (HS4)

“It’s easier to know what you (or someone else) is thinking when everything is explained.” (HS3)

Students also found that structured derivations had potential to help others understand.

“Other people understand [your solutions] better.” (HS2)

Easier to find errors

Almost a third of all students experienced that structured derivations facilitated finding and correcting errors in their solutions.

“Easy to check what you have done.” (U3)

“The errors are easier to notice.” (HS14)

As a consequence, this improved their confidence in deciding whether a solution was correct or not.

“Most of the time you know if you have solved it correctly.” (U13)

“Easier to see if the solution is correct.” (HS1)

Requires thinking and/or understanding

The questionnaires also revealed that students, at high school in particular, appreciated the thinking required for solving a task using structured derivations.

“The solutions require more explanations and therefore become more thought through.” (HS14)

“You see what you do and must think things through thoroughly.” (HS8)

Students noted that in order to write reasonable justifications, they must understand what they are doing.

“Here you have to ‘tell a story’ about what you’re doing, which requires more time and puts your brain to work more.” (HS4)

“The justifications require that you understand what you have done.” (U19)

Other

The “other” category included opinions such as structured derivations being easier and more exact than the traditional format.

4.2 Drawbacks

Figure 2 illustrates the main categories for the perceived drawbacks related to structured derivations.

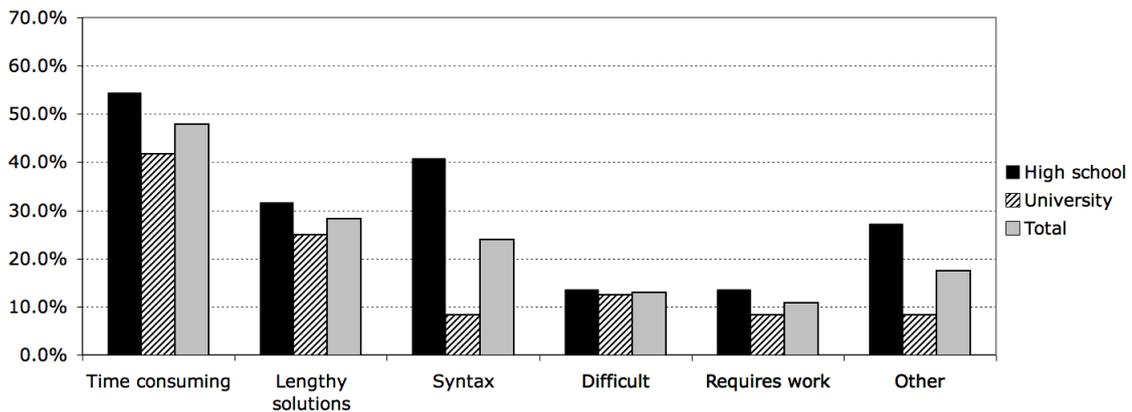


Figure 2: Experienced drawbacks of structured derivations (% of students)

Syntax difficulties were shown to be considered a drawback more frequently among high school students than at university level. Comparing the two education levels, the analysis indicated that high school students (41%) were more likely to express negative opinions with regard to the syntax than university students (8%) ($p=0.01$, $\chi^2=6.695$, $df=1$). The effect size for the difference showed a moderate association ($\Phi=0.381$).

In the following, we describe the drawback categories separately and again use excerpts from the questionnaires for illustration purposes.

Time consuming

The largest drawback of structured derivations was found to be time related, as almost half of all students commented on this aspect.

“It may take longer since you have to write more.” (U8)

“It takes more time as you also have to think about what to write in the curly brackets [i.e. the justifications].” (U9)

Lengthy solutions

This category is closely related to the previous one. Nearly 30% of all students found the length of structured derivations a drawback.

“A waste of paper when solving simple and trivial problems.” (U7)

“A lot to write, many unnecessary steps.” (U16)

Syntax

Almost 25% of all students experienced some kind of notational issues. Aspects of this category were, for instance, the need to master relevant terminology

“You need to know the names of all the rules.” (U14)

“You have to use difficult terminology.” (HS2)

and to remember the syntax of the format.

“All the dots, squares that you must remember take concentration away from the actual task.” (HS4)

“You can’t focus 100% on the problem.” (U17)

Difficult

Slightly more than 10% of the students found writing solutions using structured derivations difficult.

“If you don’t know what to do it’s difficult to continue.” (HS6)

Requires work

One out of ten students did not like the work required to solve a problem using structured derivations.

“You can’t ‘improvise’ since you need to understand and write down what you have done.” (HS3)

Other

The “other” category on the drawback side included feelings of confusion when using the method and the frustration of using the format for very simple tasks. Students also mentioned problems related to coming up with what to choose as the first term in a derivation as well as deciding on a suitable level of detail.

4.3 Benefits and Drawbacks of the Traditional Approach

As indicated in section 3.1, we also asked students about the benefits and drawbacks of the traditional approach used in their other mathematics courses. Although we did not ask the students to compare this approach to structured derivations, the results suggest that this is what happened in practice. This is understandable as the students were only to comment on two approaches; in such a situation it becomes quite natural to compare the two against each other.

The analysis revealed that these results were roughly the reverse of the results for structured derivations (Figure 3): the drawbacks found for the traditional method were more or less the opposites of the benefits experienced when using structured derivations. Similarly, many of the drawbacks experienced with structured derivations were reflected as benefits for the traditional approach.

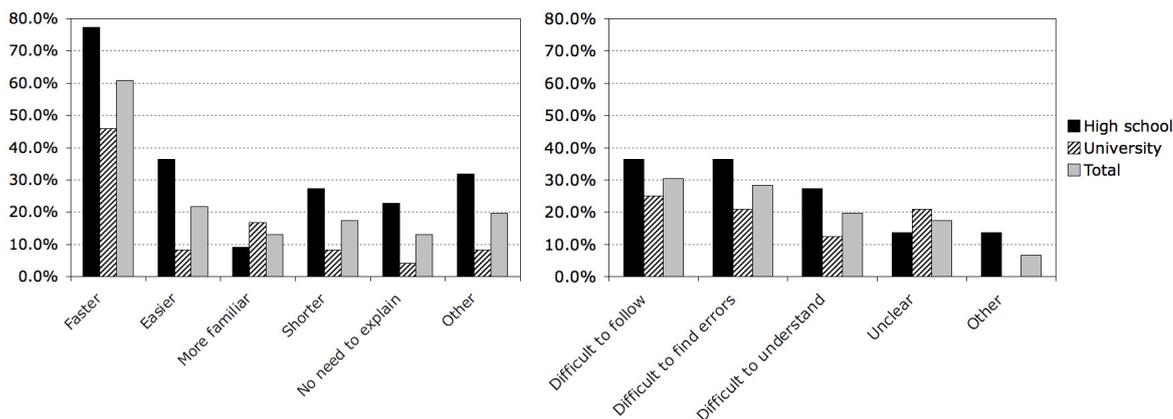


Figure 3: Benefits (to the left) and drawbacks (to the right) of the traditional approach (% of students).

The main benefit found for the traditional approach was time related: solving problems using the traditional format was not considered as time consuming. When comparing the two education levels, the χ^2 -test indicated that a larger proportion of high school students (77%) viewed “faster” as a major benefit of the traditional approach compared to university students (46%) ($p=0.029$, $\chi^2=4.763$, $df=1$). The effect size for the difference showed a moderate association ($\Phi=0.322$).

“You save much time [using the traditional approach].” (HS2)

“[The traditional approach] is considerably faster...” (HS18)

The traditional approach was also considered easier and shorter, apparently mainly due to not having to write any justifications.

“[Using the traditional approach], you have a bit more freedom when it comes to how exact you are and why you calculate in a certain way.” (HS7)

“You don’t have to explain what you do!... It’s enough to get a reasonable answer.” (HS6)

“... less space is needed.” (HS18)

One aspect not touched upon in the comments on structured derivations was revealed in those for the traditional method, namely that of familiarity. In total 13% of all students (9% and 17% for high school and university participants respectively) reported that the familiarity of the traditional method was a benefit, thus suggesting that the unfamiliarity of structured derivations could be a potential drawback. The familiarity aspect was found to be related either to what students themselves are used to

“[The traditional approach] is so ingrained that you might not need to think as much.” (U4)

“Have used [the traditional format] since first grade.” (HS12)

or to what people around them may know.

“The calculations we have done now could be difficult to understand if you hadn’t taken the course because of structured derivations.” (HS3)

“It’s easier to ask for help using the traditional approach.” (U19)

The drawbacks of the traditional approach were mainly related to lack of clarity, as students found solutions difficult to follow, check and understand. Although speed was considered one of the main benefits of the traditional approach, some students also found it a drawback: when doing calculations quickly, one does not necessarily spend enough time thinking about the solution, which leads to errors that can be difficult to find as the thinking is not documented in any way.

“[Using the traditional approach, you] can more easily make mistakes when you calculate so fast.” (HS4)

“It is more difficult to find careless mistakes.” (HS11)

“The risk for making mistakes and errors increases. The solution does not become as thought through as when using structured derivations.” (HS15)

5 Discussion

In this section we discuss the results presented above. We will arrange the discussion around the main findings aiming at addressing the research questions put forward in the introduction.

5.1 Clarity and Understanding

One of the main benefits of structured derivations was related to clarity and understanding. Most comments suggesting an increased understanding, directly or indirectly implied the increased clarity and justifications as reasons. Understanding is however quite difficult to define [20]. Knowing what the students meant when using the word is even more difficult, and the data for this study are not sufficient for drawing any conclusions about the intended meaning. Understanding is, nevertheless, a subjective phenomenon, and every person has a feeling for how well he or she understands things; even if one cannot know precisely what understanding is, one can still recognize it [20]. Regardless of its exact meaning it should thus be regarded as something positive when a student expresses an increased level of understanding.

High school students were found to be more likely to express appreciation for the understanding aspect than those at university level. This may naturally have many different causes, but one explanation could possibly be found in the course contents. The university course included only new material, and hence the students had nothing to compare their understanding to (i.e. they could not compare their learning and understanding during this course with any other on the same topic earlier). This was however possible for the high school students, as parts of the material covered in the high school course was familiar (e.g. equation solving). Thus, one can hypothesize that these students found that they now better understood topics they had seen before.

5.2 Time and Length

The main drawbacks of structured derivations were related to time and length. This is understandable, as the explicit justifications naturally increase the length of solutions and also take some time to write down. We will here discuss these two related issues separately.

Issue of time Research has shown that there is a wide spread belief among students that all problems can be solved quickly [15]. Therefore, students are inclined to give up if they do not succeed in solving a given problem in a short amount of time. For instance, Schoenfeld [31] found that high school students think 12 minutes is a reasonable amount of time to spend on a given problem before concluding that it is impossible. The longest response was 20 minutes.

A related issue is wait time in the mathematics classroom, i.e. the amount of time the teacher waits after asking a question before picking a student to answer it or before answering it him/herself. Studies have shown that an extended wait time is associated with higher achievement levels among students when it comes to challenging problems [18]. However, after simple questions, longer wait time is counterproductive and does not make sense. Thus, it is understandable that the justifications required in structured derivations feel frustrating for simple tasks.

The reluctance to spend time on a certain task can also be explained to some degree by considering the generation of students involved. They have all grown up with the Internet and thus have become accustomed to immediate feedback, expecting quick responses and no waiting time. This generation may even value the speed at which tasks are completed more than the level of accuracy of the result [27].

Whereas students considered the time consumption a drawback when solving problems themselves, they regarded it as a benefit when it came to the teacher.

“You have time to follow and understand why things should be in a certain way. It doesn’t just pass by.” (HS16)

Structured derivations force not only students but also teachers to calculate slower, which is essential in the mathematics classroom [33]. While teachers are used to handling concentrated ideas, students are not, and consequently need more time to digest the information.

Issue of length Solutions become longer using structured derivations mainly due to the justifications, which are written on separate lines. In the traditional way in which mathematics is taught, students do not get used to justifying their solutions [12] as they are commonly asked to explain their reasoning only when they have made an error [8, 15]. This was reflected in some questionnaire answers related to the traditional approach:

“You don’t always have to explain why you have done in a certain way as long as you get the correct result.” (HS7)

“You lack explanations for why you do things the way you do.” (HS10)

Clearly, it is then no wonder that students express an initial resistance to writing thorough justifications when they are required to. However, without justifications, the reasoning that drives the solution forward remains implicit [12, 23] and consequently it is more or less impossible for the teacher (and for the student him/herself) to assess whether a student has learned/understood or not [30]. Arguments such as “Because my teacher said so” or “I can see it” are insufficient to reveal any reasoning and a brief answer such as “ $26/65 = 2/5$ ” does not tell the reader anything about the student’s understanding [12]. What if he or she has “seen” that this is the result after simply removing the number six (6)?

The additional information given in the justifications was nevertheless appreciated by many students as a source of increased clarity:

“It’s easier to later remember how you have solved it.” (HS1)

“It’s easy to see what rule or assumptions each step is based upon.”
(U20)

Based on this discussion, we do not consider the length or time issues drawbacks. The length of the justifications and the detail level of the steps taken can be freely chosen. In the beginning, when learning a new topic, one would prefer to write detailed justifications, but later it is natural to start taking larger steps and abbreviating the justifications. Doing so, the length of the derivation decreases. In order to facilitate self-study, we still believe that material prepared by the teacher in electronic format should always be as detailed as possible. The details can be hidden using subderivations, but need to be available for those who feel they need to see the example or a model solution at a more detailed level.

As for the issue of time, we believe it is a benefit in disguise. When students spend more time on the solutions, they think things through more carefully. Thus we get into another benefit found in the study, that of facilitated error prevention and checking.

5.3 Ease of Finding Errors

Students (30%) found that structured derivations made it easier to check solutions and find errors as each step is justified.

“...the calculations become more careful since you take the time to think every step through.” (HS15)

The structured format and the documentation available in the justifications appear to make it easier to spot mistakes, as the derivation can be checked locally, one step at a time. Compare this to the traditional format, where one often has to start checking everything from the beginning in order to find a mistake. Research has shown that this type of error checking is time consuming; for instance, Lithner [24] found that it took a first-year undergraduate student seven minutes to find an error when going through a solution to a mathematical problem. Without justifications, one has to once again reconstruct the thought processes that took place when solving the problem in the first place in order to be able to follow the chain of reasoning.

Our results also indicate that the need for taking it slow when solving problems using structures derivations helps avoid making careless errors. Although mistakes made due to quick calculations can be considered insignificant, they do count, for instance, in assessment. More important, however, careless errors can transform or simplify a task so that the result answers something totally different from what was originally intended. Consider, for instance, the difference of simplifying the expression $x \geq 0 \wedge x < -3$ and the expression $x \geq 0 \vee x < -3$. Writing the incorrect connective by mistake can happen to anyone, but when you have to explain how you do the simplification you are more likely to pay more attention to the original expression, thus increasing the chances of finding and avoiding the mistake in the first place.

To illustrate how a careless error can simplify a problem, we show a solution to a derivative problem given by a student in a traditional mathematics course:

$$\begin{aligned} D\left(\frac{x+1}{x^2+1}\right) &= D\left(\frac{x+1}{(x+1)(x-1)}\right) = D\left(\frac{1}{x-1}\right) = D(x-1)^{-1} = \\ &= -1(x-1)^{-2} \cdot 1 = (-x+1)^{-2} = \frac{1}{(1-x)^2} \end{aligned}$$

Here, the problem was simplified already in the very first step, where the student incorrectly factorized $x^2 + 1$ as $(x + 1)(x - 1)$. Instead of calculating the derivative of a fraction, the task was simplified to finding the derivative of a polynomial raised to a negative power.

5.4 Structure and Syntax

Research has suggested that a standard format can be of great assistance when learning mathematics [14, p. 70]:

“if ordinary students are to make genuine progress in mathematics, almost all need standard templates of this kind to provide a framework

- (a) within which their solutions can be presented and checked,
- (b) by means of which they can be expected to organise a sequence of steps in a way that makes plain to the reader the validity of the final conclusion, and
- (c) through which their understanding of proof, and their acceptance of responsibility for identifying and correcting errors can mature.”

As mentioned in the introduction, there is, however, no clear format for how solutions should be presented in traditional mathematics education. As a result, students are left confused [12]. By only writing down the main steps of e.g. the process of solving an equation, much important information is overlooked. By condensing the solution, many steps and arguments that the teacher might have made orally in the classroom become implicit. Thus, much information that could have aided in understanding and reconstructing the solution is left out [22].

Using structured derivations, the standardized format gives students a concrete model for how solutions and proofs are to be written. The format also has potential to make the presentation of mathematics more consistent in textbooks and in the classroom. In addition, a familiar format can act as mental support, giving students belief in their own skills to solve a problem. This can be especially important when considering the common “fear” for proof found among students. When proofs and simple calculational derivations are written using the same format, students may feel less intimidated by proofs – these look just as familiar as solutions to other mathematical problems.

Our analysis indicated a significant difference in how students at high school felt about the syntax compared to the university students, as the latter were less likely to express negative opinions with regard to the syntax. Some university students even explicitly mentioned the syntax and the format as benefits. One possible explanation can be that the university students were in the computing field and took a programming course at the same time. Both of these circumstances are likely to have made these students more open to following a given syntax, whereas high school students may not have had to follow a certain format in any course.

5.5 Difficulty and Unfamiliarity

Some students found it difficult to write solutions using structured derivations. The comments were unfortunately rather vague, and did not reveal whether the difficulties lay in the format *per se* or e.g. in the assignments. In the latter case, the assignments would be difficult to solve regardless of the format. It does, however, seem reasonable to assume that the difficulties could be related to structured derivations being a completely new format for the students. Although unfamiliarity

was not specifically mentioned as a drawback of structured derivations, roughly 10% of the students found the familiarity of the traditional format beneficial. This touches upon two important aspects: students' resistance to change and possibilities to get guidance outside the classroom.

Resistance to change All students in this study had been studying mathematics since first grade, that is, for over ten years. Thus, they had already been initiated into an acceptable practice, making them reluctant to changes in the ways they are taught [26]. This resistance is understandable and should be expected. Every new approach requires an investment of time and energy in acquiring new skills with no certainty of payoff. This may be further aggravated in situations where students are not unhappy with their previous learning experiences, producing little incentive for change [1].

Guidance outside the classroom Structured derivations does not introduce any new mathematics, only a format for structuring and presenting proofs and derivations in a clearer and more consistent way. Whereas parents may not be able to help with the presentation of solutions, the format might reduce their need to explain the same thing repeatedly, since each example and solution is justified and can be used as reference by the students. Parents are also likely to more easily be able to follow and check a solution using structured derivations compared to an unstructured one without any justifications.

5.6 Additional Observations

Structured derivations introduces several new features compared to the traditional way in which mathematics is presented: the use of logical notation, subderivations and justifications. Our findings suggest that students, after taking one course using the format, recognize and appreciate the justifications. We did not find a single comment with regard to, for instance, the use of connectives, relations or subderivations. There are several possible explanations for this. First of all, both the high school and university course were on logic, which may explain why students did not explicitly comment on the use of connectives and relations; these were simply part of the course content. On the other hand, we do not precisely know what students were referring to when making negative comments with regard to the syntax; these might be related to the logical notation to some extent. Especially the relations can in the beginning quite easily be viewed as merely part of the syntax instead of as an essential part of the logic of the solution.

6 Summary and Future Work

In this paper, we have introduced structured derivations and presented the results from a study of students' reactions to the new approach. Our findings suggest that the fixed format and the justifications bring many benefits. As each step is justified, the final product contains a documentation of the thinking that the teacher or the student was engaged in while completing the derivation. Hence, the format and the justifications facilitate presentation of proofs and derivations in class. Moreover, the clarity of proofs and derivations is increased, making them easier to read, check and correct, both for students and teachers. Structured derivations also has potential to increase students' self-perceived level of understanding.

As is the case with all changes, instructors should expect and be prepared for some initial reluctance when introducing a new format for writing mathematics. Our findings suggest that time and length are the main drawbacks experienced by the students. As discussed above, the time spent on solving a problem is associated with many benefits, and we do therefore not consider that aspect a drawback. With regard to the length issue, abbreviating the justifications and chunking the problem in larger steps can reduce the length of the solution — *after* the student has become used to the format and feels on top of the topic at hand. This should, however, never be done to such an extent that the readability of the derivation suffers.

We are continuously broadening the use of the method to new schools and education levels. In addition, we have received funding for teacher's training from the Finnish National Board on Education, and are currently (spring 2009) educating mathematics teachers in using this approach. From a research perspective, many interesting questions are still to be answered. One topical question is whether an adapted version of the format can be used at lower levels of education than high schools. Another point of interest is related to the use of computer tools for structured derivations in educational settings.

References

- [1] Gerlese S. Akerlind and A. Chris Trevitt. Enhancing self-directed learning through educational technology: When students resist the change. *Innovations in Education and Training International*, 6(2):96–105, 1999.
- [2] Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculation proof. *Formal Aspects of Computing*, 9:469–483, 1997.
- [3] Ralph-Johan Back, Linda Mannila, Mia Peltomäki, and Patrick Sibelius. Structured derivations: a logic based approach to teaching mathematics. In *FORMED08*, 2008.

- [4] Ralph-Johan Back, Linda Mannila, and Solveig Wallin. Student justifications in high-school mathematics. In *CERME 6: Sixth Conference of European Research in Mathematics Education*, Lyon, France, 2009.
- [5] Ralph-Johan Back, Mia Peltomäki, Tapio Salakoski, and Joakim von Wright. Structured derivations supporting high-school mathematics. In A. Laine, J. Lavonen, and V. Meisalo, editors, *Current Research on Mathematics and Science Education*. Department of Applied Sciences of Education, University of Helsinki, 2004.
- [6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.
- [7] Ralph-Johan Back and Joakim von Wright. A method for teaching rigorous mathematical reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, Aug 1999.
- [8] J. Cai, M.S. Jakabcsin, and S. Lane. Assessing students' mathematical communication. *Schools Science and Mathematics*, 96(5):238–246, 1996.
- [9] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research Methods in Education*. Routledge, 6 edition, 2007.
- [10] Edsger W. Dijkstra. The notational conventions i adopted, and why. *Formal Aspects of Computing*, 14(2):99–107, 2002. EWD 1300.
- [11] Edsger W. Dijkstra and Carel S. Scholten. Predicate Calculus and Program Semantics. *Texts and Monographs in Computer Science*, pages 21–29, 1990.
- [12] Tommy Dreyfus. Why johnny can't prove. *Educational Studies in Mathematics*, 38(1-3):85–109, 1999.
- [13] W. H. Feijen. Exercises in formula manipulation. *Formal development programs and proofs*, pages 139–158, 1990.
- [14] Tony Gardiner. Learning to prove: Using structured templates for multi-step calculations as an introduction to local deduction. *ZDM*, 36(2):67–76, 2004.
- [15] Barbara Glass and Carolyn A. Maher. Students problem solving and justification. In *Proceedings of the 28th Conference of the International Group for the Psychology of Mathematics Education*, volume 2, pages 463–470, 2004.
- [16] G. Hanna and H. N. Jahnke. Proof and application. *Educational Studies in Mathematics*, 24:421–438, 1993.
- [17] Gila Hanna. Challenges to the importance of proof. *For the Learning of Mathematics*, 15(3):42–49, November 1995.

- [18] Aiso Heinze and Markus Erhard. How much time do students have to think about teacher questions? an investigation of the quick succession of teacher questions and student responses in the german mathematics classroom. *ZDM*, 38(5):388–398, 2006.
- [19] Kirsti Hemmi. *Approaching Proof in a Community of Mathematical Practice*. PhD thesis, Department of Mathematics, Stockholm University, 2006.
- [20] R. Hersch. Proving is convincing and explaining. *Educational Studies in Mathematics*, 24:389–399, 1993.
- [21] C. Hoyles. The curricular shaping of students’ approaches to proof. *For the Learning of Mathematics*, 17(1):7–16, 1997.
- [22] Christine Knipping. A method for revealing structures of argumentations in classroom proving processes. *ZDM Mathematics Education*, 40:427–441, 2008.
- [23] Uri Leron. Structuring mathematical proof. *American Mathematical Monthly*, 90(3):174–185, 1983.
- [24] Johan Lithner. Mathematical reasoning and familiar procedures. *International Journal of Mathematics Education in Science and Tehcnology*, 31(1):83–95, 2000.
- [25] Linda Mannila and Solveig Wallin. Promoting students’ justification skills using structured derivations. In *ICMI 19: 19th ICMI Study Conference on Proof and Proving in Mathematics Education*, Taiwan, May 2009.
- [26] Richard A. McCray, Robert L. DeHaan, and Julie Anne Schuck, editors. *Improving Undergraduate Instruction in Science, Technology, Engineering, and Mathematics: Report of a Workshop*. Center for Education, 2003.
- [27] Diana G. Oblinger and James L. Oblinger. *Educating the Net Generation*, chapter Is it Age or IT: First Steps Toward Understanding the Net Generation. EDUCAUSE, 2005.
- [28] Mia Peltomäki and Ralph-Johan Back. An empirical evaluation of structured derivations in high school mathematics. In *ICMI 19: 19th ICMI Study Conference on Proof and Proving in Mathematics Education*, 2009.
- [29] Mia Peltomäki and Tapio Salakoski. Strict logical notation is not a part of the problem but a part of the solution for teaching high-school mathematics. In *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education - Koli Calling*, pages 116–120, 2004.
- [30] Susan Pirie and Thomas Kieren. Creating constructivist environments and constructing creative mathematics. *Educational Studies in Mathematics*, 23:505–528, 1992.

- [31] A. H. Schoenfeld. Exploration of students' mathematical beliefs and behavior. *Journal for Research in Mathematics Education*, pages 338–351, 1989.
- [32] A. H. Schoenfeld. What do we know about mathematics curricula? *Journal of Mathematical Behavior*, 13(1):55–80, 1994.
- [33] R.R. Skemp. Relational understanding and instrumental understanding. *Mathematics Teaching*, 77:20–26, 1976.
- [34] Antonetta J.M. van Gasteren. On the Shape of Mathematical Arguments. *Lecture Notes in Computer Science*, pages 90–120, 1990.

Paper IV

Student Justifications in High School Mathematics

R-J. Back, L. Mannila & S. Wallin

Originally published in *CERME 6: Sixth Conference of European Research in Mathematics Education*. Lyon, France, January-February 2009.

STUDENT JUSTIFICATIONS IN HIGH SCHOOL MATHEMATICS

Ralph-Johan Back, Linda Mannila and Solveig Wallin

Åbo Akademi University, Finland

In this paper, we continue our previous work on evaluating the use of structured derivations in the mathematics classroom. We have studied student justifications in 132 exam solutions and described the types of justifications found. We also discuss the results in light of Skemp's (1976) framework for relational and instrumental understanding.

Keywords: student justifications, structured derivations, high school, instrumental and relational understanding

INTRODUCTION

The ability to justify a step in, for instance, a proof can be considered a skill that needs to be mastered, at least to some extent, before proof is introduced. In a wider sense, proof can even be regarded as justification (Ball and Bass, 2003). Unfortunately, students are not used to justify their solutions (Dreyfus, 1999). It is common for teachers to ask students to explain their reasoning only when they have made an error; the need to justify correctly solved problems is usually de-emphasized (Glass & Maher, 2004). Consequently, without the explanations, the reasoning that drives the solution forward remains implicit (Dreyfus, 1999; Leron, 1983).

A previous study (Mannila & Wallin, 2008) indicated that high school students can improve their justification skills in one single course. In this paper, we will present the results from a follow-up study, focusing on the types of justifications given by the students. We will first discuss some related work and also give a brief introduction to the approach used when teaching the course. The main research questions are the following: What types of justifications do students give in a solution? Do the types of justifications change as the course progresses, and in that case how?

RELATED WORK

Justifications as a condition for proof

The importance of proof and formal reasoning for the development of mathematical understanding is also recognized by the National Council of Teaching Mathematics (NCTM), which issues recommendations for school mathematics at different levels. According to the current document (NCTM, 2008), students at all levels should, for instance, be able to communicate their mathematical thinking, analyze the thinking of others, use mathematical language to express ideas precisely, and develop and evaluate mathematical arguments and proof. While discussing mathematical ideas is important, communicating mathematical thinking in writing can be even more efficient for developing understanding (Albert, 2000).

To think mathematically, students must learn how to justify their results; to explain why they think they are correct, and to convince their teacher and fellow students. “[M]athematical reasoning is as fundamental to knowing and using mathematics as comprehension of text is to reading. Readers

who can only decode words can hardly be said to know how to read. ... Likewise, merely being able to operate mathematically does not assure being able to do and use mathematics in useful ways.” (Ball & Bass, 2003; p. 29)

Justifications are not only important to the student but also to the teacher, as the explanations (not the final answer) make it possible for the teacher to study the growth of mathematical understanding. Using arguments such as “Because my teacher said so” or “I can see it” is insufficient to reveal their reasoning (Dreyfus, 1999). A brief answer such as “ $26/65=2/5$ ” does not tell the reader anything about the student’s understanding. What if he or she has “seen” that this is the result after simply removing the number six (6)?

Types of understanding and reasoning

A review of literature on mathematics education shows that there is an interest in studying the distinction between being able to apply a determined set of instruction in order to solve a mathematical problem and being able to explain the solution by basing it on mathematical foundations. Several frameworks have been presented for investigating types of learning and understanding.

Skemp (1976) discusses two types of understanding named by Mellin-Olsen: *relational* (“knowing both what to do and why”) and *instrumental* (“knowing what”, “rules without reasons”). People who exhibit an instrumental understanding know how to use a given rule and may think they understand when they really do not. For instance, getting the correct result when applying a given formula is an example of instrumental, not relational, understanding. One typical example can be found in equation solving, where students learn to “move terms to the other side and change the sign”, without necessarily knowing why they do it.

Sfard (1991) investigates the role of algorithms in mathematical thinking and discusses how mathematical concepts can be perceived in two ways: as objects and as processes. Pirie and Kieren (1999) present a theory of the growth of mathematical understanding and its different levels. More recently, Lithner (2008) has created a research framework for different types of mathematical reasoning, distinguishing between two main types: *imitative* and *creative*. Imitative reasoning is rote learnt and can be divided into two subtypes: memorised reasoning, where the student, for instance, solves a problem by recalling a full answer given in the text book or by the teacher, and algorithmic reasoning, where a problem is solved by recalling and applying a given algorithm. The other main type, creative reasoning, includes a novel reasoning sequence, which can be justified and is based on mathematical foundations. One of the main differences between imitative and creative reasoning is that the former does not necessarily involve analytical and conceptual thinking, whereas such thinking processes are essential to creative reasoning.

STRUCTURED DERIVATIONS

Structured derivations is a logic-based approach to teaching mathematics (Back & von Wright, 1998; Back & von Wright, 1999; Back et al, 2008a). The format is a further development of Dijkstra's calculational proof style, where Back and von Wright have added a mechanism for doing subderivations and for handling assumptions in proofs. Using this approach, each step in a solution/proof is explicitly justified.

In the following, we illustrate the format by briefly discussing an example where we want to prove that $x^2 > x$ when $x > 1$.

- Prove that $x^2 > x$: *task*
- $x > 1$ *assumption*
- ||- $x^2 > x$ *term*
- ≡ { Add $-x$ to both sides } *justification*
- $x^2 - x > 0$ *term*
- ≡ { Factorize } *...*
- $x(x - 1) > 0$
- ≡ { Both x and $x-1$ are positive according to assumption. Hence, their product is also positive }
- T*

The derivation starts with a description of the task (“Prove that $x^2 > x$ ”), followed by a list of assumptions (here we have only one: $x > 1$). The turnstile (||-) indicates the beginning of the derivation and is followed by the start term ($x^2 > x$). In this example, the solution is reached by reducing the original term step by step. Each step in the derivation consists of two terms, a relation and an explicit justification for why the first term is transformed to the second one.

Another key feature of this format is the possibility to present derivations at different levels of detail using subderivations, but as these are not the focus of this paper, we have chosen not to present them here. For information on subderivations and a more detailed introduction to the format, please see the articles by Back et al. referred to above.

Why use in education?

As each step in the solution is justified, the final product contains a documentation of the thinking that the student was engaged in while completing the derivation, as opposed to the implicit reasoning mentioned by Dreyfus (1999) and Leron (1983). The explicated thinking facilitates reading and debugging both for students and teachers. According to a feedback analysis (Back et al., 2008b), students appreciate the need to justify each step of their solutions. They also find that the justifications makes solutions easier to follow and understand both during construction and afterwards.

Moreover, the defined format gives students a standardized model for how solutions and proofs are to be written. This can aid in removing the confusion that has commonly been the result of teachers and books presenting different formats for the same thing (Dreyfus, 1999). A clear and familiar format also has the potential to function as mental support, giving students belief in their own skills to solve the problem. Also, as solutions and proofs look the same way using structured derivations, the traditional “fear” of proof might be eased. Furthermore, the use of subderivations renders the format suitable for new types of assignments and self-study material, as examples can be made self-explanatory at different detail levels.

STUDY SETTINGS

Data collection

The data were collected during an elective advanced mathematics course on logic and number theory (about 30 hours in class) that was taught at two high schools in Turku, Finland, during fall 2007. All in all, twenty-two (22) students completed the course at either school and participated in the study (32 % girls, 68 % boys). The students were on their final study year.

The course included three exams held after 1/3, 2/3 and at the end of the course. The exams were of increasing difficulty level, i.e. the first was the easiest and the last the most difficult one. Two assignments from each were chosen for the analysis. Hence, we have in total analyzed 132 solutions (six solutions for each student) written as structured derivations.

The assignments analyzed were the following:

- A1: Determine the truth value of the expression $(x^2 + 3 \leq 7 \wedge y < x - 4) \vee x + y \leq 5$, when $x = 2$ and $y = 4$.
- A2: Solve the equation $|x - 4| = 2x - 1$.
- A3: Use de Morgan's law $(\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q)$ to determine if the expression $(\neg p \vee \neg q) \wedge (p \wedge q)$ is a tautology or a contradiction.
- A4: Prove that $b^2 - d^2 = ad + bc - ab - cd$ if $a + b = c - d$.
- A5: Prove or contradict the following: For any integers m and n , it is the case that if $m \cdot n$ is an even number, then both m and n are even.
- A6: Prove that $2 + 14^{30} \equiv_{13} 106 + 27^{30}$.

The topics covered in assignments A1 and A2 were familiar to the students from previous mathematics courses. The aim of these assignments was mainly to let students practice structured derivations and writing solutions using the new format.

The topics covered in the rest of the analyzed assignments (A3-A6) were new to the students. A3 and A4 focused on logical concepts and manipulation of logical expressions, whereas A5 and A6 covered number theory.

Method

The data collected, i.e. the justifications, were of qualitative nature. Qualitative data are highly descriptive, and in order to interpret the information, the data need to be reduced. In this study, a content-analytical approach was chosen for this purpose. The basic idea of content analysis is to take texts and analyze, reduce and summarize them using emergent themes. These themes can then be quantified, and as such, content analysis is suitable for transforming textual material into a form, which can be statistically analyzed (Cohen, 2007).

A first round of the content analysis was done by one of the authors, who analyzed 18 solutions from E1 and 24 solutions from E2. This initial coding resulted in a first view of the types of justifications. The authors discussed the results and agreed on how to combine the detailed justifications into higher-level categories. Next, all solutions were analyzed using the preliminary categories as the coding scheme. The second round analysis showed that the categories found in the

initial phase were sufficient for covering all justifications found in the 132 solutions. A quantitative approach was then taken in order to be able to illustrate the results graphically.

The use of both quantitative and qualitative methods has several benefits. Mixed methods avoid any potential bias originating from using one single method, as each method has its strengths and weaknesses. A mixed methods approach also allows the researcher to analyze and describe the same phenomenon from different perspectives and exploring diverse research questions. Whereas questions looking to describe a phenomenon ("How/What..?", our first research question) are best answered using a qualitative approach, quantitative methods are better at addressing more factual questions ("Do...", our second research question) (Cohen, 2007).

RESULTS

The content analysis revealed five main justification types:

- *Assumption*: Referral to an assumption given in the assignment directly or in a rewritten format.
- *Vague/broad statement*: A very brief and uninformative justification type: "logic" or "simplify".
- *Rule*: Referral to a name of a rule or a definition, e.g. the rule for absolute values, tautology, congruence etc. In some cases, the justification also included the rule explicitly written out in text.
- *Procedural description*: An explanation of *what* is done in the step, i.e. a description including a verb. E.g. "add $2x + 4$ to both sides", "move 3 to the other side and change the sign" and "calculate the sum".
- *Own explanation*: An explanation for *why* the step is valid in own words and/or with symbols, e.g. " $2k^2 + 2k$ is an integer if k is an integer. Therefore $2(2k^2 + 2k)$ is an even integer". In some justifications a mathematical definition was written out in own words, e.g. " $2 \equiv_{13} 106$ because $2 - 106 = -104, 13 \mid -104$ ".

Figure 1 illustrates the proportion of different justification types found in the assignments respectively. The diagram also shows how the types of justification used varied depending on the assignment.

Some justification types are highly assignment specific. For instance, assumptions can naturally only be used in assignments where assumptions are present. In such assignments, it is common for the assumption to be used only once or twice, and the proportion of this type of justification will be rather low. The analysis showed that all students but one were able to handle assumptions correctly already in the first exam, i.e. after 1/3 of the course.

The use of rules can also be considered assignment specific. For instance, when manipulating logical expressions, rules become important as these make up the basis for the manipulation. When students gave a rule as a justification, most usually stated only the name of the rule, whereas only a few also wrote out the rule itself. In the final and most difficult assignment, where the rule was central to the solution, a larger proportion of students (46 %) had written it out explicitly, compared to those who had only provided the name of the rule (22 %).

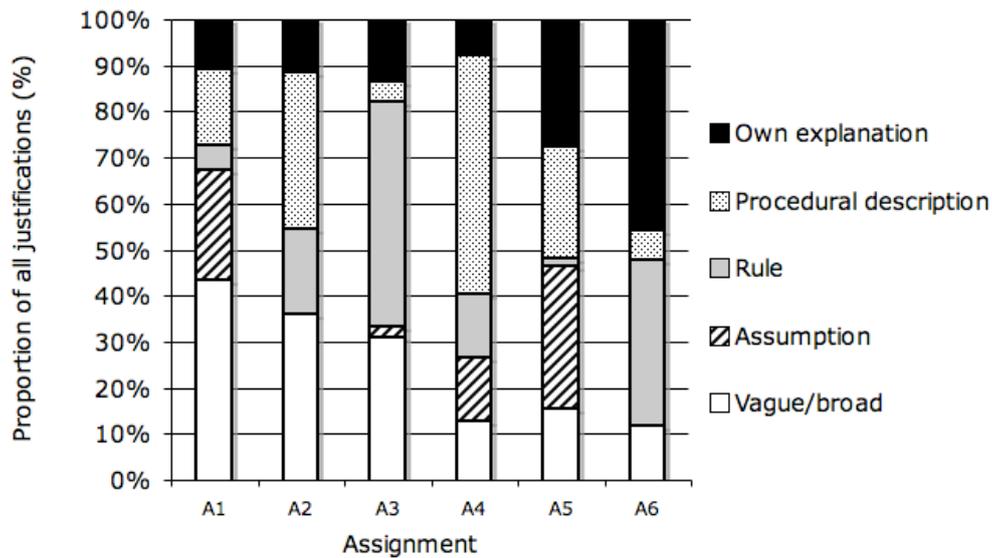


Figure 1: The proportion of justifications of different types in the six assignments

In addition to these specific dependencies, the analysis also revealed some other relationships. The assignments in the first exam (A1-A2) were not trivial but still familiar to the students (determine the value of an expression and solve an equation), who consequently mainly used short justifications (vague/broad, assumption, rule). Given the nature of equations, the solutions to A2 also contained a large proportion of procedural descriptions (“move 3 to the other side”).

In the second exam, students faced assignments (A3-A4) that were not as familiar anymore. In A3, students were to make explicit use of logical rules, which, as stated above, naturally has an impact on the types of justifications: almost half of all justifications referred to a given rule. The following assignment, A4, called for a formal proof (the Finnish high school curriculum does not include proofs in any other course than the elective one described in this paper). As the expression used in the proof was an equation, the main justification type used was, again, procedural descriptions.

The third exam (A5-A6) is probably the most interesting one from a research perspective. The assignments were in a completely new domain, with which students had no prior experience: constructing proofs in number theory. Thus, these assignments have potential to provide insight into how students use justifications when adventuring into a new terrain. As indicated in the diagram (figure 1), the proportion of own explanations increased, in particular at the expense of the less informative justification type “vague/broad”.

DISCUSSION

As seen above, the justification types changed throughout the course. Whereas some of the variation (e.g. the use of assumptions and rules) is a direct result of the nature of the task at hand, some seems to be more related to the perceived level of difficulty.

For instance, the most noticeable changes are found for “vague/broad” justifications and “own explanations”: whereas the former dominate the solutions early on, their frequency decreases towards the end as the number of the latter increases. The first exam was easier than the final one, and as easy assignments include more “straightforward” steps, students may not have seen the need

to justify those steps in any more detail. Rather, students seem to find the need to justify more carefully as the assignments become more difficult. Consequently, the occurrence of own explanations increase. Similarly, it is understandable that students are reluctant to write lengthy justifications when solving tasks similar to tasks they have solved many times before, whereas they may feel a need for writing more careful justifications in assignments that deal with new topics. This is supported by the results from our feedback study, where students found “extra writing” unnecessary for simple tasks (Back et al., 2008b).

Can justifications aid in assessing understanding?

Only two justifications types, “own explanations” and “procedural descriptions”, involve students writing in their own words. There is an important difference between these types. In a “procedural description”, students write *what* they do, but not *why* they have chosen or are allowed to do so. The “own explanation”, on the other hand, also gives information regarding *why* the step is valid.

This is closely related to Skemp’s instrumental and relational understanding (1976). Own explanations are clearly relational, but the remaining four types (vague/broad, assumption, rule, procedural descriptions) cannot easily be mapped to either type of understanding. We will therefore refer to own explanations as “relational justifications” and the other four types as “instrumental justifications”.

Although Skemp argued that instrumental justifications such as “move -3 to the other side” are examples of an instrumental approach to understanding, we do not think the situation is as black-or-white. For instance, a simple justification such as “logic” may be the result of complex thought processes. Knowing that students are not keen on writing, one can also assume that students may choose to write a short justification even in places where they could have been more expressive in order to indicate their understanding. An instrumental justification simply does not reveal enough information about whether the student has truly understood what he or she has done. Ruling out the possibility of relational understanding in such situations requires more than a mere justification.

To exemplify this, we now look at three different solutions to an assignment involving absolute values. The absolute value rule referred to below is the following: $|x| = c \Leftrightarrow (x = c \vee x = -c) \wedge c \geq 0$

- **Tom:** *instrumental justification, relational understanding*

Tom did not use the rule for absolute values learnt in class, but rewrote the expression in a way showing that he had really understood the absolute value concept. The solution was correct and indicated a relational understanding of absolute values.

$$|x - 4| = 2x - 1$$

\Leftrightarrow { rewrite the absolute value }

$$(x - 4 = 2x - 1 \wedge 2x - 1 \geq 0) \vee (-x + 4 = 2x - 1 \wedge 2x - 1 \geq 0)$$

- **Layla:** *instrumental justification, instrumental or relational understanding*

Layla used the absolute value rule and solved the problem correctly. Despite the correct solution, we cannot know whether Layla understood the concept or merely used a rule she had learnt that “should work” for this type of problems.

$$|x - 4| = 2x - 1$$

⇔ { rule for absolute values }

$$(x - 4 = 2x - 1 \vee x - 4 = -2x + 1) \wedge 2x - 1 \geq 0$$

- **Joe:** *instrumental justification, instrumental understanding*

Just like Layla, Joe also justified the initial step with “the rule for absolute values”. However, he used the rule incorrectly, as he “forgot” the second part of it (the requirement on x).

$$|x - 4| = 2x - 1$$

⇔ { rule for absolute values }

$$x - 4 = 2x - 1 \vee x - 4 = -2x + 1$$

This was a rather common error in our study (made by almost 36% of all students in assignment A2). Had Joe had a relational understanding for absolute values, the additional requirement would have been clear to him even if he had forgotten what the rule looked like.

Thus, it seems as if one can in fact conclude that a given instrumental justification is *not* an example of relational understanding – this is the case if the step is incorrect as for Joe above. However, doing the opposite, i.e. concluding that an instrumental justification to a correct step is relational, is not as straightforward.

Is a clearly relational approach always needed?

In high school mathematics, much time is spent on things like solving equations and simplifying expressions. Thus, to a large extent it boils down to using rules, and consequently a seemingly instrumental approach becomes dominant. However, this is foregone by the teacher explaining the theory behind the rules and the definitions. If the student later uses the rules in an instrumental or a relational way is up to how well he or she understood the theory. If the underlying concept is not clear to the students, the rules are most likely applied without reasons, i.e. instrumentally. One area of high school mathematics where relational understanding most likely becomes more evident is in textual problems, where students first need to formalize the problem specification. In order to correctly specify the problem, the student needs to understand the problem domain and the underlying concepts. Relational understanding is naturally also important when constructing proofs.

Furthermore, sometimes a justification with a seemingly instrumental approach is the best one that can be given. Take for example a complex trigonometric expression. Finnish high school students have a collection of rules that they can always have with them, even on exams. One can hardly require them to start explaining rules in order to be allowed to apply them. What is essential in such a situation is that they a) have an underlying understanding for trigonometry, b) know how to apply trigonometric rules correctly, and c) are able to manipulate the expression into a form where one of the many rules can be applied correctly.

As another example we can take equation solving and the “add -3 to both sides” type of instrumental justification mentioned above. Let us say we have two students: one who understands that whenever you have an expression of the form $a = b$, you can add the same value to both sides without changing the truth value of the full expression ($a + c = b + c$), and another who knows that

one should move “lonely numbers” to the other side while changing the sign. Both of these students would probably use similar justifications, but only one of them would have a relational understanding. This student would, however, hardly write out the rule ($a = b \Leftrightarrow a + c = b + c$), which would be needed in order for the teacher to be able to distinguish the justification from that given by the other student.

Justifications and validity of steps

As was described above, a seemingly “correct” justification can lead to an incorrect derivation step. This can happen for several reasons, one being the one exhibited by Joe above: not completely remembering a rule. Careless mistakes in a step do not seem to correlate with the type or the accuracy of the justification. Only a small number of this type of errors was found (in 9 % of the assignments throughout all three exams), which was also supported by students’ feedback as they pointed out that they made fewer careless mistakes using structured derivations than what they usually do (Back et al., 2008b).

CONCLUDING REMARKS

The type of justification chosen in a certain situation is closely related to the assignment and/or the step at hand. For example, assumptions or rules will not be used in problems where there are no assumptions or rules to apply. Our findings suggest that students choose the level of detail in their justifications mainly based on the difficulty level of the task at hand: in tasks that are familiar, students tend to opt for broad and vague justifications, whereas justifications which say more come into play as the topics covered are new and/or the assignments become more difficult. Especially justifications written in own words are of great importance to the teacher for understanding a solution and the student’s thinking; this is not necessarily the case for vague and broad justifications.

The study presented in this paper is a continuation on earlier qualitative studies on the use of structured derivations in education. Previous results indicate that students appreciate the approach (“it takes me longer, but I understand better”) and that it improves students’ justification skills as soon as during one single course (Mannila & Wallin, 2008). Furthermore, we have found that explicit justifications make students think more carefully when solving a problem (Back et al., 2008b). With this study, we now also have a rather clear picture of how students justify their solutions and how the justifications change throughout the course.

Getting students to clearly document their solutions step by step is a step forward, although “judging” the justifications is everything but straightforward. Thus, many questions still remain. Is it possible to teach a way of writing “good” justifications? And if we want to try, what characterizes such justifications? Another aspect, not considered so far, is related to teachers and course books. How do teachers justify their solutions when teaching using structured derivations? How are examples justified in texts? In order for students to develop relational understanding, we believe that it is essential that examples are explained freely (“using own words”) as often as possible.

REFERENCES

Albert, L.R. (2000). Outside-In – Inside-Out: Seventh-Grade Students’ Mathematical Thought Processes. *Educational Studies in Mathematics*, 41(2), 109-141.

- Back, R-J., Mannila, L., Peltomäki, M. & Sibelius P. (2008a). *Structured Derivations: a Logic Based Approach to Teaching Mathematics*. FORMED08, Budapest, Hungary, March 2008.
- Back, R-J., Mannila, L. & Wallin, S. (2008b) “It Takes Me Longer, but I Understand Better” - Student Feedback on Structured Derivations”. Work in progress.
- Back, R-J. & von Wright, J. (1998). *Refinement Calculus – A Systematic Introduction*. Springer.
- Back, R-J. & von Wright, J. (1999). *A Method for Teaching Rigorous Mathematical Reasoning*. In Proceedings of Int. Conference on Technology of Mathematics, University of Plymouth, UK, August 1999.
- Ball, D. L & Bass, H. (2003) Making Mathematics Reasonable in School. In Kilpatrick, J., Martin, W. G., Schifter, D. (Eds.) *A Research Companion to Principals and Standards for School Mathematics*, 27-44. Reston, VA.
- Cohen, L., Manion, L. & Morrison, K. (2007) *Research Methods in Education*. 6th Edition. London: Routledge New York.
- Dreyfus, T. (1999). Why Johnny Can't Prove. *Educational Studies in Mathematics*, 38(1-3), 85-109.
- Glass, B. & Maher, C.A. (2004). *Students Problem Solving and Justification*. Proc. of the 28th Conference of the International Group for the Psychology of Mathematics Education, 2, 463-470.
- Leron, U. (1983). Structuring Mathematical Proofs. *American Mathematical Monthly*, 90(3), 174-185.
- Lithner, J. (2008). A Research Framework for Creative and Imitative Reasoning. *Educational Studies in Mathematics*, 67, 255-276.
- Mannila, L. & Wallin, S. (2008). *Promoting Students Justification Skills Using Structured Derivations*. To be presented at ICMI 19, Taiwan, May 2009.
- Pirie, S. & Kieren, T. (1999). Growth in Mathematical Understanding: How Can We Characterise It and How Can We Represent It? *Educational Studies in Mathematics*, 26, 165-190.
- NCTM (2008). Standards for School Mathematics. Available online: <http://standards.nctm.org>. Accessed on June 12, 2008.
- Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational studies in mathematics*, 22, 1–36.
- Skemp, R.R. (1976). Relational Understanding and Instrumental Understanding. *Mathematics Teaching*, 77, 20-26.

Paper V

Why Complicate Things? Introducing Programming in High School Using Python

L. Grandell, M. Peltomäki, R-J. Back & T. Salakoski

Originally published in *Proceedings of the 8th Australasian Computing Education Conference (ACE2006)*, pp. 71-80. Hobart, Tasmania, January 2006.

Why Complicate Things?

Introducing Programming in High School Using Python

Linda Grandell, Mia Peltomäki, Ralph-Johan Back and Tapio Salakoski

Turku Centre for Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
Email: linda.grandell@abo.fi, mia.peltomaki@utu.fi, backrj@abo.fi, tapio.salakoski@utu.fi

Abstract

Deciding what to teach novices about programming and what programming language to use is a common topic for debate. Should an industry relevant programming language be taught, or should a language designed for teaching novices be used? Typically, these questions are raised at university level, but in this paper we address them from a high school perspective.

We present a case study with a twofold goal: (1) examining how programming can be introduced at high school level, and (2) evaluating how suitable the programming language Python is to support both teachers and learners in this process. During the school year 2004/2005, an introductory programming course was given to four student groups in two different high schools. The students enjoyed programming and learnt to think in terms of re-use and interfaces. In addition, we found that many features of Python facilitated both teaching and learning (for instance, a simple and flexible syntax, immediate feedback, easy-to-use modules and strict requirements on proper indentation).

Our findings support results from previous studies in that students have difficulties in dealing with abstract concepts - even though the syntax for implementing these is simple. In addition, compared to university students, high school students are young and have necessarily not yet developed the writing skills required for producing proper documentation. The course was designed to be well suited for high school students in general, but still all participants were boys. Since high schools should provide all-round learning to all students, we, as do all computer science teachers, face the challenge of making programming more appealing to girls.

1 Introduction

Computer Science in High Schools

Subjects that have previously been considered university level topics are increasingly introduced at lower levels of education; computer science (CS) is an example of such a topic (Ben-Ari 2004a). The aim of elementary and secondary education is to provide students with knowledge needed in every-day life, and considering the increasing role of computer technology and applications in today's society, CS can be seen as an essential part of this all-round learning.

Consequently, efforts have been made in order to introduce CS at high school (HS)¹ level: for instance, in the USA (Merritt et al. 1993) and in Israel (Gal-Ezer et al. 1995), CS curricula for HSs were developed in the 1990s, and according to the information network on education in Europe, *Eurydice*,² most European countries specified CS and programming as part of HS education in 2004. As CS education has become more common at HS level, the interest in studying this topic has increased; for example in 2004, an entire issue of the journal *Computer Science Education*³ was devoted to this topic.

Difficulties in Learning Programming

Although there seems to be a general agreement that CS should be taught at HS level, there are many differing ideas about what exactly should be taught. The main point of disagreement is usually whether programming should be included in CS curricula at HS level or not. Many studies have pointed out difficulties experienced by university students when learning programming. Spohrer and Soloway (1986) showed that a few bug types constitute the majority of the mistakes made by novice programmers: construct-based problems, which make it difficult to "learn the correct semantics of language constructs", and plan composition problems, which make it difficult to "put plans together correctly". Recent studies support these findings (for example (McCracken et al. 2001, Ben-Ari 1998)). Moreover, the results from a multi-national study conducted in 2004 (Lister et al. 2004) showed that students lack the skills needed to trace the execution of short pieces of code after taken their first course on programming. Other reported sources of difficulty or confusion have been related to, for instance, parameter passing (Fleury 1991), analogies (Halasz & Moran 1982) and mathematical notation (for example (Haberman & Kolikant 2001, Bayman & Mayer 1983)). For a further review of this topic see, for example, Robins et al. (2003).

Programming Languages in Education

The difficulties mentioned above seem to be related to the programming activity - not the programming language. Nevertheless, deciding what, and how, to teach novice programmers is not the only topic for debate. Equally common a topic is which programming language to teach. According to Palumbo (1990), the learning results in programming classes depend on the

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Eighth Australasian Computing Education Conference (ACE2006), Hobart, Tasmania, Australia, January 2006. Conferences in Research and Practice in Information Technology, Vol. 52. Denise Tolhurst and Samuel Mann, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16-19. The principal objective of these schools is to offer general education preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies. <http://www.edu.fi/english/page.asp?path=500,4699,4840,4845>

²<http://www.eurydice.org/Documents/KDICT/en/FrameSet.htm>

³<http://www.tandf.co.uk/journals/titles/08993408.asp>

time devoted to actual programming and language access. In order to maximize the time spent on programming, one should avoid having to "waste" time on discussing irrelevant language constructs and syntax errors.

According to Milbrandt (1993), a programming language to be used in education should be easy to learn, structured in design, universal in use and powerful in computing capability. The language should also have a simple syntax, provide easy I/O handling as well as output formatting, use meaningful keywords, give immediate feedback, etc.

Pascal and Logo were both designed with education in mind, and many studies have indicated the suitability of these in education (for example (Schollmeyer 1996, Shaffer 1986)). Unfortunately, both languages suffer from drawbacks that have led to decreasing significance over the years. For instance, Logo is seen as a language for children, and Pascal has not followed the same pace of development as more recent languages (deRaadt et al. 2002).

Today, C, Java and C++ top the list of the most popular languages based on the availability of engineers, courses and third party vendors around the world.⁴ Researchers have found that Java, Visual Basic, C++ and C are the languages most widely taught (or planned to be taught) at universities (deRaadt et al. 2002, Stephenson & West 1998). These results are further confirmed when looking at web pages of CS departments at universities worldwide. Despite their popularity, there has been much debate about the suitability of these languages in education, especially when introducing programming to novices (for example (Mody 1991, Hadjerrouit 1998, Biddle & Tempero 1998, Clark et al. 1998, Close et al. 2000)). The languages are, for example, considered too verbose, enforcing notational overhead that has little to do with learning to think algorithmically and to write structured programs. All the time spent on sorting out these issues is time away from actual programming.

Given the difficulties related to learning programming at university level, the skepticism against introducing the topic at HS level is understandable; after all, if something is considered problematic by university students, it is reasonable to assume that it is equally troublesome for younger students - especially when indications of the difficulties can be found also in studies conducted among HS students (for example (Aiken 1972, Ginat 2003, Haataja et al. 2001)). Most of the problems that have been discussed in the literature have, however, been based on the traditional instructional languages mentioned above; those who take a critical stand against programming being taught at lower levels might be doing so based on their experiences in teaching these languages. In that case, the criticism can be seen as aimed towards the *languages* and not at *programming* per se.

Should HS Students Learn Programming?

Prior programming experience has been shown to have a positive impact on university CS studies (Hagan & Markham 2000). However, programming skills are not only important to students pursuing careers as computer scientists. Researchers (for example (Mayer 1986, Treese 2003, Soloway 1993)) have studied the impact of programming on the development of thinking skills, and although the results vary, indications of the connection between programming and problem solving, algorithmic thinking and logical reasoning have been found. Students who will not become programmers will still benefit from these meta skills learnt in the process. Knowing how to

program also implies understanding how software and computers work, which may reduce computer anxiety and frustration. Moreover, having some knowledge of programming makes it easier to communicate with CS professionals. In addition, we feel that the aforementioned difficulties in learning programming make it even more motivated to start teaching programming at HS level; doing so, we can deal with some of the problematic issues related to learning to program at an early stage and thus improve the prior knowledge and skills of future university students. In our opinion, programming should therefore be included in CS curricula at HS level. Even though some students would not take the courses, they should be given the chance to do so.

In this paper, we present an approach to introducing programming and algorithmic thinking in HS using the programming language *Python*. Our aim is twofold: (1) examining how programming can be introduced at HS level, and (2) evaluating how Python is suited for supporting this teaching process. We begin by motivating our research, particularly by reviewing previous work relevant for our two goals. Thereafter we present the course, its goals, material and syllabus. Starting in fall 2004, the course has been given to four groups of HS students, and we present and discuss the results from these courses. We briefly discuss our own experiences before concluding the paper with some final remarks and suggestions for future work.

2 Motivation and Related Literature

Although Pascal and Logo have lost much of their significance, the same things that one earlier hoped to achieve using these are still desirable in programming education. In many respects, the aims of Python can be compared to those of Pascal and Logo, with the difference of Python being a popular language.⁵

Python is a high-level scripting language, originally designed by Guido van Rossum to facilitate learning; van Rossum has even suggested that everybody could master programming using Python (van Rossum 1999). The language has many of the features characteristic for a language suitable for teaching programming, for instance the following ones:

Small and clean syntax Compared to languages such as Java or C++, Python has a more intuitive syntax (code listing 1).

Dynamic typing Python is dynamically typed, which further reduces the notation.

Expressive semantics Python's basic types are powerful: for example, lists can be introduced at the same time as other built-in types.

Immediate feedback The interpreter enables fast and interactive demonstration of programming concepts, and gives immediate and understandable feedback on potential errors.

Enforced structural design Python enforces an indented and structured way of writing programs, and the code resembles pseudo code.

Relevant open-source software Python is free and widely used.⁶ It comes with a text editor (*IDLE*), and a large amount of tutorials, books, course material, exercises, assignments and extensive documentation is available on the web.

⁴<http://www.tiobe.com/tpci.htm>

⁵<http://www.tiobe.com/tpci.htm>

⁶<http://www.pythonology.org/success>

Code listing 1 *Comparison of a simple output program in Python and Java.*

```
Python      Java
print "Hi!"  class Hi {
              public static void main (String args[]) {
                System.out.println("Hi!");
              }
            }

              % javac Hi.java
              % java Hi
```

For a more detailed review of Python's features see the official website⁷ or the articles cited below. As any language, Python naturally also has features that might be drawbacks:

Performance That Python is a scripting language makes it time saving to write and execute short programs, whereas large programs might suffer from a loss in performance. This should not be relevant in introductory programming courses, in which the programs written are relatively small.

No information hiding In Python, attributes are visible by default, thus making discussion on information hiding and encapsulation more difficult. This is not an issue while not dealing with object-oriented programming.

Dynamic typing Dynamic typing was mentioned as an advantage, but it might also turn out to be a drawback. In imperative programming, the possibility of assigning different types to the same variable can be seen as a weakness that might make programs more prone to errors.

These potential drawbacks must naturally be pointed out to the students. Although Python might be a good starting language, our goal was to introduce programming and not Python per se; it was therefore important to make students aware of at least the biggest differences so that they would know what to expect/not to expect from other languages they might use.

Many indications of Python's suitability as the first programming language can be found on the web.⁸ During recent years, Python has also become increasingly popular in HS settings; for instance, the *Python Bibliotheca*,⁹ a site acting as both a repository for learning materials and a virtual meeting place for teachers, includes a list of HSs in the USA that use Python.

Although seemingly on the rise in education, studies on teaching introductory programming using Python are still quite few. When adding the HS perspective to this research, the number of conducted studies decreases further. Some research papers on the use of Python in education (at both university and HS level) can be found (for example (Elkner 2001, Elkner 2002, Ceder & Yergler 2003, Oldham 2005, Stajano 2000, Agarwal & Agarwal 2005, Shannon 2003)), but these have mainly discussed experiences and provided suggestions, without presenting any results. Guzdial (2003) has developed a new type of introductory programming course in which a Java related version of Python (Jython) is used. Nevertheless, although research on Python or

HS programming is by no means new, the combination of these seems to be a field in which much still remains to be explored. The aim of this paper is to begin this exploration.

3 The Introductory Programming Course

Since there is no compulsory programming curriculum for HSs in Finland,¹⁰ we began by defining course goals according to what we feel HS students having taken their first course on programming should be able to do: (1) understand and use basic programming concepts and data structures, (2) develop specifications of problems to be solved, (3) write and test programs, and (4) communicate their work to others by thorough documentation.

Syllabus

According to the Finnish educational system, HS courses should include at least 28 hours of instruction. The course covered the basics of imperative programming: variables, types, boolean logic, I/O, subroutines, flow control, exception handling, modules and documentation. In addition, we began with a rather extensive introduction to algorithms, flowcharts and pseudo code, before moving on to actual programming. This theoretical introduction was considered essential, particularly for students who had no prior experience in CS-related topics. Only after understanding the concept of algorithms can one go on to translating these into programs. We also covered lists and dictionaries (a collection type similar to hashmaps in Java); topics that generally, using other languages, are not included in introductory programming courses.

Teaching Approach

*Active learning*¹¹ builds on *constructivist* principles, according to which students become active participants in their own learning process (Ben-Ari 1998). Instead of viewing learning as passive transmission of information from the teacher to the students, constructivists see learning as an active, reflective process in which the students themselves construct the knowledge by building further upon their prior knowledge.

Moreover, both constructivism and active learning are related to the cone of experience¹² developed by Dale in the 1940s. This model suggests that learners retain more information by what they "do" (90 %) compared to what they "read" (10 %), "hear" (20 %) or "observe" (30 %). Consistently with this model, recent literature on active learning (for example (Ramsden 1992)) suggests that most students cannot learn unless they are actively involved.

As a result of the issues discussed above, we took a "learning by doing"-approach,¹³ and did not give traditional lectures, but instead introduced so called *lab-lessons* in which theory and practice were interweaved during class. The lab-lessons were interactive, hands-on sessions, which took place in a computer lab. The exercises and other activities were *problem-oriented*, engaging the students in both problem-solving and analysis, in addition to writing the actual

¹⁰CS is not included as an individual subject in the curriculum, but is instead integrated in other subjects. Schools also have the opportunity to arrange optional courses in CS and related topics.

¹¹<http://www.ericdigests.org/1992-4/active.htm>.

¹²<http://www.mc.uky.edu/pharmacy/edinnovation/pdf/StepDalesCone.pdf>

¹³The concept of "learning by doing" was introduced by John Dewey in the early 1900s (Dewey 1910).

⁷<http://www.python.org>

⁸<http://www.cs.ubc.ca/wccce/Program03/papers/Toby.html>,
<http://zen.sandiego.edu:8080/luby/papers/python.pdf>,
<http://www.python.org/sigs/edu-sig/miller-dissertation.pdf>,
<http://mcs.wartburg.edu/zelle/python/python-first.html>

⁹<http://www.ibiblio.org/obp/pyBiblio/schools.php>

code. Each lab-lesson started with the teacher formulating a problem, covering topics likely to be relevant and motivating to the students, including simple games and web programming. A solution was developed collaboratively by gradually refining the code. The classroom thus made up a community, in which authentic problems were solved in a "real" context. This can be seen as an implementation of *situated learning*, a theory introduced by Lave and Wegner in the early 1990s, and discussed by Ben-Ari (2004b).

We chose the lab-lesson approach in order to address issues that have emerged from literature: Winslow (1996) states that "[g]ood pedagogy requires the instructor to keep initial facts, models and rules simple, and only expand and refine them as the student gains experience". According to Spohrer and Soloway (1986), "students are not given sufficient instruction in how to 'put the pieces together.'" Lahtinen et al. (2005) have stated that "[t]he more practical and concrete the learning situations and materials are, the more learning takes place". The lab-lessons were designed to make it easier for the students to see the reason for introducing new concepts: why new constructs are needed, and when, and how, these are to be used. Programming concepts and structures were covered in the order needed to be able to solve the problem; not just because they "had to" be introduced according to the order of the chapters in a book. In this manner, new features were learnt one at a time, at the same time as they were combined with previously learnt programming structures using known strategies. This approach of mixing theory and practice is not unique (Haberman & Kolikant 2001), but to our knowledge not common in Finnish HSs. The more traditional way of instruction is to separate the two parts by first covering theory and only then implementing the newly learnt ideas in practice as a separate part of tuition.

Material

A tutorial was written for each problem containing a problem description and theory about the concepts needed to solve the problem, also providing a strategy for solving the problem at hand. Although the course was held in class, all material was made available on a web page in the course framework *Moodle*.¹⁴ The course page acted as a repository for course material, examples, assignments, announcements and links. The students were to document 1) what they had experienced as most difficult and 2) what they had learnt after each lab-lesson in their private online learning diary (called *progress report* since we wanted to use a more neutral term that would not have direct associations with "traditional" diaries). The purpose was to make it easier for us to see when students had problems; in our experience, HS students are seldom willing to express their uncertainties or admit their difficulties in front of their classmates.

Examination

The final grade was based on the level of participation in the lab-lessons, a final project (including documentation) and a two-part exam. The final project was a larger assignment that the students completed at the end of the course. Since we wanted to meet the individual needs and interests of the students, they were allowed to choose the topic for the project quite freely. The students were allowed to complete the project either individually or in pairs. In cases where the students collaborated, documentation had to be precise, clearly stating who had done what. The final exam

consisted of one part to be completed using only pen and paper and another, in which the students were to write the solution programs on the computer.

4 Study Methodology

Action Research

Our research is loosely built on the principles of *action research* (Clear 2004). In action research, practitioners in a field improve practice by doing or changing something and reflecting on the results. The improvement can be in three areas: "improving a practice; improving the understanding of a practice [...] and improving the situation in which the practice takes place" (p. 106). The main purpose is to collect data and experience that help in gaining a better understanding of the practice. Our study follows a practical action research model, in which "the researcher facilitates reflection by individual practitioners upon some aspect of their practice" (p. 109). In our case, the researchers and the practitioners were the same persons.

Settings

The introductory course presented in the previous section was given four times as an optional HS course during the school year 2004/2005 (four different student groups in two different schools). In total, 42 boys aged 16-19 have completed the course; unfortunately the course did not attract any female students. The students in our study constituted nearly 40% of all HS students taking a basic programming course in Turku (the center of the third largest metropolitan area in Finland) in 2004/2005.

When teaching programming, one is bound to be faced with a situation in which the students are on different levels; some students might have programmed a lot, whereas others might not have any programming background at all. There were also differences between the groups: all students in groups A-C had taken at least one basic CS course, our programming course was part of their curriculum, and they met regularly thrice a week during a period of 1½ months. The students in group D had not had any previous CS courses, and due to school arrangements, they met only once a week for three hours in the late afternoon, making the course span a period of three months.

Data Collection

The learning results were mainly derived by analyzing the progress reports, the final projects and the exam. In order to gather additional information we conducted two surveys during each course. The aim of the pre-course survey was to collect background data of the students, such as information about age, programming background and mathematical skills. More general questions, such as concerning the benefits experienced from learning to program and the students' awareness of Python, were also included. The purpose of the post-course survey was to gather information about the students' attitudes towards the course, its difficulty level and Python as the language of instruction. Students who had some pre-course programming background were also asked to compare Python with the languages used earlier.

5 Results and Discussion

In this section we present and discuss the learning results and the data extracted from the surveys. We also discuss our experiences from teaching Python

¹⁴<http://www.moodle.org>

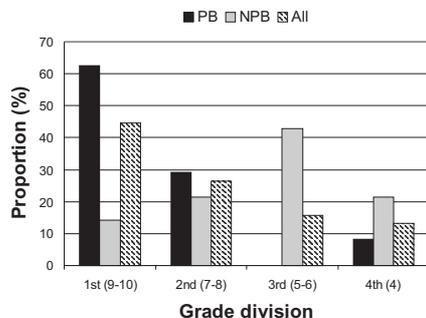


Figure 1: The distribution (in percent) of students in divisions based on whether they had programmed before (PB) or not (NPB).

compared to prior experiences using other languages such as C, C++ and Java.

5.1 Learning Results

Course Grades

The course grade was given on the scale 4-10 (4 = failed), and was derived from the different parts of the examination mentioned in section 3. When calculated based on the data from all four student groups, the average course grade was 7.71. Over 85 % of the students passed the course. The grades were divided into four divisions (1st (9-10), 2nd (7-8), 3rd (5-6) and 4th (4)), which resulted in a distribution following a reversed Gaussian curve; this finding is well in line with our experiences from university level programming courses. Our data indicate that the results for the students who had some programming background were good: nearly 90 % achieved the grade 8 or higher. Although the diagram in figure 1 does not provide any comparable data, since this was the first time we gave these courses, it does show that nearly 80 % of students who had not programmed before passed the course. For students who achieved one of the lowest grades, the course exam was the weakest point of the examination parts. This calls for a discussion on how to prepare students for written exams: should tuition perhaps include elements of writing programs using only pen and paper? This would enforce the students to really think their programs through and trace the execution, test and debug the code - all by hand.

The number of students that failed the course was somewhat higher than for HS courses in general, but completely normal for programming courses, and therefore an expected result. The number of students in the first division was, however, larger than what can be seen as normal for any HS course (especially for programming courses): almost 45 % of all students passed with one of the two highest grades.

Since this was the first time a programming course was given at the school of group D, we did not have any data to compare the results of this group with. However, programming courses have been given at the school of the other groups (A-C), most recently using Java. For these courses, all other factors but the language have been the same as for groups A-C in our course (at least one background CS course, same teacher, same course requirements, scheduled and frequent meetings). When comparing the results, the differences were notable. As figure 2 shows, the proportion of students that failed decreased using Python, whereas the number of students achieving one of the two highest grades increased: half of the

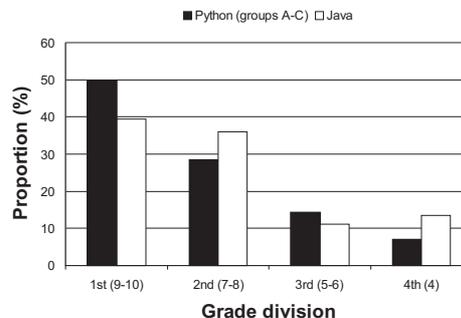


Figure 2: Comparison of the grade distribution (in percent) from introductory programming courses using Python and Java.

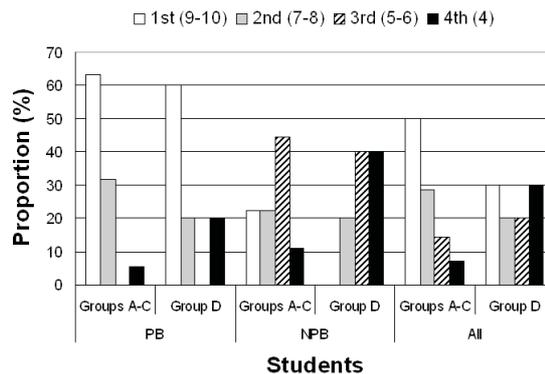


Figure 3: The distribution (in percent) of grades achieved by students in different groups with regard to whether they had programmed before (PB) or not (NPB).

students in groups A-C achieved a grade in the first division. The reliability of this result might naturally also depend on other variables, such as motivation level, prior programming experience and mathematical skills, of the students. However, given that the prerequisites and the practical arrangements of the courses were identical, as was the teacher, the results should be comparable.

As expected, there were differences between the grades achieved by students in groups A-C and those in group D. Figure 3 shows that the overall result for the students in groups A-C was better than for those in group D. The biggest difference was among students who had not programmed prior to the course; whereas only 11 % of the novices in groups A-C failed, 40 % of those in group D failed the course. On the other hand the percentage of students achieving a high grade is just as good for students with some programming background in both groups A-C and group D. Our findings thus suggest that the differences in the environment (for example meeting rarely late in the afternoon) and the lack of previous CS-courses, complicated the learning process among true novices. Moreover, not meeting more frequently resulted in the students finding it difficult to remember the syntax and programming in general. Considering, in addition, that over 75 % of all students stated that they programmed most in class, one can conclude that the students in group D were not exposed to programming as much as would have been needed. However, when starting on the final project, most students also worked at home. Since the students enjoyed working on the project, it could be beneficial to let the students work on smaller projects throughout the course to ensure that they program continuously.

Progress Reports

We were pleasantly surprised to see that most students took the progress report seriously. Naturally, there were also non-informative comments, but most students filled out the report regularly, stating what they had learnt and what they had experienced as most difficult. The analysis of the reports showed that students had in general been positive. Coding itself was frequently mentioned as the most enjoyable part of the lab-lessons. No difficulties were mentioned at the beginning of the course, whereas some students started to report difficulties when dealing with sub-routines and exceptions.

An interesting observation was that the weaker students more often wrote comments such as "rather easy", "did not sound too difficult", "it will be OK when I practice", whereas the more advanced students were able to point out the difficulties more precisely if they had any. One explanation for this could be that weaker students do not dare admit their difficulties, but it might also be that they do not recognize which things are problematic. This could imply that teachers should try more actively to make students aware of their difficulties, for example by arranging more mid term quizzes and discussing the results - perhaps even in private.

Characteristics of Student Programs

We are currently in the process of analyzing the students' final projects, and we will later report on our findings from this analysis elsewhere. Our preliminary results indicate that the programs have been of at least the same standard as during similar HS-courses given using other languages. Most final projects include different control structures, self-made and built-in subroutines, modules as well as some documentation. Several projects can be ranked as particularly good considering both quality and complexity: programs that consist of several modules interacting at runtime, providing user guides and thorough documentation including testing and debugging reports.

Code listing 2 *for-loops in Python*

```
Python          Java
for element in set:  for (i = 0; i < set.length; i++) {
    print element    System.out.println(set[i]);
                    }
}
```

Some interesting findings have been made already at this point of our analysis. First, the "off-by-one bugs" (Spohrer & Soloway 1986) seem to be less frequent in students' programs using Python compared to our previous experiences in, for instance, Java. One evident explanation can be found in the differing format of the for-loop in Python. When writing for-loops that are truly "Pythonish", no indexes are needed (code listing 2). Whereas the for-loop in Java can be seen as quite cryptic, the Python for-loop might remain unclear due to the missing indexes: the syntax does not give an intuitive understanding for what the computer does and how many times it repeats it. The avoidance of indexes naturally cleans up and simplifies the code, but the Python version might even be too simplified. In this sense, the for-loop of Pascal can be seen as a good compromise: simplified, but still containing indexes, thus forcing the students to get used to "thinking" in terms of the computer and its memory. However, the lack of indexes can also be seen as an advantage: when teaching programming as an all-round learning skill in HSs (to students, of which by no means all will become programmers), the

most important thing is for students to learn how to solve problems and translate solutions into program code. The question is whether indexes play an important role in this. If the answer is yes, the Python for-loop can indeed be regarded as too simple; otherwise, its simplicity is an additional feature due to which Python can be found suitable for novice programmers.

Another interesting issue concerns the use of I/O. When teaching HS programming using Java and C++, our students have frequently found user input difficult. Difficulties related to input statements have also been reported on by several researchers (for example (Bayman & Mayer 1983, Haberman & Kolikant 2001, Rosenberg & Kölling 1996)). In Python, no input or output streams have to be opened or closed. Getting input from the user is accomplished by using simple built-in functions (code listing 3), and the students learnt to use I/O effectively in an hour. The same comments as for the for-loop can be made concerning I/O as well; while Python's solution is compact and simple, it can be discussed whether it has been made even trivial.

Code listing 3 *I/O in Python*

```
# Prints out a question and reads user input as a string
name = raw_input("What is your name? ")
```

```
# Prints out a question and reads user input as a numeric
age = input("How old are you? ")
```

5.2 Survey Results

Difficulty Level

The data from the post-course survey showed that the course was not considered particularly difficult. The students graded the difficulty level of the course on a five-point scale (1 = very easy, 5 = very difficult), giving a mean of 2.58 (std dev = 1.03).

Students rated the difficulty level of each course topic on the same scale as the course. The data (table 1) showed that no topic had been regarded as particularly difficult, the averages lying between 1.33 and 3.11 on the scale 1-5 (1 = very easy, 5 = very difficult). Subroutines, exception handling and documentation were considered the most difficult topics. Open questions established this finding: whereas variables and control structures were considered rather simple, abstract topics such as algorithms, exception handling, documentation and subroutines were regarded as more difficult.

Topic	Mean	Std Dev
Algorithms	2.68	0.87
Variables	1.72	0.97
Lists	2.51	1.21
Boolean expressions	2.32	1.04
For-loops	2.32	0.93
While-loops	2.05	0.94
If-statements	1.33	0.74
Subroutines	2.76	1.08
Modules	2.44	1.02
Dictionaries	2.51	1.07
Files	2.69	1.15
Exception handling	2.89	1.09
Documentation	3.11	0.85

Table 1: *The perceived difficulty level of course topics (1 = very easy, 5 = very difficult)*

Difficulties in Abstract Topics

Subroutines are known to be difficult among novices, for example due to parameter passing (Fleury 1991). Another plausible explanation is that it might be difficult to understand why subroutines are needed when writing short programs. The same goes for exception handling: students might not grasp why such a thing is needed. Finding out what might go wrong in one's programs, and recognizing the parts of the code in which errors could occur, can be difficult. Exception handling has traditionally not been seen as one of the first things to cover in programming courses, resulting in it being introduced rather late in the schedule. This might make it even more difficult for the students to understand why it is suddenly needed; after all, their programs have "worked" before (provided that the user has given the correct input).

As for the difficulties experienced with documentation, researchers have suggested different reasons: students find writing comments and documentation useless for small programs, and is also considered to slow down the programming process (Deveaux et al. 1999). Moreover, Weinberg (1998) stated that "the good documenter has to be a good programmer" (p. 169). Clearly, not all students having taken their first programming course can be regarded as "good programmers" and explaining one's programs, ideas and logic in one's own words is completely different from writing code. In our case, the fact that the students were young might be an additional reason: students of this age are simply not used to writing thorough documentation, and have, most likely, not yet developed the necessary writing skills.

Many of our findings are supported by previous studies: in a study made by Haataja et al. (2001) among students in a web-based introductory programming course in Java, 60 % of the respondents stated that methods (in other words, subroutines in Java) were one of the most difficult topics. According to a study conducted by Lahtinen et al. (2005), students find error handling (exception handling) and libraries (modules) more difficult than, for example, selection and looping structures when learning to program in Java or C++.

Although many of our findings are supported by results from earlier studies, some differences can be found: compared to the results reported by Lahtinen et al. (2005), variables and selection structures were considered easier by our students (average difficulty of 1.72 and 1.33 vs. 2.10 and 1.98). Loop structures were considered to be of similar difficulty level in both studies. Moreover, our students found the if-statement easy, whereas 48% of the respondents to the study of Haataja et al. (2001) perceived if-statements as one of the most difficult topics. It thus seems that although some of the difficulties are the same across studies, there are differences as well, which, in addition, can be contradictory. These differences can naturally be due to various factors, the differing programming language being only one. More research comparing students' attitudes towards the different topics in introductory programming courses would be needed to make clearer the potential impact of the language.

Petre et al. (2003) have found indications of topics being introduced early in a course to be generally perceived as "easy" by students, whereas later topics are usually considered more difficult. They found two possible explanations for this: 1) students have more time to truly learn the early topics and 2) the early topics are emphasized more by the teacher. In this light, our findings and the results from previous afore-mentioned studies suggest that the "difficult" topics would benefit from being introduced ear-

lier in the syllabus. For instance, exception handling could be introduced when discussing user input, to let the students get used to including it in their programs from the beginning. Later on in the course, we find it imperative that the students are made aware of the different types of exceptions that can occur and learn how to look for parts in the code that could be prone to errors. This could potentially result in students paying more attention to debugging and trying to write correct programs. Documentation could also be emphasized more in the very beginning of the course, in order for the students to get used to describing the goals and the process of their programs. This can be done for small programs, but takes away some time from actual programming. One must then consider the benefits of introducing these "difficult" topics earlier - is it worthwhile to rethink the way in which we teach programming if we can give students better opportunities to learn the difficult topics? We are tempted to say "yes", or as one student stated in the post-course survey: "Is it truly necessary to spend so much time on printing in the beginning of the course? The simple issues will still be practiced throughout the rest of the course[...]". It would be interesting to see if the results were to be different if the "simple" topics were covered quickly, and the ones that have commonly been viewed as "difficult" were given more time.

Attitudes towards Python

According to the pre-course survey data, almost half of the students had never heard of Python before. About 65 % of the students mentioned a language they would prefer to use, the majority mentioning C++, C or Java.

The students who had prior experience in programming (60 %) were asked to compare the languages used previously to Python. We used a five-point scale (1 = totally disagree, 5 = totally agree) and the results were in Python's favor: compared to other languages, Python was perceived as easier to learn and also as somewhat more pleasant to use. The students strongly agreed with the statement "Python was easier to learn [than languages previously used]" (mean of 4.1) and they also agreed with the statement "Python was more fun to use" (mean of 3.7). Naturally, new languages might be considered "easier" when one has already learnt to program in another one. However, the fact that Python was regarded as at least somewhat more fun to use is a positive finding. According to the students, aspects such as simplicity, compactness (resulting in shorter code), clarity and a rich range of modules made Python better than other languages; that is, the kind of features that initially suggested that Python would be suited as a first language. The other languages were, on the other hand, perceived as more flexible and "hard-core", enabling programming on a lower level. However, the course being an introductory one, no "hard-core" details could (or even should) be covered.

Nearly 80 % of the students that filled out the post-course questionnaire planned to continue programming, and over 65 % of these stated that they would do so using Python. Other languages mentioned were C++, C, Java and PHP. Half of the students who had programmed prior to the course (in some other language than Python) reported that they would continue programming using Python after taking this course. This can be seen as an interesting result, which could imply that Python was considered better than the languages used previously; one could expect that one does not lightly change one's preferences concerning what language to use.

Student Satisfaction

The post-course survey showed that the course had fulfilled the expectations of most students (90 %). Over half of the students had learnt exactly as much as they had expected in the course, whereas 23 % felt they had learnt more and 20 % felt they had learnt less than expected. The students found writing code, working on the final project and "getting a program to work" most satisfying. The lack of graphical programming was the main reason why the expectations had not been fulfilled. Considering again the fact that the students were young, one can assume that some did not have any expectations at all, and some had expectations focusing on technical and "cool" details.

Collaborative or Single Programming

The results from the post-course survey did not indicate any preference to work alone or with a friend. In groups A-C, all students worked individually on the project, whereas most students in group D worked in pairs. One explanation for most students working alone could be the additional requirements on documentation, since, as stated earlier, the students found documentation difficult. We believe that collaborative programming could facilitate producing documentation: when working together with someone else, one is bound to express thoughts and ideas to the other(s), that is, exactly the same things that one does when documenting programs. In the future, we will therefore consider guiding the students into working in pairs or groups.

5.3 Our Experiences

Powerful Built-in Constructs

Compared to for instance teaching Java, it was possible to introduce the powerful list data type at an early stage, thus enabling writing more advanced programs. The students did not use the same variables for different data types, although this is allowed, and dynamic typing did, thus, not turn out to be a drawback as was presumed in section 3.

Ease of Debugging and Error Detection

We had expected that Python's easy and clear syntax would facilitate students' learning, but we had not considered the consequences for us as instructors. The indented structure of Python programs forced the students to write structured programs, since incorrect indentation results in syntax errors. The clear indentation facilitated debugging, error detection and made correcting students' programs easier than, for instance, in Java or other languages where program blocks are denoted by curly brackets. In such languages, the compiler does not put any requirements on the structure of the program; one can even write an entire program on one single line, provided that all semicolons and brackets are in the right places. Programs of this kind are nearly impossible to check. However, such horror code cannot be written in Python. We feel that writing structured programs, which are easy to check, follow and maintain is one of the main lessons in any programming course. Using Python, this lesson is taught automatically.

Interactive Mode

The interactive mode was useful when introducing new constructs and showing examples. Since each concept could be demonstrated in isolation, no other

syntax "got in the way". In addition, the students frequently used the interactive mode for checking name spaces and reading documentation.

Ease of Re-Use

The core of a programming language should not be overly extensive (Weinberg 1998): all necessary constructs and concepts should be compact. To be truly useful, the language should, however, be extendable. Most of the languages today can be extended by individual programmers, who are also given the possibility to re-use code written by others or themselves. However, in Python the advantages of re-use are even more evident, since all programs automatically become modules that can be used in other programs. Our students were able to share their programs with others and re-use programs written earlier already during the first parts of the course. The students recognized the importance of also writing small and seemingly unusable programs, by seeing how these are used when building more complex software.

Our suspicions of the negative impact of the lack of information hiding proved to be without grounds. Even though the students found writing documentation difficult, they learnt to efficiently read and make use of formal documentation, and were able to use ready-made subroutines correctly without knowing what was going on "behind the scenes". This made it possible for us to discuss abstraction and information hiding, which suggests that the students managed to develop some level of abstract thinking skills in this short time - even though object orientation was not included in the syllabus.

Code listing 4 *Use of the webbrowser module*

```
import webbrowser

options = {"A" : "http://www.google.com",
          "B" : "http://www.yahoo.com",
          "C" : "http://www.altavista.com"
          }

for site in options:
    print site + " : " + options[site]

try:
    choice = raw_input("\nChoose a site: ")
    webbrowser.open(options[choice])

except:
    print "You did not pick a valid alternative."
```

In addition, using modules that appealed to the students made it possible to teach the basics of programming in a motivating way. Since the modules are easy to use, and do not enforce notational overhead, these did not take the students' attention away from the programming itself. In our experience, when trying to incorporate external packages or modules in other languages, the extra notation usually turns the students' focus to learning the specific package instead. In Python, the modules and the functionality they provide can really be used as what they should be, in other words, as tools, without becoming the focus of attention. One example is the *open*-function of the *webbrowser*-module, which can be used to open web pages using a browser. The sample program in code listing 4 illustrates a program that displays a menu of search engines of which the user chooses one by simply entering the corresponding menu letter, whereas the web page is opened using the default browser. As the code illustrates, this lets the students practice important concepts such as user input,

iteration, dictionaries and exception handling in an interesting way, without extra notation. In our opinion, modules truly supported teaching and we feel that this ease of making tuition interesting is a great advantage for a language used by novices.

6 Summary

When choosing a programming language for educational settings, one has to consider many different factors: although learning aspects should be most important, institutions, pressured by for instance industry, usually feel a responsibility to select a language with market appeal. The competition is heavy, and individual institutions do not dare to make radical changes or try something new - even if they wanted to; the risk of losing students to a competitor is too big. Python is not one of the most widely used languages today, but statistics indicate that it is among the top ten languages.¹⁵ But should this even be an issue at HS level? This is again a question of what we want our students to learn. If we focus on a certain language, we are most likely primarily teaching the language and its syntax, the core concepts of programming becoming a secondary objective. Should it not be the other way around? Should we not focus on teaching programming skills and leaving the language to be something of subordinate importance? The fact still remains - we need a language when teaching programming, and the challenge is to find the most suitable one. In this paper we have presented a case study of using Python to introduce programming in HSs, with emphasis on programming, not Python, and we will now summarize our main findings.

Introducing Programming in High School...

Our intention was to introduce programming as an all-round learning skill at HS level, and to some extent we succeeded. The students experienced feelings of success, learnt to write structured programs and think in terms of re-use and interfaces; a highly needed and appreciated skill in today's society.

Our findings indicate that students find it difficult to deal with abstract concepts, regardless of language. Naturally, the easier the syntax, the better, but in order to truly understand, one cannot focus on pure syntax or technical details; these can be looked up in a book and copied into one's programs. However, using abstract concepts properly and in the correct place of the code is nearly impossible without understanding the underlying principles of why these concepts are needed and in which situations. Abstract topics should thus be given more time in the syllabus, even early on in the course.

Another source of difficulty was related to the young age of the students: HS students have not necessarily yet developed good writing skills required, for example, for producing proper documentation. A final issue is that of attracting girls into CS: so far, all students have been boys, and considering the role of HSs as all-round learning institutions, this is alarming. We tried to build a programming course that would be suited for everybody, but were not able to attract any female student - even though both teachers were women. A future challenge is therefore, as is probably the case for all CS faculty, trying to find ways to make programming more appealing to girls.

...Using Python

Python's simple and flexible syntax significantly reduced notational overhead compared to other lan-

guages we have used in instruction, and thus left more time for actual coding. In addition, the interactive environment offers immediate feedback even with limited language experience, and lends itself to in-class coding, testing and demonstration. This interaction also supports active, hands-on learning, since the students can easily try out and analyze different solutions. We identify the advantages of using compiled system languages such as Java and C++ at higher levels of education and in industry, but we do believe that one should remember that there is quite a difference between teaching professionals or experienced programmers, and high school students, who are about to take on programming for the first time. The main goals of programming education at HS level are (1) to teach the basics of programming and algorithmic thinking as general all-round skills, and (2) to prepare the students for future studies.

Moreover, the ease of re-use has many benefits: the easy-to-use modules makes it possible for the students to practice basic concepts in a motivating manner, and the students learn how smaller programs are crucial when building larger systems. Although the final product might be very complex, the building process must be easy; using Python this is something that seems possible to achieve. Python is an open-source language, and this openness further increases its suitability for teaching purposes. Books, course readings, syllabi, exercises and other material can be downloaded from the Internet free of charge, and news groups and mailing lists provide forums for discussions and questions. The international community may also have welcome side effects: reading material in foreign languages and exchanging ideas with persons with the same interests in other countries develop language skills and promote understanding for other cultures. In some cases, Python's syntax can be seen as perhaps even too simplified, but as a whole, our initial experiences indicate that Python could be suitable as a language for novices.

7 Future Work

It is always difficult to judge the effects of curricular changes objectively, and the case study presented in this paper is too small for giving any significant statistical results; that was, however, not our original goal either. We set out to investigate programming at HS level and the suitability of Python in this process. Our experience has been positive, but to be able to make more conclusive statements, we are actively continuing our work in the field of using Python in HS programming courses.

Since the four courses presented in this paper, we have given a HS continuation course in programming covering object-oriented topics and graphics using Python. As we speak, we are also conducting a new study, similar to the one presented here. Some changes have been made, based on the results from this first study, for example by rethinking the syllabus in order to introduce abstract topics earlier. We will also continue the analysis of the student projects from the courses given so far. Finally, some of the students that have participated in our Python courses are now studying Java, and we are following up on their progress.

References

- Agarwal, K. K. & Agarwal, A. (2005), 'Python for CS1 CS2 and Beyond', *J. Comput. Small Coll.* **20**(4), 262-270.
- Aiken, R. M. (1972), Experiences and observations on teaching computer programming and simulation concepts to high school students, in 'SIGCSE '72: Proceedings of the 2nd SIGCSE technical symposium on CS Education', ACM Press, pp. 67-71.

¹⁵ <http://www.tiobe.com/tpci.htm>

- Anthony Robins, Janet Rountree, N. R. (2003), 'Learning and teaching programming: A review and discussion', *Computer Science Education* **13**(2), 137-172.
- Bayman, P. & Mayer, R. E. (1983), 'A diagnosis of beginning programmers' misconceptions of basic programming statements', *Commun. ACM* **26**(9), 677-679.
- Ben-Ari, M. (1998), Constructivism in Computer Science Education, in 'Proceedings of the 29th SIGCSE technical symposium on CS education', Atlanta, Georgia, United States, ACM Press, pp. 257-261.
- Ben-Ari, M. (2004a), 'Computer Science Education in High School', *Computer Science Education* **14**(1), 1-2.
- Ben-Ari, M. (2004b), 'Situated Learning in Computer Science Education', *Computer Science Education* **14**(2), 85-100.
- Biddle, R. & Tempero, E. (1998), 'Java pitfalls for beginners', *SIGCSE Bull.* **30**(2), 48-52.
- Ceder, V. & Yergler, N. (2003), Teaching Programming with Python and Pygame, in 'PyCon 2003'.
- Clark, D., MacNish, C. & Royle, G. F. (1998), Java as a teaching language - opportunities, pitfalls and solutions, in 'ACSE '98: Proceedings of the 3rd Australasian conference on CS education', The University of Queensland, Australia, ACM Press, pp. 173-179.
- Clear, T. (2004), *Computer Science Education Research*, Taylor and Francis Group, chapter 2, Critical Enquiry in Computer Science Education, pp. 101-125.
- Close, R., Kopec, D. & Aman, J. (2000), Csl: perspectives on programming languages and the breadth-first approach, in 'CCSC '00: Proceedings of the 5th annual CCSC north-eastern conference on The journal of computing in small colleges', New Jersey, United States. Consortium for Computing Sciences in Colleges, pp. 228-234.
- de Raadt, M., Watson, R. & Toleman, M. (2002), Language Trends in Introductory Programming Courses, in 'Informing Science and IT Education Conference', pp. 329-337.
- Deveaux, D., Fleurquin, R. & Frison, P. (1999), Software engineering teaching: a "docware" approach, in 'ITiCSE '99: Proceedings of the 4th annual ITiCSE conference', Cracow, Poland, ACM Press, pp. 163-166.
- Dewey, J. (1910), 'How We Think'. Retrieved June 21, 2005 from <http://spartan.ac.brocku.ca/~lward/Dewey/Documents.html>.
- Elkner, J. (2001), Using Python in a High School Computer Science Program, in '9th International Python Conference'.
- Elkner, J. (2002), Using Python in a High School Computer Science Program - Year 2, in '10th International Python Conference'.
- Fleury, A. E. (1991), Parameter passing: the rules the students construct, in 'SIGCSE '91: Proceedings of the 22nd SIGCSE technical symposium on CS education', San Antonio, Texas, United States, ACM Press, pp. 283-286.
- Gal-Ezer, J., Beeri, C., Harel, D. & Yeduhai, A. (1995), 'A high-school program in computer science.', *Computer* **28**(10), 73-80.
- Ginat, D. (2003), The novice programmers' syndrome of design-by-keyword, in 'ITiCSE '03: Proceedings of the 8th annual ITiCSE conference', Thessaloniki, Greece, ACM Press, pp. 154-157.
- Guzdial, M. (2003), A media computation course for non-majors, in 'ITiCSE '03: Proceedings of the 8th annual ITiCSE conference', Thessaloniki, Greece, ACM Press, pp. 104-108.
- Haataja, A., Suhonen, J., Sutinen, E. & Torvinen, S. (2001), 'High School Students Learning Computer Science over the Web', *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning* **3**(2).
- Haberman, B. & Kolikant, Y. B.-D. (2001), Activating "black boxes" instead of opening "zipper" - a method of teaching novices basic cs concepts, in 'ITiCSE '01: Proceedings of the 6th annual ITiCSE conference', Canterbury, United Kingdom, ACM Press, pp. 41-44.
- Hadjerrouit, S. (1998), 'Java as first programming language: a critical evaluation', *SIGCSE Bull.* **30**(2), 43-47.
- Hagan, D. & Markham, S. (2000), Does it help to have some programming experience before beginning a computing degree program?, in 'ITiCSE '00: Proceedings of the 5th annual ITiCSE conference', Helsinki, Finland, ACM Press, pp. 25-28.
- Halasz, F. & Moran, T. P. (1982), Analogy considered harmful, in 'Proceedings of the 1982 conference on Human factors in computing systems', Gaithersburg, Maryland, United States, ACM Press, pp. 383-386.
- Lahtinen, E., Ala-Mutka, K. & Järvinen, H.-M. (2005), A study of the difficulties of novice programmers, in 'ITiCSE '05: Proceedings of the 10th annual ITiCSE conference', Capriccia, Portugal, ACM Press, pp. 14-18.
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004), A Multi-National Study of Reading and Tracing Skills in Novice Programmers, in 'ITiCSE-WGR '04: Working group reports from the ITiCSE conference', Leeds, United Kingdom, ACM Press, pp. 119-150.
- Mayer, R. E., Dyck, J. L. & Vilberg, W. (1986), 'Learning to program and learning to think: what's the connection?', *Commun. ACM* **29**(7), 605-610.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001), 'A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students', *SIGCSE Bull.* **33**(4), 125-180.
- Merritt, S. M., Bruen, C. J., East, J. P., Grantham, D., Rice, C., Proulx, V. K., Segal, G. & Wolf, C. E. (1993), 'Acm model high school computer science curriculum', *Commun. ACM* **36**(5), 87-90.
- Milbrandt, G. (1993), 'Using problem solving to teach a programming language in computer studies', *Journal of Computer Science Education* **8**(2), 14-19.
- Mody, R. P. (1991), 'C in education and software engineering', *SIGCSE Bull.* **23**(3), 45-56.
- Oldham, J. D. (2005), 'What Happens after Python in CS1?', *J. Comput. Small Coll.* **20**(6), 7-13.
- Palumbo, D. B. (1990), 'Programming Language/Problem-Solving Research: a Review of Relevant Issues', *Review of Educational Research* **60**(1), 65-89.
- Petre, M., Fincher, S. & et al., J. T. (2003), My criterion is: Is it a boolean?: A card-sort elicitation of students' knowledge of programming constructs., Technical Report 6-03, Computing Laboratory, University of Kent, Canterbury, Kent, UK.
- Ramsden, P. (1992), *Learning to Teach in Higher Education*, London: Routledge.
- Rosenberg, J. & Kölling, M. (1996), I/o considered harmful (at least for the first few weeks), in 'ACSE '97: Proceedings of the 2nd Australasian conference on CS education', The Univ. of Melbourne, Australia, ACM Press, pp. 216-223.
- Schollmeyer, M. (1996), Computer programming in high school vs. college, in 'SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on CS education', Philadelphia, Pennsylvania, United States, ACM Press, pp. 378-382.
- Shaffer, D. (1986), 'The use of logo in an introductory computer science course', *SIGCSE Bull.* **18**(4), 28-31.
- Shannon, C. (2003), Another breadth-first approach to cs i using python, in 'SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on CS education', Reno, Nevada, USA, ACM Press, pp. 248-251.
- Soloway, E. (1993), 'Should we teach students to program?', *Commun. ACM* **36**(10), 21-24.
- Spohrer, J. C. & Soloway, E. (1986), 'Novice mistakes: are the folk wisdoms correct?', *Commun. ACM* **29**(7), 624-632.
- Stajano, F. (2000), Python in Education: Raising a Generation of Native Speakers, in 'Proceedings of the 8th International Python Conference'.
- Stephenson, C. & West, T. (1998), 'Language choice and key concepts in cs 1', *Journal of Research on Computing Education* **31**(1), 89-95.
- Treese, W. (2003), 'Programming literacy: is it for everyone?', *netWorker* **7**(2), 15-17.
- van Rossum, G. (1999), *Computer Programming for Everybody*. CNRI: Corporation for National Research Initiatives, 1999.
- Weinberg, G. M. (1998), *The Psychology of Computer Programming*, Dorset House Publishing.
- Winslow, L. E. (1996), 'Programming Pedagogy - a Psychological Overview', *SIGCSE Bull.* **28**(3), 17-22.

Paper VI

What About a Simple Language? Analyzing the Difficulties in Learning to Program

L. Mannila, M. Peltomäki & T. Salakoski

Originally published in *Computer Science Education*, 16(3), pp. 211-228. Routledge, 2006.

What About a Simple Language? Analyzing the Difficulties in Learning to Program

Linda Mannila*, Mia Peltomäki and Tapio Salakoski
Turku Centre for Computer Science, Finland

In this paper, we present the results from a two-part study. We analyze 60 programs written by novice programmers aged 16–19 after their first programming course, in either Java or Python. The aim is to find difficulties independent of the language used, and such originating from the language. Second, we analyze the transition from a “simple” language to a more “advanced” one, by following up on eight students, who learned programming in Python before moving on to Java.

Our results suggest that a simple language gives rise to fewer syntax errors as well as logic errors. The qualitative part of our study did not reveal any disadvantages from having learned to program in a simple language when moving on to a more complex one. This suggests that not only can a simple language be used when introducing programming as a general skill, but also when providing basic skills to future professionals in the field.

1. Introduction

Novice Programming

Programming has traditionally been considered a difficult topic, and research has shown that novices face several problems when learning to program (Spohrer and Soloway, 1986; Lister et al., 2004; McCracken et al., 2001; Robins, Rountree, & Rountree, 2003). Winslow (1996) refers to Linn and Dalbey, who have suggested an “ideal chain” for learning to program: (1) learn one language feature at a time (both syntax and semantics), (2) learn to combine the newly learned feature with previously known skills, and (3) develop general problem-solving skills. When learning to program, novices are not only faced with solving a problem; they also have to learn the syntax and semantics of a programming language and how to express solutions in a form that the computer can understand. Although the language is not the main focus, it still plays a substantial role in the learning process.

*Corresponding author. Turku Centre for Computer Science, Joukahaisenkatu 3–5, FIN 20520 Turku, Finland. E-mail: linda.mannila@abo.fi

Programming Languages

Most recent results related to difficulties in programming are based on experiences from using languages such as Java and C++. This is understandable, since these are the languages most popular both in industry¹ and at universities (de Raadt, Watson, & Toleman, 2002; Stephenson & West, 1998). Despite their popularity, there has been much debate about their suitability in education, especially when introducing programming to novices (e.g. Hadjerrouit, 1998; Biddle & Tempero, 1998; Close, Kopec, & Toleman, 2000).

Only few programming languages have been designed for teaching. Such a language should, for instance, have a simple syntax, be easy to learn, powerful, structured in design and universal in use (Milbrandt, 1993). Pascal and Logo were designed with education in mind, fulfilling many of these characteristics, and studies have indicated that they are suitable in education (e.g. Schollmeyer, 1996; Shaffer, 1986). Unfortunately, both languages suffer from drawbacks that have led to decreasing significance over the years. In the late 1990s, Guido van Rossum proposed that the high-level scripting language Python could be used in education; he has even suggested that everybody could master programming using this language (van Rossum, 1999). Many indications of Python's suitability as the first programming language can be found in the literature and on the Internet (see Appendix 1).

Motivation

We have been teaching programming courses at high school² (HS) for many years using different languages: Pascal, C++, Delphi, Java and most recently Python. (Grandell, Peltomäki, Back, & Salakoski, 2006). The main criticism against introducing programming using a "non-traditional" language, such as Python, has been that students are taught too simple a language, which makes them run into a wall when having to deal with a more complex one later on. Our main argument against this is recognizing the difference between teaching future professionals and teaching HS students. Using a scripting language with a simple syntax might be out of the question in some contexts. However, the main goals of programming education at HS level are (1) to teach the basics of programming and algorithmic thinking as general, all-round skills, and (2) to prepare the students for future studies. Other contexts should not decide how these goals are fulfilled.

The aim of this paper is to evaluate the use of a "simple" language as the first language, and the effect such a decision might have on future learning of more complex languages. We have conducted a two-part study: first, we have analyzed 60 programs written by novice programmers at HS level, half using Java and the other half using Python. We have studied errors found in the programs and made comparisons in order to see whether any difficulty could be attributed to the specific language used. In the second part, we have followed up on eight of the Python students' progress as they have moved on to learning Java at university level, in order to examine the transition process from a simple language to a more "advanced" one. We have thus evaluated

Python and Java both in parallel as first programming languages, and as subsequent languages. We begin by describing the study and the methods used, after which we present and discuss the results. The paper is concluded with a section giving some final remarks and suggestions for future work.

2. The Study

Our study is built on an *action research* (Clear, 2004) approach. In action research, practitioners aim at improving practice by doing something or making changes and then reflecting on the results. The improvement can come in three forms: “improving a practice; improving the understanding of a practice [...] and improving the situation in which the practice takes place” (p. 106). The main aim of this study is to collect data and experience that help in gaining a better understanding of the practice.

2.1. Part 1: Program Analysis and Comparison

In the first part, we analyze the structure and characteristics of programs written on the exam by HS students having taken their first programming course in Java (2002/2003) or Python (2004/2005). The only difference in the preconditions was the programming language used; all other aspects, e.g. teacher, course environment and contents (basics of imperative programming), were the same during both years. The course goals were also unchanged: after completing either course, the students should have been able to (1) understand and use basic programming concepts and data structures, (2) develop problem specifications, (3) write and test programs, and (4) communicate their work to others.

Settings. In total, 30 Java programs and 30 Python programs were analyzed (one program/student). The courses taught in Java had fewer participants than the Python ones, and we had to choose two Java assignments to collect the needed number of programs. The students completed the assignments using a computer, and were thus able to execute and test the programs. The assignment specifications can be found in Appendix 2.

The assignments were not identical, but for the sake of this study the main thing was that all analyzed programs covered the same programming constructs. The problem-solving process can be divided into different stages, for instance as follows: (1) understanding the problem, (2) coming up with a solution, (3) developing an algorithm, and (4) implementing the algorithm. When analyzing errors and program criteria such as the ones in this study, the main focus is on the logic (stages 2–3) and the syntax (stages 3–4) of the program. The first stage of understanding the problem and from that coming up with a solution is not as relevant. All programs were non-blank, i.e. contained code related to the assignment specification.

One of the assignments required checking for the leap year property. Since the other two assignments did not require the students to write such complex conditions, we did not include logic errors originating from incorrect conditions in our analysis. Reliability issues are discussed more extensively in Appendix 2.

Analysis. We created an analysis instrument that let us evaluate some pre-determined aspects in the programs. The analysis was conducted manually, and the code of each program was analyzed separately. Each program was executed with different input, in order to see how the program behaved on general, arbitrary input and on more “tricky” input.

The number and type of errors in each program were analyzed and classified using two categories: logic and syntax errors. Logic errors indicate a misunderstanding or an incorrect algorithm (e.g. accepting erroneous input, using selection instead of iteration or vice versa, and declaring unused variables), whereas syntax errors can be considered less severe, since these do not necessarily indicate any misunderstandings, but can be made by mistake (e.g. missing brackets or semicolons, misspelled keywords and syntactically malformed statements). They are, however, frustrating for novice programmers (Kelleher & Pausch, 2005), and including these errors in the analysis was, in our opinion, essential for the purpose of the study. The categories of errors are broad and imprecise, but are sufficient for distinguishing between errors related to understanding, and errors that arise from features of the language used.

Each program was also analyzed based on four additional criteria: (1) program execution, (2) satisfaction of specification, (3) error handling, and (4) program structure. The level of program execution (i.e. whether the program runs correctly all the time, on some input only, or does not compile/run at all) and satisfaction of problem specification (i.e. whether the program complies with the requirements and does what it is expected to do) can be seen as vital characteristics of any program. Error handling has traditionally been viewed as problematic when learning programming, and we wanted to find out to what extent students’ programs check for and deal with errors. Finally, the structure is important in any program: a clear structure makes the code easy to read, debug, maintain and modify, which is important to both the programmer and to everyone who comes in contact with the code.

2.2. Part 2: Transition to the Second Language

Close collaboration between the university and high school gives us a rather unique opportunity to make longitudinal follow-ups of students from their very first programming course at HS to specialized university courses.

Settings. We followed up on eight students from the Python group as they moved on to learning Java in fall 2005, when they participated in the first programming course at the university. The contents of the course were largely the same as what the eight HS students had learned earlier using Python.

At the end of the university course, they were given a Python solution to a problem, and were asked to write a corresponding Java version. The difficulty level of the problem was similar to that of the assignments they were used to solving. In addition, semi-structured interviews about the programming task and learning programming

provided qualitative insight into the experiences of individual students. Care was taken to make the interview questions (see Appendix 3) open and non-leading.

Analysis. When analyzing the Java programs resulting from the programming task, special attention was put on finding indications of errors or misconceptions that could originate from the students' Python background. We were also interested in seeing whether the students had continued to use any Python conventions in Java.

The interviews were recorded, transcribed and analyzed. Main focus was put on finding comments related to (1) conceptual comprehension (what is programming?), and (2) sources of difficulty when learning a new language. For each student, the interview transcript and the program were also analyzed in conjunction, in order to see whether there was an agreement between the two data sets, or if any discrepancy could be found.

3. Results

3.1. Part 1: Program Analysis and Comparison

Syntax and logic errors. The Python programs included substantially fewer syntax errors than the Java ones (2 versus 19). The most common syntax errors in Java were missing brackets, missing semicolons, use of uninitialized variables or badly spelled keywords. In Python, the errors originated from incomplete statements or a missing colon.

The difference in the number of logic errors was also notable (17 in Python, 40 in Java). We were able to classify the logic errors in five distinct categories: (1) accepting erroneous input, (2) incorrect or missing algorithm, (3) mix up of constructs (if and while), (4) incorrect or redundant declarations of variables or subroutines, and (5) other errors (Figure 1).

Functionality. As Figure 2 shows, there was a difference between the languages when considering the proportion of programs running correctly and satisfying the problem specification (Figure 2). More Python programs than Java programs ran correctly and satisfied the specification, whereas more Java programs did not run or satisfy the specification. Reliability issues are discussed in Appendix 2.

Interestingly, the proportion of Python programs that ran correctly was smaller than the proportion (of the same programs) that satisfied the specification (Figure 2). One would expect that a program satisfying the specification also would run correctly; however, a couple of Python students had “done more” than what was called for in the assignment, resulting in the program not running correctly on some input, although the program satisfied the specification.

Specific findings. Lack of error handling was a commonly found bug: many students failed to identify situations in which error checking and handling should be included. For instance, in one of the Java assignments the user was to input a positive

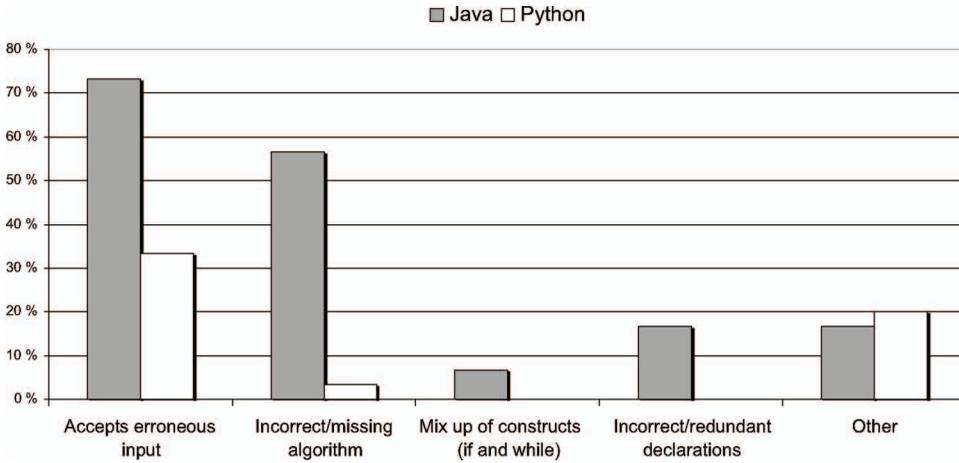


Figure 1. The type of logic errors and their proportional occurrence in Java and Python programs respectively

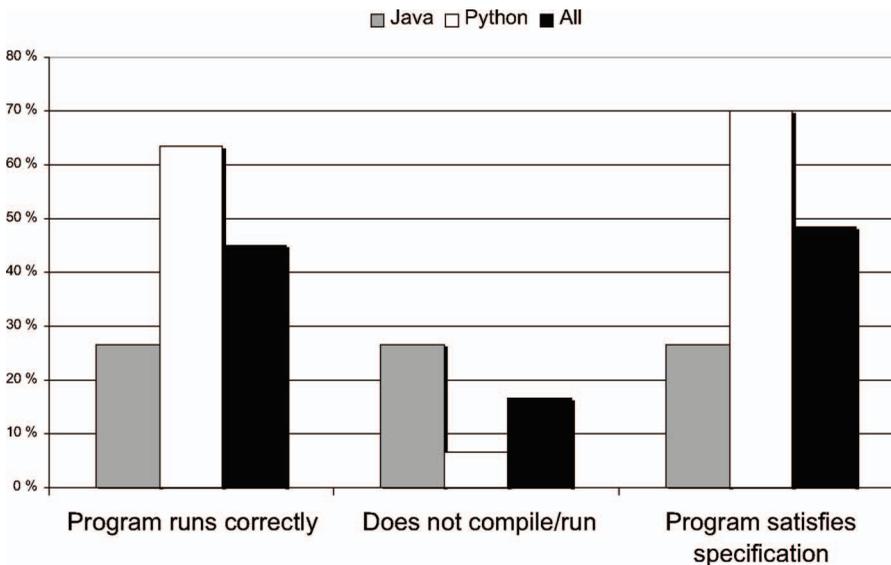


Figure 2. The proportions of programs that ran correctly, did not run, and satisfied the specification

integer, and less than 30% of the programs checked that the input data were acceptable. The percentage of Python programs including exception handling was higher (about 65%).

Another interesting difference was found with regard to typing and variables. Examples of such errors were found in 13% of the Java programs, e.g. in the form of choosing the wrong type for a variable or declaring variables that were never used. Moreover, 7% of the Java programs used uninitialized variables. As for Python, we found no such errors. The dynamic typing of Python did not give rise to any problems

either; the students consequently used new names when storing data of different types.

A small difference in the structuring of code was found. The syntax of Python requires a strict indentation, which enforces students to write structured code. Thus, as expected, no examples of bad indentation were found in the Python programs. Most Java programs were also well structured, although non-severe examples of unclear indentation were found in almost 25% of these. Python uses indentation to denote blocks, whereas Java uses brackets, and bugs resulting from missing begin and/or end-brackets were found in 10% of the Java programs.

3.2. Part 2: Transition to the Second Language

Conceptual comprehension. The interviews (I1–I8, below) revealed that the students distinguished between learning programming and learning a programming language. They also felt that the idea of programming is something that remains the same, regardless of the language used.

“The programming idea is the same, no matter what language you use.” (I7)

“The idea of programming, a certain kind of constructs and ways of doing things.” (I3)

One aim of the follow-up study was to evaluate the consequences of knowing Python when learning Java. Whereas the results from the programming task let us develop our interpretation of the consequences, the interviews gave insight into the students’ own opinions and experiences. Most students indicated that their previous Python background had been a definite advantage during the university course.

“I haven’t had to learn programming in this course anymore, just another syntax.” (I1)

“It’s easier to learn new languages once you know some from before. The idea is exactly the same, only the syntax that is different.” (I2)

None of the students would have preferred to learn Java already at HS level, but all appreciated having learned programming using a simple language. Nevertheless, some found arguments for learning Java later on.

“No, I still think Python is that much easier, and most students probably would learn that easier. Should teach the idea, not the language. The idea probably comes out easier . . . [using Python].” (I5)

“And it has been good to learn the semicolons and brackets now too, since they are there in most languages.” (I8)

A summary of the opinions related to learning a new language that came up in the interviews can be found in Appendix 3.

Sources of difficulty. The interviews revealed that all students had seen the “necessary details” in the Java syntax, such as semicolons, brackets and other syntactical features as a challenge. However, this appeared to have been easy to handle and overcome; all but one student pointed out that it did not take them long to adopt the

“Java way” of writing code. Moreover, half of the students considered the Java libraries (e.g. I/O) problematic, and one student felt that the reference semantics of Java were illogical.

We had expected that the static typing and the need for compilation would have been problematic for some students but, surprisingly, this was not the case: none of these issues was mentioned in the interviews. The programs from the programming task confirmed that the static typing had not been a problem; all variable declarations included a type. Moreover, both the interviews and the programs indicated that students had continued to use the strict indentation of Python also in Java.

During the interviews, students were also asked about their experiences from the programming task. Only one student found the task difficult, and of the eight programs, his was the only one that was clearly faulty. The other solutions were quite correct. Although the students were not able to write completely correct programs, they pointed out that the main idea of the program was there:

“Didn’t remember all the [syntactic] details exactly, but I don’t regard such things as difficulties. [...] It [the program] doesn’t necessarily work, but it contains the idea.” (I4)

This was confirmed when analyzing the programs—although some students had not known how to implement I/O and exception handling, they had written comments at the correct places of the code, stating what should be accomplished at that specific point of the program. The interviews also explained why exception handling and user input were problematic: exception handling had not been covered at all during the university course, and the I/O class used was rather new, and there had not been time to cover it thoroughly.

4. Discussion

The most eye-catching finding in part one of the study was expected: Python programs included less syntax errors than Java programs. This is natural when considering that Python is marketed as a language with a simple and clear syntax. Although it is a rich language, it has an integral subset that provides sufficient power for implementing structured solutions. Java, on the other hand, has a verbose syntax, which enforces notational overhead even when writing short programs. Examples of the syntactical differences can be found in Appendix 1.

Although syntax errors are annoying, they are still quite trivial and easy to correct. We were more interested in finding out whether the results indicated any difficulties beyond the syntactical level. The analysis showed that the Java programs also contained more logic errors than the Python ones. A verbose and complex syntax might result in novices finding it necessary to focus on getting the syntax correct to such an extent that the algorithm becomes a secondary concern only. In that case, it would be reasonable to assume that a simple syntax leads to fewer logic errors—a hypothesis which seems to be supported by our findings.

For instance, in the Java programs, declarations resulted in logic errors, e.g. not using declared variables, whereas no such errors were found in the Python programs. Moreover, dynamic typing is assumed to make programs more prone to errors, since it is possible to use the same name for storing different types of data. The risks of introducing hard-to-find bugs are also believed to increase when type checks cannot be performed at compile time. However, no such problems were found in the data from the first part.

Furthermore, the low proportion of Java programs that included error checking indicates that Java students do not understand when e.g. user input should be checked. Due to lack of time, exception handling was not covered thoroughly in the Java courses, but this does not justify the lack of error checking; the teachers still emphasized the importance to always check that user input is valid and deal with erroneous data, either using exception handling or sentinels.³ The 27% of the Java students that did check for errors used a sentinel, which is something all students should have known how to do. All Python students, who included error checking in their programs, used exception handling.

When comparing Python and Java as languages, Python seems to be a more suitable choice in light of the afore-mentioned features suggested by Milbrandt (Milbrandt, 1993). The differences in errors found in the first part of our study support a similar view. However, one could also argue that by teaching Python instead of Java, we are doing our students a disservice; that we are teaching them something useless, and that they will have to “relearn” most of the things when starting to program in a “real” language, such as Java. The idea of such an argument is that it would be better to get exposed to this common type of syntax from the very beginning.

The second part of our study has shown that this is not the case. When analyzing the Java programs written by the Python students, we found almost no error originating from the transition: the programs included semicolons, brackets, modifiers, and other syntactical details not required in Python. In cases where the students had not known how to implement something, e.g. exception handling or I/O that had only been touched upon briefly or not at all in the university course, they wrote comments in the code explaining what should be done in that specific place of the code. In our opinion, this indicates that the students would have understood how to complete the task, had they only been aware of the tools needed.

We were pleased to hear that the majority of students talked about having learned “the idea of programming” during the HS courses; after all, this is exactly the main goal of programming education at HS. All students found benefits from learning programming in a simple language before turning to a more complex one, and none of them would have preferred to learn Java at HS. No indications of the students having problems in adopting the static typing of Java after learning Python were revealed. Furthermore, the students indented the code as strictly in Java as in Python, which, in our opinion, can be seen as a welcome side effect. Our findings thus suggest that there is no harm in teaching the basics of programming as a general skill using a simple, dynamically typed language, which strengthens the arguments for using such a language in the first programming course.

5. Conclusion and Future Work

HS courses span a limited amount of time (28 hours in the Finnish educational system), and it is essential that programming tuition can focus on the important things, and avoid wasting time on irrelevant issues, such as going through and memorizing a verbose syntax. Given the complex and rich syntax of Java, there is not time to cover all basic concepts during an introductory course; this was demonstrated by the lack of error handling above. In addition, programming courses at HS level should also inspire those students, who are interested in a future career as computer scientists; especially given that prior programming experience has been shown to have a positive impact on university CS studies (Hagan & Markham, 2000). HS is the place in which future university students receive their basic skills; should we not then concentrate on giving them a solid ground in algorithmic thinking and programming, instead of in programming languages and syntax?

One criticism towards teaching Python as the first language is that we teach students a strange language. However, the results presented in this study suggest that students learning programming using Python do not make as many errors as those learning using Java. This is, in our opinion, reason enough for considering introducing programming using a simple language at HS level. Moreover, the results indicate that learning programming using such a language is no drawback for those who do want to become professionals in the field either. Learning the idea of programming using, for instance, Python seems to have potential for facilitating learning more complex and advanced languages further on. Thus, although our original interest was in teaching programming as a general, all-round skill, it seems as if it could be beneficial to choose a simpler language as the first one also when educating future professionals in the field.

Although a lot of research has been done in the field of introductory programming, we still feel that much remains undone. We will continue working according to the principles of action research, i.e. iteratively giving courses, reflecting on the results and making changes as needed. For instance, after the four courses using Python discussed in this paper, we have given the same course to four new HS student groups—with some changes made. For instance, we rethought the syllabus in order to introduce abstract topics such as exception handling at an earlier stage. We also emphasized reading and tracing skills from the very beginning of the course, and collected data on how well students are able to read and understand ready-made programs. Moreover, the exams used in the courses included exactly the same assignments as the Java ones used in the first part of this study, which gave us completely comparable programs for further investigation of the substantial difference in error rates between programs written in the two languages. We are currently in the process of beginning to analyze these new data sets.

Besides teaching basic programming, we have also taught HS students object-orientation and graphics in a continuation course using Python. This course has now been given twice, and the collected data will make it possible for us to analyze how well suited Python is for teaching more advanced topics. We are also interested in

studying students' experiences of the difficulties in learning programming using different languages.

Notes

1. www.tiobe.com/tpci.htm
2. In the Finnish education system, high schools are referred to as upper secondary schools, providing education to students aged 16–19. The principal objective is to offer general education, preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies. A course is a study entity that gives credits.
3. A sentinel is a loop-terminating condition. For error checking purposes, one can, e.g. use a sentinel-controlled loop to keep asking the user for input until the input provided is acceptable.

References

- Biddle, R., & Tempero, E. (1998). Java pitfalls for beginners. *SIGCSE Bull.*, 30, 48–52.
- Clear, T. (2004). Critical enquiry in computer science education. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research* (pp. 101–125). London: Taylor & Francis.
- Close, R., Kopec, D., & Aman, J. (2000). CS1: perspectives on programming languages and the breadth-first approach. *Journal of Computing in Small Colleges*, 228–234.
- De Raadt, M., Watson, R., & Toleman, M. (2002). Language trends in introductory programming courses. Informing Science and IT Education Conference (pp. 329–337).
- Grandell, L., Peltomäki, M., Back, R.-J., & Salakoski, T. (2006). 'Why complicate things? Introducing programming in high school using python. 8th Australasian Computing Education Conference, 52 (pp. 71–80).
- Hadjerrouit, S. (1998). Java as first programming language: a critical evaluation. *SIGCSE Bull.*, 30, 43–47.
- Hagan, D., & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? *ITiCSE '00: Proceedings of the 5th annual ITiCSE conference* (pp. 25–28).
- Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37, 83–137.
- Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ITiCSE-WGR '04: Working group reports from the ITiCSE conference, Leeds* (pp. 119–150).
- McCracken, M., Almstrum, V., Dia, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.*, 33, 125–180.
- Milbrandt, G. (1993). Using problem solving to teach a programming language in computer studies. *Journal of Computer Science Education*, 8, 14–19.
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education*, 13, 137–172.
- Schollmeyer, M. (1996). Computer programming in high school vs. college. *SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on CS education* (pp. 378–382).

- Shaffer, D. (1986). The use of Logo in an introductory computer science course. *SIGCSE Bull.*, 18, 28–31.
- Spoehrer, J.C., & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29, 624–632.
- Stephenson, C., & West, T. (1998). Language choice and key concepts in CS1. *Journal of Research on Computing Education*, 31, 89–95.
- van Rossum, G. (1999). Computer programming for everybody. CNRI: Corporation for National Research Initiatives.
- Winslow, L.E. (1996). Programming pedagogy—a psychological overview. *SIGCSE Bull.*, 28, 17–22.

Appendix 1: Python

The official website (www.python.org) is the source of a large collection of material and tutorials, and contains a review of Python's features. The special interest group of Python in education (www.python.org/sigs/edu-sig/) brings together educators from around the world.

Publications

- Agarwal, K.K., & Agarwal, A. (2005). Python for CS1, CS2 and beyond. *J. Comput. Small Coll.*, 20, 262–270.
- Ceder, V., & Yergler, N. (2003). Teaching programming with Python and Pygame. Presented at PyCon DC 2003, Washington DC, USA, March 26–28, 2003.
- Elkner, J. (2001). Using Python in a high school computer science program. *9th International Python Conference*.
- Elkner, J. (2002). Using Python in a high school computer science program—Year 2. *10th International Python Conference*.
- Guzdial, M. (2003). A media computation course for nonmajors. *ITiCSE '03: Proceedings of the 8th annual ITiCSE conference*, Thessaloniki, Greece (pp. 104–108).
- Oldham, J.D. (2005). What happens after Python in CS1? *J. of Comput. Small Coll.*, 20, 7–13.
- Shannon, C. (2003). Another breadth-first approach to CS1 using Python. *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on CS education*, Reno, ND (pp. 248–251).
- Stajano, F. (2000). Python in education: raising a generation of native speakers. *Proceedings of the 8th International Python Conference*.

Websites

www.cs.ubc.ca/wccce/Program03/papers/Toby.html
<http://zen.sandiego.edu:8080/luby/papers/python.pdf>
www.python.org/sigs/edu-sig/miller-dissertation.pdf
mcs.sp.wartburg.edu/zelle/python/python-first.html

Examples

A version of the simple “Hello World” program.

Python

```
print ``Hi!``
```

Java

```
class Hi {
    public static void main (String args[]) {
        System.out.println(``Hi!``);
    }
}

% javac Hi.java
% java Hi
```

Linear Search in Python

```

def linearSearch(list, element):
    for x in list:
        if x == element:
            return True
    return False

v=[ 'Bob' , 'Doug' , 'Alice' ]
while (True):
    value=raw_input('\nEnter search string or q to quit:')
    if value.lower() == 'q' :
        break
    else:
        print linearSearch(v, value)

```

Linear Search in Java

```

import java.util.Vector;

public class JavaEx {

    public static boolean linearSearch(Vector v, Object o){
        for (int i=0; i < v.size(); i++){
            if (v.elementAt(i).equals(o)) {
                return true;
            }
        }
        return false;
    }

    public static void main(String args[]){
        Vector v=new Vector();
        BufferedReader in=new BufferedReader
            (new InputStreamReader (System.in));
        String[] values={ 'Bob' , 'Doug' , 'Alice' };
        for (int i=0; i < values.length; i++){
            v.addElement(values[i]);
        }
        String value;
        while (true) {
            System.out.println('\nEnter search string or' +
                + '\nq to quit:');
            value=in.readLine();
            if (value.toLowerCase().equals('\nq')) {
                break;
            }
            else{
                System.out.println(linearSearch(v, value));
            }
        }
    }
}

```

Appendix 2: Program Analysis

Assignment Specifications

1. Write a program that asks the user for a positive integer and outputs the multiplication table for that number (Java, N = 18).
2. Write a program that asks the user to input two years and then outputs all leap years in that interval (Java, N = 12).
3. Write a program that checks if integers are divisible by three. The user should first input a positive integer indicating how many numbers he/she wants to check, after which the program should let the user input one integer at a time, check if it is divisible by three, and output the result. This is repeated until the user chosen number of integers has been checked (Python, N = 30).

Reliability Discussion

The exam assignments that were the basis for the programs analyzed in the first part of the study were not identical. However, they did require using the same concepts and constructs, testing basic programming skills of the students. The leap year program in Java can be regarded as more difficult than the other two programs, but the impact of this on our analysis has been diminished, e.g. by excluding errors that clearly originated from the complex condition required in order to check for the leap year property.

We did, however, want to analyze the frequency of incorrect or missing algorithms in the programs, although the number of this bug was rather high for the leap year assignment (Figure 1). If the leap year data were excluded from the analysis, logic errors originating from an incorrect or a missing algorithm were found in 33% of the remaining 18 Java programs; i.e. still higher than the corresponding amount in Python (3%). Likewise, one could take away the impact of the leap year assignment from the amount of programs that did not run, resulting in 39% of the remaining 18 Java programs running correctly; this is still less than the corresponding proportion for the Python programs (63%).

Appendix 3: Interviews

Interview Questions

I The programming task

1. What was it like solving the problem?
 - What was difficult?
 - What was easy?
2. Do you think your Java version is flawless?

II Experiences from the university course

1. What did you, in general, find difficult?
2. What did you, in general, find easy?

III Opinions about the high school courses

1. Do you feel that you have had any use for what you learned at the programming courses at school? What? In what way?
2. Would you have preferred to learn Java already at school? Why/ why not?

IV Programming in general

1. Do you feel that there is something in programming, which is the same regardless of the language used? What? In what way?
2. Did our talk bring up any thoughts related to programming that you would like to tell me about?

Summary of Opinions

	Q1: Do you feel that you have had any use for what you learned at the programming courses at school?	Q2: Would you have preferred to learn Java already at school?	Q3: Do you feel that there is something in programming, which is the same regardless of the language used?
I1	Yes, actually, the advantage has been that I haven't had to learn programming in this course anymore, just another syntax. When knowing Python, I haven't actually had to study at all during this Java course. The basic ideas have been the same as in the school courses.	Difficult to say, but I feel that Java would have had a higher learning threshold. Python is a good language to start with in that sense that it is very simple. But on the other hand, I'm not sure about Java, I don't know if I really want to learn it now either. C or something corresponding instead since you run in to them more frequently.	The idea of programming.
I2	I already know how to program, so there hasn't actually been anything else than the syntax issues. It's easier to learn new languages once you know some from before. The idea is exactly the same, only the syntax that is different.	No, because I like Python better, and I have no motivation to learn Java now either.	The idea of programming: putting statements after each other, trying to accomplish something that makes sense. Have to remember all the small things, can't forget anything.
I3	Yes. In general learning the way of thinking, using methods, the idea of structure, different constructs (loops),	Starting with Python was a good soft start into the world of programming, perhaps could have been nice to have a course in Java immediately after the Python one to see how the same things could be done in Java. Python is good for starting, and getting to understand programming, clarifying the main constructs.	The idea of programming, a certain kind of constructs and ways of doing things.
I4	Python teaches programming in a very nice way. First of all, it works, it is not a pseudo language. But the syntax reminds quite strongly of a pseudo language and doesn't have much... It's similar to natural language. When thinking about programming languages, at a certain point, the logic, or the programming per se changes into a different skill, compared to a given language. When I look at Java, I see how "mean" they have made it, all the semicolons etc. which are more related to how you get the idea itself into becoming a running program, than to the idea or the structure. You just have to learn it, but then you have to spend a lot of time thinking about how to implement the ideas. And then you start to realize that you are not using the time usefully.	Python is a very good language. I'm not sure how much it has to do with to what extent you understand that programming languages are mechanical, and not illustrative as natural langs. If you are not used to exact language use, you might get confused since Python is very exact. Python is a good language for teaching, shouldn't go back to teaching Java at school. Python teaches more how to "get the picture" because you aren't forced to write down the semicolons, it's hidden. In my opinion, in a prog course, the aim should be to teach programming. The syntax of Java could make it easier for a newbie to realize that a programming language is mechanical, since it has things that don't mean anything to humans but are very important to the machine, whereas Python's syntax can be seen as a "natural" language, and thus might not show the difference as well. However, I would not choose a difficult language for teaching for this reason.	Covered in the answer to question Q1.
I5	Well, I didn't actually learn anything new [had programmed before] but for those new to programming Python was probably a good language to start with.	No, I still think that Python is that much easier, and most students probably would learn that easier. Should teach the idea, not the language. The idea probably comes out easier...[using Python].	Easy to learn a new language when you have some other language as a base. The simple things are all the same.
I6	Yes, since I learned the basic things [the constructs] there, and then when you go on to Java and see a while, you can see that it is almost the same.	No... Well I'm not 100% sure... The thing about Python is that it is really simple, and I'm not sure if that's good if you think of it as the default language, and never have used anything more complex.	The basic things (while, for, if) are all the same, but there can be differences in eg I/O, a bit more difficult in Java. Could be good to learn the basics in two languages, e.g. Python and C++. Either in parallel or in different courses. First learn basics in Python, and then when you understand them, you can go on to learning the same things in C++/Java.
I7	Not from the school courses in particular. I already knew how to program before taking those courses, so I would say I have had use of my programming background in general.	No, Python is a good starting language. It's fast, gets you down to business quickly. The high school course should at least not be in Java.	The programming idea is the same, no matter what language you use.
I8	Well yes, all the basic things are the same. But I had done some Python programming before the school courses too. And it has been good to learn the semicolons and brackets now too, since they are there in most languages.	Have thought about it. On one hand, Python is an easy scripting language with a simple syntax, but on the other hand it doesn't teach all hardware related issues, as e.g. Java does (variable declarations, classes etc.). But no...	The idea of programming is always the same.

Paper VII

Novices' Progress in Introductory Programming Courses

L. Mannila

Originally published in *Informatics in Education*, 6(1), pp. 139-152. Institute of Mathematics and Informatics, Lithuanian Academy of Sciences, Lithuania, 2007.

Novices' Progress in Introductory Programming Courses

Linda MANNILA

*Dept. of Information Technologies, Åbo Akademi University, Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
e-mail: linda.mannila@abo.fi*

Received: August 2006

Abstract. This paper presents an approach for educators to evaluate student progress throughout a course, and not merely based on a final exam. We introduce *progress reports* and describe how these can be used as a tool to evaluate student learning and understanding during programming courses. Complemented with data from surveys and the exam, the progress reports can be used to build an overall picture of individual student progress in a course, and to answer questions related to how students (1) understand program code as a whole, (2) understand individual constructs, and (3) perceive the difficulty level of different programming topics. We also present results from using this approach in introductory programming courses at secondary level. Our initial experience from using the progress reports is positive, as they provide valuable information during the course, which most likely would remain uncovered otherwise.

Key words: introductory programming, learning programming, program understanding, SOLO study.

1. Introduction

In (Grandell *et al.*, 2006), we reported on a study from teaching introductory programming at high school level.¹ The results showed that abstract topics such as algorithms, subroutines, exception handling and documentation were considered most difficult, whereas variables and control structures were found rather straight forward. These results were in line with those of other researchers (e.g., (Haataja *et al.*, 2006; Lahtinen *et al.*, 2005)). In addition, our findings indicated that most novices found it difficult to point out their weaknesses. Moreover, exam questions asking the students to read and trace code showed a serious lack in program comprehension skills among the students. One year later (2005/2006) we conducted a new study, in which we further investigated these issues using what we have called *progress reports*. This paper presents the results from this study.

¹In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16–19. The main objective of these schools is to offer general education preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies.

We begin the paper with a background section, followed by a section describing the study and the methods used. Next, we present and discuss the results, after which we conclude the paper with some final words and suggestions for how the ideas presented can be implemented in practice.

2. Background

Introductory programming courses tend to have a strong focus on construction, with the overall goal to get students to write programs as quickly as possible using some high-level language. This is understandable as the aim of these courses is to learn programming, which is commonly translated into the ability to produce code using language constructs. Writing programs is, however, only one part of programming skills; the ability to read and understand code is also essential, particularly since a programmer spends much time maintaining code written by somebody else.

One could assume that students learning to write programs automatically also learn how to read and trace code. Research has, however, indicated otherwise. For instance, Winslow (1996) notes that "[s]tudies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught [...]" (p. 21). In 2004, a working group at the ITiCSE conference (Lister *et al.*, 2004) tested students from seven countries on their ability to read, trace and understand short pieces of code. The results showed that many students were weak at these tasks.

Why then is it difficult to read code? Spinellis (2003) makes the following analogy: "when we write code, we can follow many different implementation paths to arrive at our program's specification, gradually narrowing our alternatives, thereby narrowing our choices [...] On the other hand, when we read code, each different way that we interpret a statement or structure opens many new interpretations for the rest of the code, yet only one path along this tree is the correct interpretation" (p. 85–86).

Fix *et al.* (1993) cite Letovsky, who has stated that it usually is quite easy to infer the overall goals of a program only by reading the code, based on for instance variable names, comments and other documentation. Letovsky also suggests that the same thing applies to the program implementation: a person reading through a program understands the actions of each line of code separately. The difficulty arises when trying to map high-level goals to their representation in the code. Similarly, Pennington (1987a) has found that whereas experts infer what a program does from the code, less experienced programmers make speculative guesses based on superficial information such as variable names without confirming their guesses in any way. A study conducted by Lister *et al.* (2006) based on the SOLO (Structure of the Observed Learning Outcomes) taxonomy also supports these findings. The SOLO taxonomy was originally introduced by Biggs and Collis (1982), and can be used to set learning objectives for particular stages of learning or to report on learning outcomes. The taxonomy is a general theory, not specifically designed to be used in a programming context. Lister *et al.*, however, used the taxonomy to describe how code is understood by novice programmers, and illustrate the five SOLO levels applied to novice programming as follows:

Prestructural "In terms of reading and understanding a small piece of code, a student who gives a prestructural response is manifesting either a significant misconception of programming, or is using a preconception that is irrelevant to programming. For example, [...] a student who confused a position in an array and the contents of that position (i.e., a misconception)" (p. 119).

Unistructural "[...] the student manifests a correct grasp of some but not all aspects of the problem. When a student has a partial understanding, the student makes [...] an 'educated guess'" (p. 119).

Multistructural "[...] the student manifests an understanding of all parts of the problem, but does not manifest an awareness of the relationships between these parts – the student fails to see the forest for the trees" (p. 119). For example, a student may hand execute code and arrive at a final value for a given variable, still not understanding what the code does.

Relational "[...] the student integrates the parts of the problem into a coherent structure, and uses that structure to solve the task – the student sees the forest" (p. 119). For instance, after thoroughly examining the code, a student may infer what it does – with no need for hand execution. Lister *et al.* also note that many of the relational responses start out as multistructural, with the student hand tracing the code for a while, then understanding the idea, and writing down the answer without finishing the trace.

Extended Abstract At the highest SOLO level, "the student response goes beyond the immediate problem to be solved, and links the problem to a broader context" (p. 120). For example, the student might comment on possible restrictions or prerequisites, which must be fulfilled for the code to work orderly.

Lister *et al.* found that a majority of students describe program code line by line, i.e., in a multistructural way, and that weak students in particular seem to have difficulties in abstracting the overall workings of a program from the code. They talk about students failing to "see the forest for the trees" (p. 119) and argue that students who are not able to read and describe programming code relationally do not possess the skills needed to produce similar code on their own.

Pennington (1987a) has developed a model describing program comprehension, according to which a programmer constructs two mental models when reading code. First, the programmer develops a program model, which is a low-level abstraction based on control flow relationships (e.g., loops or conditionals). This program-level representation is formed at an early stage and inferred from the structure of the program code. After that, the programmer develops a domain model, which is a higher-level abstraction based on data flow containing main functionality and the information needed to understand what the program truly does. Similarly, Corritore and Wiedenbeck (1991) have found that novices tend to have concrete mental representations of programming code (operations and control flow) with only little domain-level knowledge (function and data flow).

3. The Study

3.1. Data

The data analyzed in this study were collected during two high school introductory programming courses in 2005/2006. The majority of the 25 students had no previous programming background. The courses were taught using Python and covered the basics of imperative programming. To validate our results, as well as to deepen and widen our understanding of how novices' progress in an introductory programming course, we used data triangulation (Mathison, 1988) to investigate the same phenomenon from different perspectives. The data collection was conducted using surveys, progress reports and a final exam.

- The *pre-course survey* was used to collect background information about the students, and, e.g., their programming experience, whereas the *post-course survey* provided information about how the students had experienced the course.
- A *progress report* can be seen as a type of learning diary written on paper, aiming at revealing the students' own opinions and thoughts about their learning. What differentiates it from a traditional diary is that in addition to calling for self reflection, the report makes it possible for the teacher to evaluate students' understanding based on their responses to "trace and explain" questions. In this study, we used two progress reports that were handed out after 1/3 and 2/3 of the course respectively. Each report included a piece of code dealing with topics recently covered in the course as well as four questions. First, in the "trace and explain" questions, the students were to read the code and in their own words (1) describe what each line of the code does, and (2) explain what the program as a whole does on a given set of input data. In addition, students were asked to state what they had learned and what had been most difficult so far in the course.
- To gain further insight into the individual progress and to see how well students were able to write code related to the topics dealt with in the progress reports, we analyzed code they generated on the *final exam*. We selected a sample of 10 exams, for which we analyzed the students' answers to two assignments: 1) a "trace and explain" question similar to the ones in the progress reports and 2) a programming task that involved developing a function that calculates and returns either the factorial of a given number or the mean of a list of numbers. Care was taken when choosing the exams to ensure that they would be representative for all students, and include work of students at different skill levels.

In total, we have analyzed 50 progress reports (two for each of the 25 students), 25 post-course surveys and 10 exams. Given that we knew who had written each answer, we were able to analyze the progress on an individual basis.

3.2. Method

The progress reports, exam assignments and post-course surveys were analyzed in order to investigate three questions: how do students (1) understand program code as a whole,

(2) understand individual constructs, and (3) perceive the difficulty level of different programming topics.

The first two questions were addressed by grouping the explanations given for the "trace and explain" questions according to qualitative differences found in the data. On this point, the study resembles the SOLO study by Lister *et al.* (2006) to some extent. However, in this study, we explicitly asked students for both a multistructural and a relational response for each piece of code, giving us the possibility to compare how well the two responses match for individual students. Using data collected on the final exam, we were also able to analyze to what extent the difficulties to understand code experienced during the course were still an issue at the end of the course.

Finally, to address the third question, we studied the difficulty level of topics as perceived by the students by looking at both the progress reports and the post-course survey. The "trace and explain" questions in the reports and the final exam also provided data pertinent to this part of the study as explanation errors were considered indications of student experiencing problems with that specific topic.

4. Results

4.1. Program Understanding

The "trace and explain" code given in the first progress report is listed below (Algorithm 1).

Algorithm 1. Program given in progress report 1

```
try:
    a = input("Input number: ")
    b = input("Input another number: ")
    a = b

    if a == b:
        print "The if-part is executed..."

    else:
        print "The else-part is executed..."

except:
    print "You did not input a number!"
```

Students were asked to explain what this piece of code does if two integers are given as input. The analysis of the overall explanations gave rise to four categories:

1. Correct explanation ($n = 13$).
2. Choosing the wrong branch in the selection after not explaining the meaning of the statement `a = b` ($n = 5$).

3. Choosing the wrong branch although having explicitly explained the statement $a = b$ ($n = 3$).
4. Giving an output totally different from the ones possible based on the code ($n = 4$).

The first category is straight forward: over half of the students gave a perfect overall explanation for the program code, not only stating the output but also explaining why that specific output was produced. The second category covers responses in which the student had failed to explain what the $a = b$ statement means, and hence also missed the fact that when arriving at the selection statement, a does, in fact, equal b .

In the third category, students who explicitly explained the $a = b$ statement still believed that the else-branch would be executed. This indicates a misunderstanding related to either the effects of an assignment statement, or the workings of the selection statement. In the fourth category, students appeared to be guessing, thinking that the program would output something that might have been expected from the code (e.g., values instead of one of the text messages) but that nevertheless was incorrect.

Algorithm 2. Program given in progress report 2

```
def divisible(x):
    if x % 2 == 0:
        return True
    else:
        return False

default = 5
number = default

while number > 0:
    try:
        number = input ("Give me a number: ")
        result = divisible(number)
        print result
    except:
        print "Numbers only, please!"
```

Algorithm 2 shows the piece of code included in the second progress report. For this program, students were given a list of input data including positive integers and a character, ending with a negative integer. The explanations were analyzed in a similar manner to the corresponding task in the first progress report, and four categories were found:

1. Correct explanation ($n = 8$).
2. Not understanding what it means to return a boolean value ($n = 10$).
3. Missing the last iteration, otherwise correct ($n = 4$).
4. Incorrect output ($n = 3$).

The number of correct overall explanations for the program in the second progress report was smaller than for the first report. The main stumbling block was related to understanding what happens when a subroutine returns a boolean value. The program checks if the input is divisible by two and outputs either `True` or `False` based on the result as long as the input number is positive. Some students thought that returning `True` results in the control being returned to the main program, whereas returning `False` makes the subroutine start all over again. Another misunderstanding was that there is no output whenever the subroutine returns `False`.

The third category indicates that some students also had difficulties deciding when a while loop stops, missing the last iteration (the loop will be executed once more after a negative value is input). The fourth category is similar to the one for the first progress report, i.e., students seemed to be guessing, stating that the program outputs something totally different (in this case the input values) from what the subroutine returns.

4.2. Understanding of Individual Statements

In order to further analyze students' skills to read and understand code, we analyzed how they explained individual statements related to a set of programming topics. The explanations found were categorized as one of the following types:

- *Correct* – the student explained the statement "by the book".
- *Missing* – the student did not write any explanation for that given statement.
- *Incomplete* – the student's explanation was correct to some extent, but lacked some parts.
- *Erroneous* – the student gave an incorrect explanation.

Although the number of students giving correct overall explanations for the programs decreased from the first to the second report (as discussed in Section 4.1), the results presented in the diagram in Fig. 1 indicate that at the same time the students became better at understanding individual statements: the number of correct explanations for individual statements increased while the number of incomplete or missing explanations decreased. When comparing the report results for each student separately, a positive progress trend

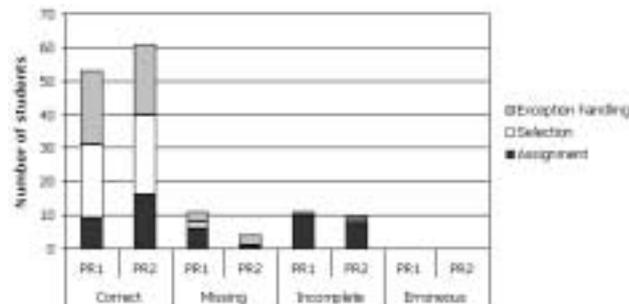


Fig. 1. The frequency of different types of explanations given by students for individual statements in the two progress reports.

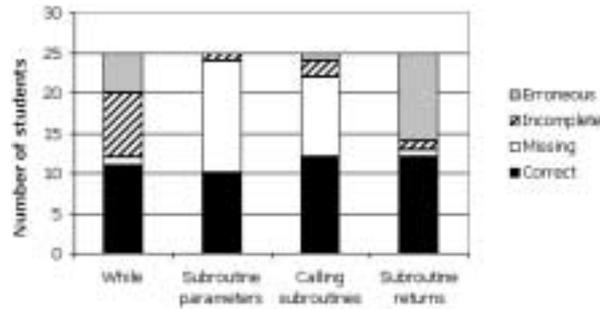


Fig. 2. The frequency of different types of explanations given by students for new topics in the second progress report.

was found: e.g., for assignment statements, 14 students improved their "type" of explanation (e.g., from missing or incomplete to correct).

We found no erroneous comments related to the topics covered in both reports. The second progress report, however, introduced some new topics not present in the first one. The distribution of explanation types for these is illustrated in Fig. 2. The data in the diagram reflect the previously mentioned difficulties related to subroutines returning boolean values: almost half of the students gave incorrect explanations on this point. Interestingly, the other half of the students explained the returns correctly. Many students did not explain the other aspects of subroutines, such as calls or parameters, which makes it difficult to say anything about how well these topics were understood.

4.3. Difficulty of Topics

Apparently, assignment statements constituted the most common difficulty for students in the first progress report. However, this was not reflected in the students' own opinions on what they found difficult in the course at that time. Instead, they mentioned topics such as loops ($n = 4$), the selection statement ($n = 2$) and lists ($n = 3$). Moreover, 40% of the students only gave an incomplete explanation for what $a = b$ means, not mentioning values or variables, but stating for instance that "a becomes b" or "a is b".

In the second progress report, the problems students faced in the "trace and explain" questions (returns and subroutines) were in line with the difficulties they reflected upon in the other questions: almost half of the students stated that subroutines were most difficult. Some students still reported having problems with lists ($n = 2$) and loops ($n = 2$).

In the post course survey, students were asked to rate each course topic on the scale 1–5 (1 = very easy, 5 = very difficult). The results showed that the perceived difficulty levels were quite consistent with the corresponding results presented in our previous study (Grandell *et al.*, 2006). Subroutines, modules, files and documentation were still regarded as most problematic (average difficulty of 2.8–3.2). There was, however, one exception: in the previous study, exception handling was also experienced as one of the most difficult topics (average of 2.9), but in the current study this was no longer the case (average of 2.1). The progress reports supported this finding: exception handling was not mentioned

as a difficult topic at all, and nearly all students gave a perfect explanation for statements dealing with exception handling in the "trace and explain" questions.

When analyzing the exam assignments, focus was put on the use of assignments, if-statements, exception handling, loops and subroutines (declaration, calling and returns), i.e., the topics covered in the progress reports. Any difficulties found related to other topics were, naturally, also taken into account. As a whole, the analysis of the student solutions to the programming assignment indicated that the students, without problems, had implemented the topics that they had found difficult to explain in the progress reports. All students but two had clearly understood the problem and developed an algorithm to solve it. Half of the students had written perfect programs, whereas two of them seemed to have had problems with lists and inputting multiple values. In addition, the "trace and explain" question included in the exam showed that the students were also able to explain these topics without difficulty. It thus seems as if students had overcome the difficulties they had had earlier in the course.

5. Discussion

The "trace and explain" questions in the progress reports revealed some surprising results. For instance, we had not expected assignment statements to be difficult. Nevertheless, this was one of the main problems in the first progress report. Explaining $a = b$ with something like "a equals b" or "a becomes b" is not a valid explanation; clearly, such a student has some idea of what happens, but without mentioning values the explanations are not exact enough. According to the students themselves, assignment statements were, however, not a problem. This strengthens our previous findings (e.g., in (Grandell *et al.*, 2006)): novices do not always recognize their own weaknesses and are therefore not able to point them out.

Moreover, we had expected that subroutines would be perceived as difficult in the second report, since these are commonly one of the main stumbling blocks in introductory programming (Grandell *et al.*, 2006; Haataja *et al.*, 2006; Lahtinen *et al.*, 2005). However, the analysis revealed that this was not necessarily the case; instead of subroutines per se being the problem, surprisingly many students seemed to not understand the effects of a subroutine returning a boolean value.

Many students did not explain subroutine calls or parameters explicitly, and it is thus impossible to say anything definite about how well the students understood those topics. If all missing explanations indicate "erroneous explanations", the number of students not understanding subroutine calls and parameters is alarmingly high. On the other hand, if the missing explanations were due to students finding those aspects "obvious", and therefore not needing any explanation, the number of correct explanations for those topics would be high. When taking into account that the overall explanations to the code in the progress reports did not indicate any specific difficulties in calling subroutines with parameters, the latter explanation might be a bit closer at hand.

Given the obvious difficulties with at least parts of the subroutine concept in the progress reports, we had expected to find some problems related to subroutines in the

exam. However, our analysis showed that all students had defined and used their own subroutines in an exemplary way in the programming assignment. In our opinion, the difficulties that novices encounter with the different aspects of subroutines nevertheless merit further investigation. Doing so, instruction could start focusing more on the problematic aspects, not on the ones that have traditionally been given most attention (unless these are the ones found to be the main difficulties).

We were pleased to see that exception handling was no longer among the most difficult topics, as we had made changes to the syllabus in order to facilitate students' learning of this particular topic. We now introduced exception handling at the very beginning of the course together with variables, output and user input, and students got used to check for and deal with errors from the start. It thus seems as if the order in which topics are introduced does have an impact on the perceived difficulty level, as suggested by Petre *et al.* (2003), who have found indications of topics being introduced early in a course to be perceived as "easy" by students, whereas later topics usually are considered more difficult. One could thus expect that the difficulty level of other "difficult" topics could be decreased by introducing the topic earlier in the course. Naturally, all topics cannot be moved to the beginning of the course just to make them all easier for the students – it is up to the teacher to decide what topics he or she thinks are most important, and then consider introducing them early in the course.

Having analyzed the explanations for both individual statements and entire programs, we can conclude that more students were able to correctly explain the program line by line than as a whole. This was found for both progress reports. When related to the levels in the SOLO taxonomy, most students were able to give correct explanations in multi-structural terms, but only part of them did so relationally. Moreover, the analysis showed that students' ability to explain given individual topics increased from the first to the second progress report. This can, however, be seen as quite natural as one could – and should – expect students to gain a better understanding for individual topics as the course goes on and they become more experienced and familiar with the topics.

The categories found were quite similar for both progress reports, and can be related to the SOLO taxonomy as presented by Lister *et al.* (2006). The first category (correct explanation) contains relational responses, whereas the second and third ones (indicating a misunderstanding) can be seen as containing explanations at the pre- or unistructural level. The fourth category (guessing) includes unistructural responses, which can also be seen as the "speculative guesses" mentioned by Pennington (1987a).

In order to bring further light on the "guessing" one could consider conducting additional interviews with some students. By talking to the students about their answers, one could remove some of the speculations and get a better understanding for different answers. Were the students only guessing, or do the answers originate in some subtle misunderstanding that needs to be corrected? Another more resource light approach would be to let the students evaluate (e.g., on a given scale) how confident they are about their explanations. This would make it easier to distinguish between students truly believing in their answers and those merely guessing.

The difficulties found in the student generated code on the exam were not the ones found in the "trace and explain" questions in the progress reports. Rather, the problems

found in the students' programs were better in line with what they had mentioned as difficult (e.g., lists) in the corresponding questions in the progress reports. It thus seems as when students are asked about what they find difficult about programming, they mainly answer based on their experiences from writing code – not reading it. This is, however, a natural result as this is also the main focus of instruction.

6. Conclusion

There seems to be a general lack of attention to program comprehension skills in education. Understanding the workings of an algorithm requires time and practice, and if students cannot understand the code presented they invent their own conceptions and strategies. This is certainly something we want to minimize and avoid. The study reported on in this paper illustrates an approach for teachers to evaluate student progress throughout a course and not merely based on a final exam. Our initial experience from using this approach is positive, especially with regard to the progress reports, as we feel that they provide important information during the course that most likely would remain uncovered otherwise.

The results from the SOLO study presented by Lister *et al.* (2006) are interesting as they divide student responses into different SOLO categories. However, asking the students for both a multistructural and a relational response makes the data even more interesting, since it gives us two different responses for each program. These can be used to analyze how well the responses match for an individual student. As seen in the previous section, students were in general able to give perfect descriptions of the programs line by line, but only a fraction of these gave a perfect explanation of what the program did as a whole. This finding suggests that novice programmers tend to understand concepts in isolation, and is thus consistent with the results presented by Lister *et al.* (2006) and with Pennington's idea of program vs. domain models (Pennington, 1987b). Our study also strengthens our previous findings related to students' awareness of their own stumbling blocks: novice programmers cannot necessarily point out their own weaknesses. In this study, another aspect of the awareness issue was raised as we found that novices seem to think about their difficulties related to programming mainly in terms of writing code, not reading it.

The progress reports serve as a simple mechanism to put a stronger focus on the need for also being able to read and understand code. In this study, the progress reports were mainly used as a feedback tool to perform *continuous checkups* of student progress throughout the course. They can also be used as a basis for intervention during the course, for instance in the form of *individual discussions* in which the teacher would review the progress reports and sort out potential difficulties with each student separately. The extra resources needed (teacher/tutor effort and time) might not be available, and a less demanding alternative would be to only arrange discussions with students who based on the report are in evident need of help.

As educators, we expect students to go through and learn from examples when we introduce a new topic. Doing so, the student's attention is on the construct (program

model) and not on understanding how the given piece of code solves a particular problem (domain model). The progress reports can be used as a *evaluation tool* for evaluating how well our students are doing on the domain level, and give us indications about topics that need to be further explained or taught in another way. We believe that, if used wisely, the information gathered in the reports can make a big difference for both us as educators and our students learning to program.

Acknowledgements Special thanks to Mia Peltomäki and Ville Lukka for collecting the data.

References

- Biggs, J., and K. Collis (1982). *Evaluating the Quality of Learning – the SOLO Taxonomy*. New York, Academic Press.
- Corritore, C., and S. Wiedenbeck (1991). What do novices learn during program comprehension. *International Journal of Human-Computer Interaction*, **3**(2), 199–208.
- Fix, V., S. Wiedenbeck and J. Scholtz (1993). Mental representations of programs by novices and experts. In *INTERCHI '93: Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*. Amsterdam, The Netherlands, IOS Press. pp. 74–79.
- Grandell, L., M. Peltomäki, R.-J. Back and T. Salakoski (2006). Why complicate things? Introducing programming in high school using python. In D. Tolhurst and S. Mann (Eds.), *Eighth Australasian Computing Education Conference (ACE2006)*, CRPIT, Hobart, Australia.
- Haataja, A., J. Suhonen, E. Sutinen and S. Torvinen (2001). High School Students Learning Computer Science over the Web. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*. Available online: <http://imej.wfu.edu/articles/2001/2/04/index.asp>. Retrieved August 29, 2006.
- Lahtinen, E., K. Ala-Mutka and H.-M. Järvinen (2005). A study of the difficulties of novice programmers. In *ITICSE '05: Proceedings of the 10th Annual ITiCSE Conference*. Capacrica, Portugal, ACM Press. pp. 14–18.
- Lister, R., E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon and L. Thomas (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, **36**(4), 119–150.
- Lister, R., B. Simon, E. Thompson, J. L. Whalley and C. Prasad (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull.*, **38**(3), 118–122.
- Mathison, S. (1988). Why Triangulate? *Educational Researcher*, **17**(2), 13–17.
- Pennington, N. (1987a). Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*. pp. 100–113.
- Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19**(3), 295–341.
- Petre, M., S. Fincher, J. Tenenberg *et al.* (2003). "My Criterion is: Is it a Boolean?": A card-sort elicitation of students' knowledge of programming constructs. *Technical Report 6-03*. Computing Laboratory, University of Kent, UK.
- Spinellis, D. (2003). Reading, writing, and code. *ACM Queue*, **1**(7), 84–89.
- Winslow, L.E. (1996). Programming pedagogy, a psychological overview. *SIGCSE Bull.*, **28**(3), 17–22.

L. Mannila is a PhD student at the Department of Information Technologies at Åbo Akademi University and at Turku Centre for Computer Science. Her main research interests are alternative approaches to teaching introductory programming. She is also interested in e-learning and leads a project at the university, within which non-university students are offered basic university level computer science courses on the web.

Pradedančiųjų pažanga mokantis programavimo pradmenų kurso

Linda MANILA

Straipsnyje pristatoma mokytojams skirta priemonė besimokančiųjų pažangai vertinti. Ji tinka ne tik galutiniam vertinimui gauti, bet ir vertinti pažangą per visą mokymosi procesą. Pristatomos pažangos proceso ataskaitos, aprašoma, kaip tai gali būti panaudota vertinant besimokančiųjų programavimo mokymąsi. Kartu su duomenimis iš apklausų ir egzaminų pažangos proceso ataskaitos gali būti naudojamos sukonstruoti išsamų besimokančiojo mokymosi eigos aprašymą ir gali pagelbėti atsakyti į šiuos klausimus: 1) kaip besimokantieji supranta programos tekstą; 2) kaip besimokantieji supranta įvairias konstrukcijas; 3) suvokti įvairių programavimo temų sudėtingumą. Taip pat pristatome rezultatus, gautus, naudojantis šia priemone vidurinės mokyklos programavimo pradmenų kursui. Pirmoji patirtis, įgyta, naudojantis pažangos proceso ataskaitomis, yra teigiama, reikšmingos informacijos pateikiama kurso teikimo metu.

Paper VIII

Teaching the Construction of Correct Programs Using Invariant Based Programming

R-J. Back, J. Eriksson & L. Mannila

Originally published in *Proceedings of the 3rd South-East European Workshop on Formal Methods*. Thessaloniki, Greece, December 2007.

Teaching the Construction of Correct Programs Using Invariant Based Programming

Ralph-Johan Back, Johannes Eriksson, and Linda Mannila

Åbo Akademi University, Dept. of Information Technologies
Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
{backrj, joheriks, linda.mannila}@abo.fi

Abstract. In most computer science curricula, formal reasoning about program correctness is taught separately from practical programming, and is thus by most students considered a purely theoretical activity. It has been a challenge to convince students of the practical applicability of formal methods. We present here an effort to apply *Invariant Based Programming* (IBP), a visual and practical program construction and verification methodology, in an introductory formal methods course as part of a pilot study at Åbo Akademi University. The course introduces a minimum of notational overhead, and allows the student to reason about correctness using mathematical concepts with which they are already familiar (such as set theory). We have used a programming environment with theorem prover support (*SOCOS*) to increase student confidence in the correctness of the program components that they construct. We evaluate the course using a mixed method approach, and provide data which show that IBP is well suited for teaching introductory formal methods.

1 Introduction

In 1989, Edsger Dijkstra called for giving formal methods a higher profile in the computer science (CS) curriculum [18]. His proposal was the starting shot for an extensive debate on CS education and the role of formal methods in it. Some scientists agreed with Dijkstra's suggestion, whereas others disagreed [16]. Two years later, David Gries followed with basically the same message, stating that undergraduates should learn formal methods as a fundamental topic [24]. Ever since, CS academics have debated the importance of encouraging formal practices in CS education.

In this paper, we present a practical invariant based approach to introducing correctness in undergraduate CS courses. The approach is highlighted by a diagrammatic notation and emphasizes formal reasoning. Introducing correctness early in the CS curriculum and the particular approach we have used naturally raise some basic questions:

- How do students experience learning formal methods using this approach?
- How applicable is the use of tool support in the course?
- What difficulties do students encounter when learning formal methods using this approach?

The contribution of this paper lies in addressing these questions as well as in describing the invariant based approach and presenting a model for how it can be used in education. We begin by discussing the role of formal methods in education in section 2, after which section 3 describes the invariant based approach. In section 4, we present the educational setting. The study is presented in section 5, and the results are put forward in section 6. After discussing the findings in section 7, we conclude the paper with some final observations and suggestions for future work in section 8.

Although this study takes place in the context of CS education, we believe that formal methods play an important role for software engineers as well. In our opinion, the mathematical foundations of programming and knowledge about how mathematics can be used to improve reliability and robustness are essential for anyone designing and creating software, regardless of whether they have a degree in CS or software engineering.

2 Formal Methods in Education

Many attempts to introduce program correctness to novice CS students have been made (e.g. [1, 15, 26, 33, 37, 38]), but convincing students of the value of formal methods is a challenge. Formal techniques are commonly perceived as difficult and requiring mathematical sophistication. Moreover, '[t]he computing education community has adopted a curriculum strategy of dividing curricula elements into areas of "theory" and "practice". This causes both faculty and students to view the theory of computing as separate and distinct from the practice of computing.' [1, p. 79] As an effect of this separation students get only little exposure to correctness concepts.

When formal verification is taught as an activity independent from the programming process [27], the students get the impression that the formal approach is merely applicable in theoretical courses. Students are more likely to be motivated by gaining skills that they know are relevant, bring immediate benefits and are valued in industry. These preferences are also used by CS faculty as arguments against teaching formal methods [31]. If such teachers do teach something related to the topic, they will most likely not be enthusiastic or show a true interest in what they are teaching. And a "I don't really believe in this, I just have to teach it" mentality hardly goes far in having a positive impact on students' attitudes to or experiences of the topic at hand.

The nature of software construction may also reduce the experienced need for formal methods. It is completely possible to break design rules when constructing software and still end up with a working program, and it has become more or less the norm in industry to release buggy software. When well-known companies can get away with not writing correct programs, it is not easy to convince novice CS students that they need to do it.

As a result of the general lack of interest in formal methods, it is common that students do not apply what they have learnt in the theoretical courses when doing actual programming. Instead, novices learning to program go about it in a manner resembling a "trial and error" activity, resorting to "endless debugging" with the approach being "try it and see what happens" [12, p. 63]. Although testing and debugging certainly have their place when learning to program, it is essential that CS students learn that

these approaches can never prove that a program is correct, and that other methods are available for that purpose. In the following, we outline one such approach.

3 Invariant Based Programming

Invariant based programming (IBP) is an approach to constructing correct programs, where not only pre- and postconditions, but also loop invariants are written before the actual code. IBP is not new — it was studied already in the 1970s by one of the authors [4, 5] and similar ideas were proposed by for instance Reynolds [29] and van Emden [35]. In 2004, Back [6] revisited the topic and has since then worked on developing IBP into a more practical hands-on method.

In IBP, a program is constructed and verified at the same time. The notion of an invariant is generalized to a *situation*. Each situation is a collection of constraints and describes the set of states that satisfy these constraints. Thus, a loop invariant is a situation, as well as a precondition or a postcondition. An invariant based program may have many different situations and is not restricted to single-entry, single-exit control structures.

In essence, IBP provides a visual representation of a program. A variety of graphical programming/pseudocode formats have been proposed in the literature [13, 30], and all of these have one common goal: “to provide a clear picture of the structure and semantics of the program through a combination of graphical constructions and some additional textual annotations.” [30, p.3] To our knowledge, all of these have, however, focused on representing control flow and data flow. IBP, on the other hand, describes programs from another perspective as it emphasizes the invariant properties of the program data structures, and thus makes it possible to reason about the correctness of the constructed program in a rather straightforward manner. This is all accomplished without sacrificing either clarity or expressiveness of the diagrams.

3.1 An Illustrating Example

We will here exemplify the work flow for developing invariant based programs by constructing a program that implements the selection sort algorithm. We use a cursor to traverse an array from left to right, and for each position we find the smallest element to the right of the cursor and swap that element with the one pointed at by the cursor. After each swap the cursor is advanced, and the array is sorted when the entire array has been traversed.

We start the process by drawing figures illustrating the basic data structures involved and how they will change during execution of the algorithm. Drawing the figures is an essential step of the IBP work flow, as the figures describe the algorithm at work and thus help the programmer get a feeling for the behavior of the algorithm. As this example illustrates, the figures also aid in identifying the situations (invariants) of the program.

The first figure (Fig. 1) illustrates the specification (the pre- and postcondition), which helps us identify the initial and final situations. As situations are considered sets of states, the final situation is a subset of the initial situation where an additional

constraint, $Sorted(A, 1, n)$, is satisfied. We use a Venn-like diagram, a *nested invariant diagram*, to represent the program and the strengthening of situations. Our first diagram is shown in Fig. 2. Due to the nesting, all constraints in an outer set implicitly also hold in all of its subsets and need therefore not be repeated (for instance, $n : integer$ holds in both the initial and the final situation). Dashed arrows are used to indicate the computation that we want to define and are labeled with a potential guard and the variables that may be changed in the computation.

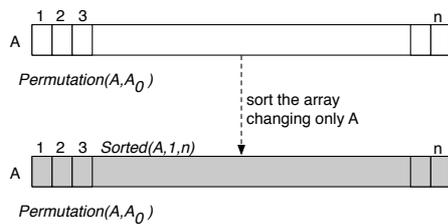


Fig. 1. Visualization of the specification

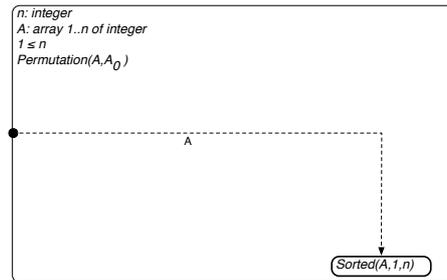


Fig. 2. Corresponding invariant diagram illustrating the initial and final situations

In the same manner as the final situation was identified as a subset of the initial one, we introduce new situations by adding new constraints to the ones present in the more general situations. We further develop the figure of the algorithm at work by introducing the intermediate situation (Fig. 3). As is shown in the corresponding diagram (Fig. 4), this newly inserted situation is a subset (i.e. a constrained version) of the initial situation.

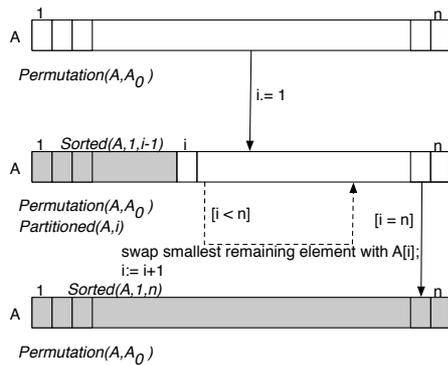


Fig. 3. Sorting program with one loop

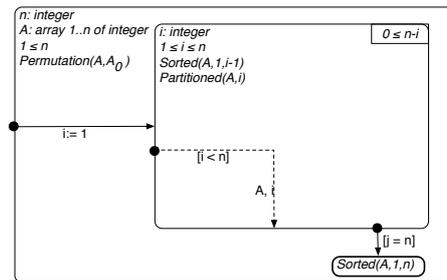


Fig. 4. Invariant diagram with the intermediate situation inserted

Whereas dashed arrows illustrate what we want to accomplish, we use solid ones to indicate computations that we have already planned and defined. We call these solid arrows transitions. Each transition is labeled with a potential guard and the program statements executed when the transition is carried out. We have to check that each transition preserves the situations as follows: assume that we initiate execution in the source situation of a transition and that all the constraints hold for the starting state. Also assume that we reach some target situation after executing the statements for the transition (there may be more than one possible target situation). Then all the constraints of the target situation must hold for the final state. We say that a program is *consistent* if each transition preserves all situations. Consistency is checked for each new transition that we add to the diagram, i.e. we make sure that the newly added transition preserves all situations.

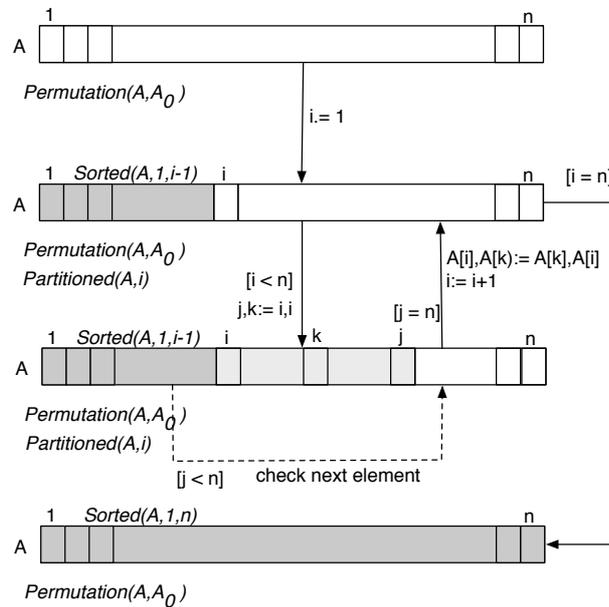


Fig. 5. Complete algorithm at work

We still need one more loop to find the smallest element in the remainder of the array. Again, we use figures as a tool to help us get an idea of how the algorithm works (Fig. 5). The corresponding invariant diagram (Fig. 6) represents our final program.

When all situations and transitions have been added to the diagram, we still need to check that no infinite execution loops exist, i.e. that the program *terminates*. We introduce a termination function (one for each loop), usually an integer function that is bounded from below and whose value is decreased before re-entering the situation. Moreover, the termination functions must be chosen so that no inner loop modifies the

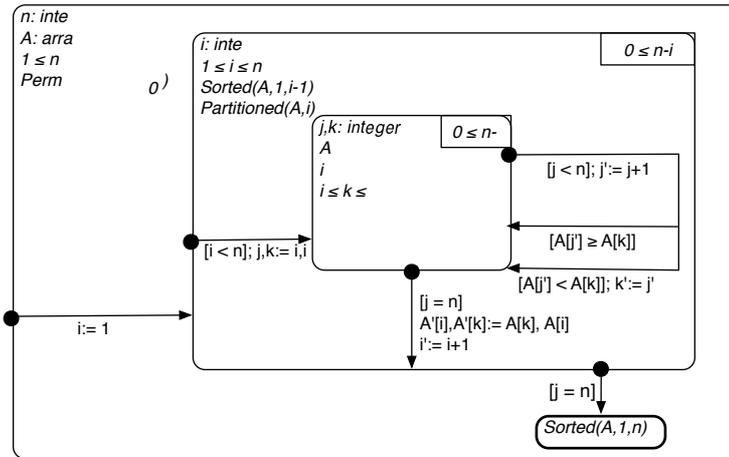


Fig. 6. Final invariant diagram

value of the termination function of an outer loop. The termination functions are written in the right upper hand corner of the respective invariant (Fig. 6).

Finally, we must check that the program is *live*, i.e. that termination only occurs in final situations. In practice, this means that we must make sure that for all situations in the diagram (except for the final ones) the available out-going transitions cover all possibilities.

An invariant based program is correct if it satisfies the three criteria above, i.e. it 1) is consistent, 2) terminates and 3) is live. For a more in-depth presentation of IBP as a method, see [6–8].

3.2 Tool Support for IBP

Invariant based programs can be constructed using only pen and paper, and in many cases this is the best way for initially drafting a program. However, even small programs generate a large number of verification conditions, many of which are rather trivial and can be automatically proved or greatly simplified by state of the art theorem provers. Additionally, the (considerable) risk of human error in manual proofs and specifications can be mitigated with proper tool support. Finally, we want to be able to execute the diagrammatic representation directly, without first having to hand translate it into some existing programming language.

SOCOS [9] is a graphical programming environment for the construction and verification of invariant based programs (Fig. 7). It analyzes invariant diagrams semantically, and generates correctness conditions which are sent to external proof tools (currently Simplify [17] and PVS [32] are supported). SOCOS also compiles invariant diagrams to executable Python code.

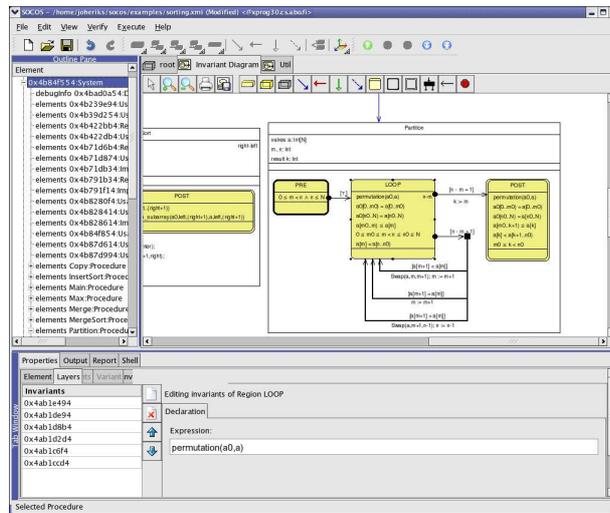


Fig. 7. The SOCOS IBP Environment.

4 IBP in Education

4.1 Motivation

The invariant property and the benefits from using it were presented in quite a natural and easily understandable way already in the original articles by Floyd [21] and Naur [28] on proving the correctness of computer programs. Introducing invariants early in the CS curriculum has been studied previously [2, 3, 20, 23], and the main message in all of these studies is that program correctness and loop invariants can be introduced at an early stage of CS education provided that the way in which it is done is adapted to the level of the students.

Starting in 2004, the development of IBP has been intertwined with informal experiments on teaching the method to see how it could be made more applicable in education. We have organized and observed 14 sessions with, in most cases, two CS students or academics having no prior experience of IBP. Each session started with an introduction to the approach, after which the participants were to solve a given problem using IBP on the blackboard. In spring 2005, a course on IBP was given to CS PhD students. These experiments have provided us with valuable feedback on the approach (positive experiences, difficulties, places for improvements etc) and two years later, the approach was deemed to be mature enough to be introduced at undergraduate level for the first time.

4.2 Undergraduate IBP Course

In spring 2007, an elective course covering the basics of IBP was developed and given to CS students at Åbo Akademi University (Turku, Finland). Our main motivation was

to address the common issues discussed in section 2 aiming at 1) changing the image of formal methods as being difficult (requiring highly advanced mathematics), uninteresting and of no use in practice and 2) showing that formal reasoning about program correctness can in fact be done in a practical manner with only fundamental logic skills. The goal of the course was for students to develop their programming and algorithmic thinking skills at the same time as learning about program correctness and formal reasoning. Another main design criteria was to make the course interesting and accessible so that it would inspire and motivate students to learn about correctness concepts.

The course is part of a project at our department aiming at designing a three course “course bundle” that would give all students a solid foundation in both the theory and practice of programming already during their first study year. The bundle includes, in the following order, a course covering “traditional” practical programming using a “simple” language (in our case, Python), a course on logic, and one that covers the mathematical foundations of programming (the IBP course). Together with the introductory course in mathematics (which introduces set theory that is needed for understanding the diagrammatic notation), the two former courses provide the students with all the background knowledge they need in order to successfully complete the IBP course.

The course on logic introduces *structured derivations*, which is a calculational proof format developed by Ralph-Johan Back and Joakim von Wright [10, 11]. They have extended the derivational style approach to proofs as presented by Dijkstra [19] and van Gasteren [36] by adding nested derivations (subderivations), allowing inferences to be presented at different levels of detail. The approach provides simple yet precise rules for how to write mathematical derivations and proofs that are easy to read as each step in the proof is clearly motivated. The goal of the course on logic is for students to learn 1) a clear format for writing well structured proofs that they know how to apply in practice and 2) basic propositional and first-order logic. Structured derivations are well suited for constructing proofs for invariant based programs, and using the same format in the IBP course was a natural choice.

Course Syllabus The course was given in an interactive format, emphasizing student activity throughout the course. All in all, the course included 17 sessions (90 minutes each) out of which 11 were used for lectures, and six for practical exercise. The main part of the course was taught without tool support; the SOCOS environment was only used in the four final sessions.

During each practice session, the students were to solve a set of IBP-related tasks, such as constructing a program, proving a certain transition or checking the correctness of a given program. Three of the assignments were reviewed collectively in class, whereas three were handed in and checked by the teacher who then gave detailed individual feedback for all tasks.

Integrating SOCOS in the Course Although the main part of the course was given without tool support, we felt a need to include SOCOS as the burden of organizing proofs quickly becomes noticeable even for relatively simple programs. Also, CS students are accustomed to using specialized software (e.g. compilers, interpreters, editors) in course work, and may regard programs and proofs constructed with pen and paper as

mere academic exercises. Actually being able execute the program may give the student some additional sense of accomplishment and thus act as a motivating factor.

Incorporating SOCOS in the course also made it possible for us to evaluate it in the context of teaching introductory formal methods, as well as to identify potential issues. The students were not expected to be familiar with PVS (mechanical theorem proving requires a separate course), so only the automatic prover was used in the course. In situations where the prover failed, students were required to complement the solution with a manual proof. The goal was to reduce the busy-work of proving simple, repetitive conditions, so that we were able to give more complex programs as exercises.

Examination The course examination consisted of active class participation, passed assignments and a final exam. The exam included programming problems similar to the ones in the assignments as well as questions that tested the students' understanding of invariant based programs. The students did not have access to the SOCOS environment on the exam.

5 The Study

Methodology The aforementioned studies on loop invariants in education ([2, 3, 20, 23]) include no evaluation of the approaches presented. Our goal was not only to present a new approach but also to evaluate its use and applicability in practice. We conducted a descriptive case study aiming at addressing the research questions presented in section 1, at the same time gaining insight into whether, and in that case how, the course and the method should be improved.

The study follows the principles of action research [14]. In action research, practitioners in a field improve practice by doing or changing something and reflecting on the results. The improvement can be in three areas: "improving a practice; improving the understanding of a practice [...] and improving the situation in which the practice takes place" [14, p.106]. The main purpose is to collect data and experience that help in gaining a better understanding of the practice.

Settings The undergraduate course was elective but still attracted 16 active participants (students that handed in at least one assignment). Nearly half were first or second year students with no background in formal methods. One of the students was absent for over half of the course due to medical treatment. Ten students participated in the SOCOS part of the course.

Data Collection Data were collected using pre- and post course questionnaires, observations, hand-in assignments and a final exam. Moreover, eight students were selected for semi-structured interviews one month after the exam. In this paper we will focus the analysis on the post course questionnaire, the assignments, the exam and the interviews. This mixed method approach with triangulation [25] was used to arrive at a multifaceted picture of the students' opinions and attitudes about the course in general

as well as the applicability of SOCOS. The use of different research instruments also increases the trustworthiness of the results, as it allows the researcher to look at the same phenomenon from several perspectives and thus arrive at a more complete account.

The post course questionnaire included both multiple choice questions and open ended ones asking the students about their opinions about the course in general as well as about IBP and SOCOS. In the multiple choice questions, Likert-type scales were used. Solutions to homework assignments were sent directly to the teacher by e-mail. Grading for the SOCOS assignments was based on the correctness of the solution (amount of verification conditions proved) and how precisely the pre- and postconditions were expressed.

The results reported on in this paper are based on the analysis of 12 post course questionnaires, 8 interviews and 13 sets of assignments and exams.

6 Results

6.1 Questionnaire Data

The answers to each open-ended question were first read and categorized as either positive or negative. In cases where the answer included both positive and negative aspects, the answer was divided into two parts accordingly. Next, all answers were reread and classified according to common themes representing the overall views of the twelve students (S1 - S12). The categories found with regard to what the students had experienced as 1) beneficial and useful, and 2) difficult in the course are listed below. Each category is exemplified with excerpts from the answers. The citations have been freely translated from Swedish by one of the authors.

Experienced Benefits

1. Introduction to program correctness and formal verification

Knowledge about proofs and correctness will hopefully lead to better programs (S2)

To learn a method for verifying programs formally (S7)

A good introduction to formal verification and how tools can be used in that context (S9)

Helps remove errors in the algorithm that could lead to bugs (S7)

2. A practical method for introducing program correctness

IBP seems to be a more practical verification approach than other methods I have seen (S4)

IBP summarizes the proof conditions in a good way (S4)

IBP is intuitive (S8)

3. Introduction to a more abstract view of programming

The course is about program design. You get a specification and design a correct program based on that (S3)

Learning to think about how a program works in general, without resorting to a given programming language (S3)

- Learned to think about a program as states and transitions instead of merely as transitions as is usually the case (S10)*
4. **More tangible overview of a program's structure**
 - Learning to draw a program makes it easier to see its structure (S12)*
 - Makes it easy to keep track of the various parts [pre- and postconditions, invariants] of a program (S4)*
 5. **The course arrangements**
 - Good teaching material, methods and lectures (S9)*
 - The assignments helped me learn (S11)*
 - All topics were thoroughly covered (S5)*
 6. **New and useful contents**
 - I learned something new (S8)*
 - The things I learned in the course will be useful in the future, especially in further studies (S9)*

Experienced Difficulties

1. **Syntax and notation**
 - It is difficult to formulate one's programs according to the standard (S8)*
 - Since I have programmed previously, e.g. the Java way of expressing things is quite ingrained (S3)*
 - Formulating the conditions in a way that makes it easier to prove the program (S4)*
2. **Proofs**
 - Proving programs by hand is very work intense (S4)*
 - Proving complex programs (S1)*
 - Proving programs 'honestly', i.e. to realize that one has made a mistake and correct it instead of trying to merely come up with explanations (S9)*
 - The formal proof conditions should have been introduced earlier in the course (S1)*
3. **Finding the appropriate conditions**
 - Finding the correct postcondition is most difficult. The difficulty of finding the invariant depends on how difficult it is to find the postcondition (S6)*
 - Finding the invariant in complex programs (S7)*

The quantitative data gathered in the questionnaire supported these qualitative findings. For instance, the course was found useful, interesting, somewhat fun and of medium difficulty level. On average, the data suggested that students found IBP rather easy to learn and useful in practice. The difficulties in constructing proofs and finding the invariant for more complex programs were also pointed out in the multiple choice questions. All students but one stated that they had enough mathematical skills for taking—and passing—the course.

Ten students attended the SOCOS part of the course and answered the related questionnaire. In line with our expectations the students preferred SOCOS over pen and paper, as the automation increases productivity. One student commented that it was “rather straightforward to understand the idea of the tool and how to apply it.” On the question whether IBP could be a practicable method in realistic software construction

the answers were scattered but still predominantly positive. Finally, the idea of supporting a formal method with a practical tool in the same course was very well received. The survey also indicates that unfamiliarity with the SOCOS syntax was the main cause of difficulty. Unfortunately, SOCOS lacks a good reference manual so teaching was mainly example-driven, and due to time constraints the students did not really achieve fluency in the SOCOS syntax.

The SOCOS related answers to the open ended questions supports the above mentioned findings, indicating that the tool was found useful, but somewhat difficult to use due to lack of time and an incomplete manual.

6.2 Assignments and Exam

The max score for the pen and paper assignments was 40, and the average was 25.5 (stdev 11.2). Seven students scored more than 30 points. Most errors were related to syntax (e.g. using Java like syntax) or the proofs not following the given format. The most common error related to program correctness was incomplete invariants, e.g. in the form of a missing lower or upper bound for a variable. A couple of students had problems with the algorithm, e.g. not updating variables to arrive at the result or writing a correct program that, however, was not the program asked for in the assignment. Some cases of using undeclared variables were also found. Merely one student seemed to have problems with the diagrammatic notation, writing the statements inside the situations instead of adjacent to the transition arrows. Only one “off by one” error was found.

Students who handed in solutions to the SOCOS assignments performed well. The highest scoring student achieved 20/20 points, while the average score was 14/20 points. Two students failed the exercise as a result of not handing in solutions—in one of these cases the student had been absent from the introductory sessions and subsequently lacked basic knowledge of the tool.

So far, 13 students have taken the exam,¹ out of which 11 have passed the course (four students with the highest grade). One of the two students who failed was the one who was absent for over half the course. As the goal of this paper is to describe how we have used IBP in education and report on the overall results, it does not contain any in-depth analysis of the students’ assignment and exam answers.

6.3 Interviews

Eight students (I1-I8) were interviewed by the lecturer one month after the exam. The interviewees were selected based on their course results in order for the interview data to be as representative as possible of the entire student group.

We chose not to conduct the interviews directly at the end of course as we wanted to have time to go through the other data first in order to construct interview questions based on the difficulties and other interesting points found in the other data. The process resulted in 12 broad questions that made it easy to ask follow-up questions when needed. The students were, for instance, asked about what they had learned and what

¹ At Åbo Akademi University, students have several alternative dates for taking the exam, and are thus not obliged to take part in the first ones that are arranged.

they had found difficult. They were also to describe the process of how they solve a problem using IBP and discuss how confident they are that the final program is correct.

The semi-structured interviews were transcribed and analyzed manually. All in all, the interview data strengthened the results found in the questionnaires. Students generally considered the IBP course a practical theory course quite different from other courses they had taken previously. The interactive nature of the class sessions was appreciated and the course considered suitable for first year university students. The interviewees were to describe how they typically solve a problem using IBP, and most of the descriptions followed the work flow presented in the course. Most students also said that they formally prove their programs after they have completed the diagram, whereas they rely on informal reasoning while while constructing the diagram.

Although the students found the invariant based approach per se useful, clear and simple, they did point out some difficulties, similar to those that were mentioned in the questionnaires. SOCOS was considered a helpful tool that, however, needs better user manuals and support. The students still pointed out the need for learning the fundamentals of IBP using only pen and paper.

7 Discussion

7.1 The Course and IBP in General

The feedback on both the course and IBP was in general quite positive. Students felt that IBP was easy to learn and the diagrammatic notation easy to understand. We were pleased to find that many students had recognized our original motivation for developing this course, that is, to present a practical method for introducing formal reasoning when constructing programs. Moreover, students also found that the approach made the general structure of the program more comprehensible.

We acknowledge that success in assignments and on exams is not a direct indicator of student learning, but we do feel that the programs written by the students on the exam and in the assignments show that they had understood the idea behind IBP and were able to construct and prove simple invariant based programs. These are the same students out of which many were not even able to explain basic concepts like “precondition” and “invariant” prior to the course.

The students clearly appreciated the diagrammatic notation of IBP. Studies [22, 34] estimate that between 75% and 83% of all students are visual learners, and because of their highly textual nature, the use of traditional programming languages or pseudocode is not necessarily the single best way for expressing algorithms to the majority of our first year students. As one of the IBP-students said in the post-course interview: *“Nice to see how a program really works. You saw it for yourself. And then you also understand the algorithm better when you see it in front of you. It’s more difficult to see what a program does directly from code.”*

Another benefit of using the invariant based approach is that it provides good support for finding bugs during the program construction (instead of after). This was also pointed out by the students in the interview. For example, we only found one single off by one error in the assignment solutions. Some of the other errors were related to the

use of undeclared variables. One could assume that writing out the type for a variable might easily be overlooked when writing the programs by hand as there is no compiler to check that the programs are correct (the SOCOS tool would naturally point out such errors). Thus, the students might simply have “forgotten” the declaration part when introducing a new variable in the program.

We had expected the students to find identifying the invariants the most difficult task, but this was not the case. Although some students mentioned the invariants, writing proofs by hand still seems to have been most problematic as they required much time and effort. The manual proofs do become rather long, for instance as all assumptions are written in each step of the derivation, but it is still interesting to see that students rate the difficulty of a given task based on how much time or effort it requires. Whether that is a reasonable indicator for the difficulty level of the task can be questioned. The format for the structured derivations has, however, been revised, and the modifications will automatically make the manual proofs less repetitive.

The questionnaire data pointed out the need for a clear standardized syntax. Students reported on sometimes finding it difficult to know how to express conditions and when to write their own definitions. This was to some extent expected, as the students had very little, if any, prior training in building their own domain theory. More practice and information about how to define predicates and reason about common data structures will therefore be included in the course from now on.

When designing the course, we thought it would be good to start by reasoning only informally about the correctness of the programs, before going further to formal mathematical proofs. This did, however, not turn out to be the case; instead, the students would have wanted the formal proof obligations to be introduced earlier. One explanation could be that students who are not mathematically mature do not know how to reason “informally” but first need to learn a formal approach with a set of rules.

7.2 Use of SOCOS

Incorporating computer aided verification in an introductory course was not an entirely uncontroversial choice. We were aware of the risk that students could apply the automatic prover as a magical tool and resort to a test-and-modify cycle (i.e., guess an invariant, guess transitions, run the automatic theorem prover, modify if the proof fails, ad nauseam). However, this risk was not manifested. Apparently the theoretical part of the course had given the students sufficient insight into the difficulty of the verification problem to realize that such testing would not converge into a correct program.

In line with Wing’s study [38], the students clearly appreciated the theory being complemented by a tool such as SOCOS. Surprisingly, most students learned to use the tool sufficiently well to solve the (non-trivial) exercises in the limited time available. Syntactical issues were the main cause of difficulty, largely due lack of documentation and occasional “rough edges” in the user interface. These usability issues are understandable and expected since the tool is experimental, and in our opinion do not indicate a fundamental flaw in the work flow.

Based on the open ended feedback, we have realized that there is a definite need for more extensive support in the form of documentation and manuals as well as personal guidance. Also, two weeks is far too little time to introduce and familiarize a verification

tool, which by nature contains considerable complexity. Both of these issues will be considered and rectified in sequel courses.

8 Conclusions and Future Work

Our initial experience from teaching IBP is positive, and we feel that our study has addressed the questions mentioned in section 1. The students appreciated learning about program correctness and seeing the programming activity from another perspective. IBP helps students further develop their programming skills at the same time as they learn how to reason formally about their programs. As opposed to the traditional separation between theoretical and practical courses that contrast formal and informal approaches, IBP integrates mathematics with the presentation of software design. Teaching IBP implies teaching all core topics in software design rather than a specific topic for which a dedicated formalism or tool exists. Moreover, the material is presented with minimal notational burden and builds upon students' previous knowledge (e.g. set theory). The use of SOCOS in the course was also appropriate; by automating trivial tasks, tool support enables the student to focus on difficult and interesting parts of the problem at hand. Furthermore, provided that the basics of invariant based programming are well understood, the exacting rigor of machine checked proofs considerably increases confidence in the correctness of the solution.

The study also shed light on some issues that need to be addressed. The syllabus will be modified according to the findings presented in this paper, for instance by including the formal proof conditions early. Moreover, a small and simple domain theory for array manipulation will be developed. In order to facilitate the construction of manual proofs, the preceding course on logic will be redesigned to better support the IBP course in providing the students with the skills needed to reason logically and write proofs using structured derivations. The format for the structured derivations has also been modified, and the changes will automatically shorten the proofs as all assumptions do not have to be written in every step of the proof. Additionally, the usability of SOCOS will be improved, the user manual will be further developed in order to become more comprehensive, and the proportion of the SOCOS part of the course will be considered and adjusted accordingly. The assignments and exam answers will also be thoroughly analyzed in order to find any indications of difficulties or problems that need to be considered.

Encouraged by the results from this pilot study, a course covering the basics of IBP will be offered annually to CS students at our department starting in the upcoming academic year (2007/2008), complementing the preceding courses on practical programming and logic. Our aim is thus not to substitute IBP for traditional programming courses. Quite on the contrary, we acknowledge the need for "traditional" programming, testing and debugging. We do not, however, see any reason for why these approaches could—or should—not coexist. We recognize that one single course is not enough for bridging the gap between theory and practice in the CS curriculum. If the students are to truly benefit from the skills acquired in the IBP course, these should be built upon in upcoming courses. A discussion with other CS faculty is thus essential in order to guarantee that there is a joint agenda on this point. We are also planning on developing

followup courses on this topic, for instance covering mechanical verification and application domain theory. By doing so, we aim at introducing a continuum in students' exposure to formal reasoning throughout their education.

References

1. Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, 2001.
2. David Arnow. Teaching programming to liberal arts students: using loop invariants. *SIGCSE Bull.*, 26(1):141–144, 1994.
3. Owen Astrachan. Pictures as invariants. In *Proc. of the 22nd SIGCSE symposium*, pages 112–118, New York, NY, USA, 1991. ACM Press.
4. Ralph-Johan Back. Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.
5. Ralph-Johan Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.
6. Ralph-Johan Back. Invariant based programming revisited. Technical Report 661, TUCS - Turku Centre for Computer Science, Turku, Finland, 2005.
7. Ralph-Johan Back. Invariant based programming. In *ICATPN*, pages 1–18, 2006.
8. Ralph-Johan Back. Invariant Based Programming: Basic Approach and Teaching Experiences, 2007. Accepted for publication in *Formal Aspects of Computing Science*.
9. Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Verifying invariant based programs in the SOCOS environment. In *Teaching Formal Methods: Practice and Experience (BCS Electronic Workshops in Computing)*. BCS-FACS, Dec 2006.
10. Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculation proof. *Formal Aspects of Computing*, 9:469–483, 1997.
11. Ralph-Johan Back and Joakim von Wright. A method for teaching rigorous mathematical reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, Aug 1999.
12. Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.
13. Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, (15):95–114, 2001.
14. Tony Clear. Critical enquiry in computer science education. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 101–125. Taylor and Francis Group, 2004.
15. Richard Denman, David A. Naumann, Walter Potter, and Gary Richter. Derivation of programs for freshmen. In *Proc. of the 25th SIGCSE symposium*, pages 116–120, New York, NY, USA, 1994. ACM Press.
16. Peter J. Denning. A debate on teaching computing science. *Commun. ACM*, 32(12):1397–1414, 1989.
17. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
18. Edsger W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, Dec 1989.
19. Edsger W. Dijkstra and Carel S. Scholten. Predicate Calculus and Program Semantics. *Texts and Monographs in Computer Science*, pages 21–29, 1990.

20. David Evans and Michael Peck. Inculcating invariants in introductory courses. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 673–678, New York, NY, USA, 2006. ACM Press.
21. Robert Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
22. Lynne Fowler, Maurice Allen, Jocelyn Armarego, and Judith Mackenzie. Learning Styles and CASE Tools in Software Engineering. February 2000.
23. David Ginat. Loop invariants and mathematical games. *SIGCSE Bulletin*, 27(1):263–267, 1995.
24. David Gries. Teaching calculation and discrimination: a more effective curriculum. *Commun. ACM*, 34(3):44–55, 1991.
25. Andrew P. Johnson. *A short guide to action research*. Allyn & Bacon, Boston, 3 edition, 2008.
26. Henry McLoughlin and Kevin Hely. Teaching formal programming to first year computer science students. In *Proc. of the SIGCSE symposium*, pages 155–159, 1996.
27. Kirby McMaster, Nicole Anderson, and Brian Rague. Discrete math with programming: better together. In *Proc. of the 38th SIGCSE symposium*, pages 100–104. ACM Press, 2007.
28. Peter Naur. Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4):310–316, July 1966.
29. J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.
30. Geoffrey G Roy. Designing and explaining programs with a literate pseudocode. *J. Educ. Resour. Comput.*, 6(1):1, 2006.
31. Abhik Roychoudhury. Introducing model checking to undergraduates. In *Formal Methods Education Workshop*, pages 9–15, 2006.
32. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.
33. Ann E. Kelley Sobel. Empirical results of a software engineering curriculum incorporating formal methods. In *Proc. of the 31st SIGCSE symposium*, pages 157–161, New York, NY, USA, 2000. ACM Press.
34. Lynda Thomas, Mark Ratcliffe, John Woodbury, and Emma Jarman. Learning styles and performance in the introductory programming sequence. In *Proc. of the 33rd SIGCSE symposium*, pages 33–37, New York, NY, USA, 2002. ACM Press.
35. M. H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, SE-5(2):148–159, 1979.
36. Antonetta J.M. van Gasteren. On the Shape of Mathematical Arguments. *Lecture Notes in Computer Science*, pages 90–120, 1990.
37. J. Stanley Warford. An experience teaching formal methods in discrete mathematics. *SIGCSE Bull.*, 27(3):60–64, 1995.
38. Jeannette M. Wing. Weaving formal methods into the undergraduate curriculum. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 2–7, May 2000.

Paper IX

Invariant Based Programming in Education – An Analysis of Student Difficulties

L. Mannila

Originally published in *TUCS Technical Reports*, number 944. Turku Centre for Computer Science, April 2009. Accepted for publication in *Informatics in Education*.



Invariant Based Programming in Education – An Analysis of Student Difficulties

Linda Mannila

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
`linda.mannila@abo.fi`

TUCS Technical Report

No 944, April 2009

Abstract

In this paper, we analyze the errors novice students make when developing invariant based programs. In addition to presenting the general error types, we also look at what students have difficulty with when it comes to expressing invariants. The results indicate that an introductory course utilizing the invariant based approach is suitable from the very beginning of university studies in CS without being "too advanced". Although inventing the invariant was not found to be trivial, the main difficulty faced by novices when applying a correct-by-construction approach to program development seems to be related to weak skills in translating intuitive and informal statements into a symbolic form using logical notation in general and quantifiers in particular.

Keywords: invariant based programming, programming education, introductory formal methods, student difficulties

TUCS Laboratory
Learning and Reasoning

1 Introduction

The concept of loop invariants was introduced by Naur [25] and Floyd [18] already in the 1960s. Continuing on their work, Hoare [21] defined inference rules for correctness proofs based on loop invariants. This development lay the basis for Dijkstra's [15] work on outlining a new programming methodology for writing programs with built-in correctness proofs. Half a decade later, Gries [20] wrote a textbook on using this methodology aimed at education. Dijkstra also discussed the question of how much formality to include in the computer science (CS) curriculum in the paper "On the cruelty of really teaching computer science" [16], a paper which was followed by many responses collected in "A debate on teaching computing science" [14].

Regardless of the rather early developments, loop invariants and formal techniques are commonly underutilized in introductory CS courses. There are many other reasons for this. First, these topics are typically considered difficult, requiring prerequisite knowledge of advanced mathematics and logic, which students simply do not have [31]. As a result, different, some more informal, approaches for introducing invariants and program correctness concepts in education have been presented (see e.g. [2, 3, 17, 19, 31]). Unfortunately, we have not found any empirical evaluation of these approaches in the literature.

Moreover, most CS curricula treat the teaching of programming and the teaching of more formal courses as separate disciplines, where, for instance, logic is viewed as a separate "add-on" instead of as an integral part of programming. Consequently, students get only little exposure to correctness concepts [1]. When formal techniques are taught as an activity independent from the programming process, students get the impression that formal techniques are only applicable in theoretical courses [24]. This in turn gives rise to attitude problems, as students do not see any point in having to take more theoretical courses. Dijkstra described this as "mental resistance" among students [20].

Reed and Sinclair [26] suggest that another reason for students' resistance may be found in the prevailing culture where students want and are used to getting quick results; a hacker mentality clearly does the job much faster than one aiming at verifying the correctness. Moreover, CS students seem to be driven by external factors, and learning the skills that they believe are relevant and are needed to earn money may make it difficult to motivate the study of other topics. Finally, the challenges can be traced back to the actual teaching situations. Most textbooks on introductory programming, for instance, do not discuss program correctness or concepts such as loop invariants [3]. In addition, it is common for CS faculty to argue against formal methods [30].

Although attempts at introducing formal methods in education have been made and the importance of more rigorous formal reasoning in CS has been emphasized in curricula and other recommendations (see e.g. [22, 23]), it seems safe to say that convincing students of the value of formal techniques is everything but easy.

Starting in 2007, we have been evaluating an approach to teaching “practicable formal methods” in a course for CS novices. We refer to this approach as *invariant based programming* (IBP). IBP is a visual program construction and verification methodology, which introduces a minimum of notational overhead and allows students to reason about correctness using mathematical concepts with which they are already familiar (such as set theory and basic logic). Through this course we aim at addressing several of the challenges discussed above: changing the image of formal methods as difficult while at the same time reducing the gap between theory and practice. Issues related to lack of teaching material or teacher attitudes are such that we as educators and method developers can address. Overcoming students’ mental resistance and changing their cultural preferences are however things that one could believe are not easily accomplished during a single course. Our experience from the course has, however, been positive, indicating that students find it fun and useful. In addition, they seem to appreciate learning about program correctness and seeing the programming activity from another perspective. [9]

For this paper, we have conducted a thorough analysis of student created invariant based programs, focusing on the errors made. The aim is also to bring light on to what extent IBP requires knowledge and skills falling under the “too advanced” section. The study seeks to investigate three research questions:

1. What kind of errors do novices make when using IBP?
2. Do these errors change as the course progresses, and if so, how?
3. Does the year of study impact student performance?

The paper is organized as follows. Next, we describe the invariant based approach, after which we present the study design. In the following two sections, we put forward and discuss the results. The paper is concluded with some final words and ideas for future work.

2 Invariant Based Programming

IBP is an approach to constructing correct programs, where not only pre- and postconditions, but also loop invariants are to be written before doing any coding. The approach is not new — it was studied already in the 1970s by Reynolds [27] and Back [4, 5]. Similar ideas were also proposed by van Emden [33]. In 2004, Back [6] revisited the topic and has since worked on developing IBP into a practical hands-on method.

In IBP, a program is constructed and verified at the same time. The notion of an invariant is generalized to a *situation*. Each situation is a collection of constraints and describes the set of states that satisfy these constraints. Thus, a loop invariant is a situation, as well as a precondition or a postcondition. An invariant based

program may have many situations and is not restricted to single-entry, single-exit control structures.

In essence, IBP provides a visual representation of a program. A variety of graphical programming/pseudocode formats have been proposed in the literature [11, 29], and all of these have one common goal: “to provide a clear picture of the structure and semantics of the program through a combination of graphical constructions and some additional textual annotations.” [29, p.3] To our knowledge, these have, however, focused on representing control flow and data flow. IBP, on the other hand, describes programs from another perspective as it emphasizes the invariant properties of the program data structures, and thus makes it possible to reason about the correctness of the constructed program in a rather straightforward manner. This is accomplished without sacrificing either clarity or expressiveness of the diagrams.

2.1 An Illustrating Example

We will here exemplify the work flow for developing invariant based programs by constructing a program that implements the selection sort algorithm. We use a cursor to traverse an array from left to right, and for each position we find the smallest element to the right of the cursor and swap that element with the one pointed at by the cursor. After each swap the cursor is advanced, and the array is sorted when the entire array has been traversed.

We start the process by drawing figures illustrating the basic data structures involved and how their values will change during execution of the algorithm. This is an essential step of the IBP work flow as the figures describe the algorithm at work and thus help the programmer get a feeling for the behavior of the algorithm. As this example illustrates, the figures also aid in identifying the situations (invariants) of the program.

The first figure (Figure 1) illustrates the specification (the pre- and postcondition), which helps us identify the initial and final situations. As situations are sets of states, the final situation is a subset of the initial one where an additional constraint, $Sorted(A, 1, n)$, is satisfied.

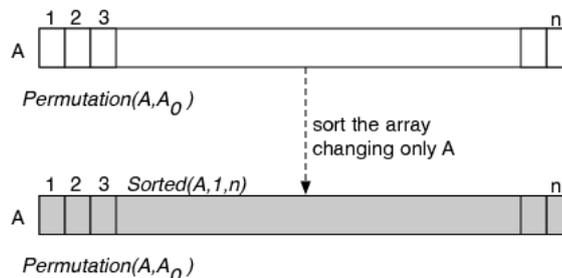


Figure 1: Visualization of the specification

We use an Euler-like diagram, a *nested invariant diagram*, to represent the program and the strengthening of situations. Our first diagram is shown in Figure 2. Due to the nesting, all constraints in an outer set implicitly also hold in all of its subsets and need therefore not be repeated (for instance, $n : integer$ holds in both the initial and the final situation). Dashed arrows are used to indicate the computation that we want to define and are labeled with a potential guard and the variables that may be changed in the computation.

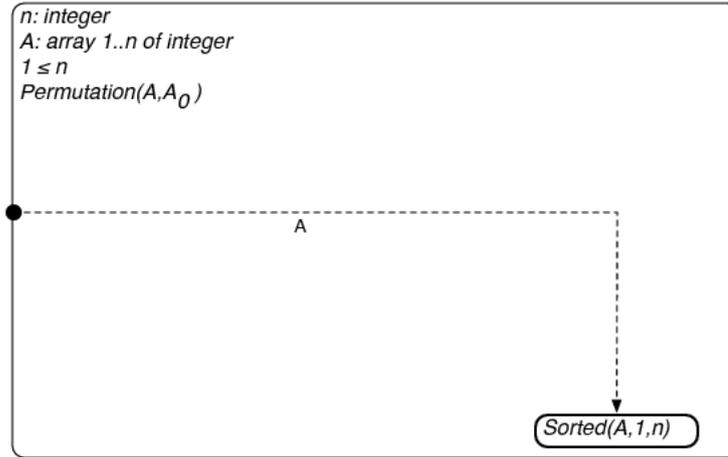


Figure 2: Corresponding invariant diagram illustrating the initial and final situations

In the same manner as the final situation was identified as a subset of the initial one, we introduce new situations by adding new constraints to the ones present in the more general situations. We further develop the figure of the algorithm at work by introducing the intermediate situation (Figure 3).

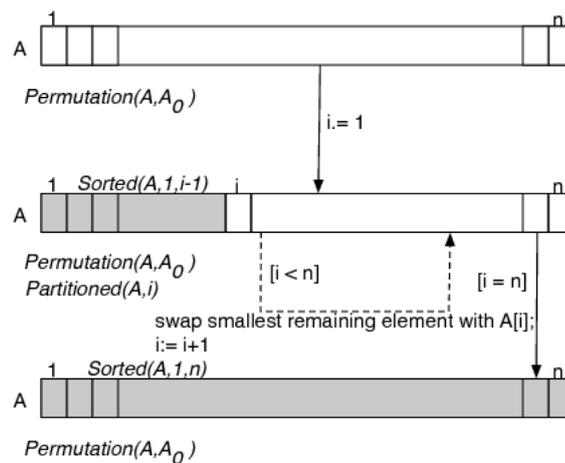


Figure 3: Sorting program with one loop

As is shown in the corresponding diagram (Figure 4), this newly inserted situation is a subset (i.e. a constrained version) of the initial situation.

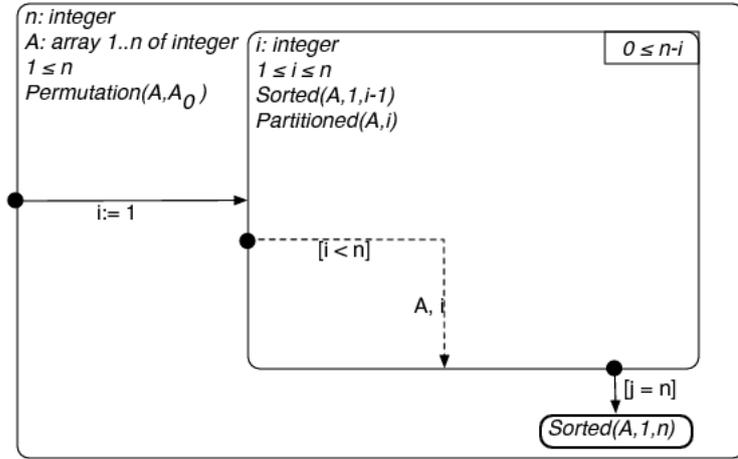


Figure 4: Invariant diagram with the intermediate situation inserted

Whereas dashed arrows illustrate what we want to accomplish, we use solid ones to indicate computations that we have already planned and defined. We call these solid arrows *transitions*. Each transition is labeled with a potential guard and the program statements executed when the transition is carried out.

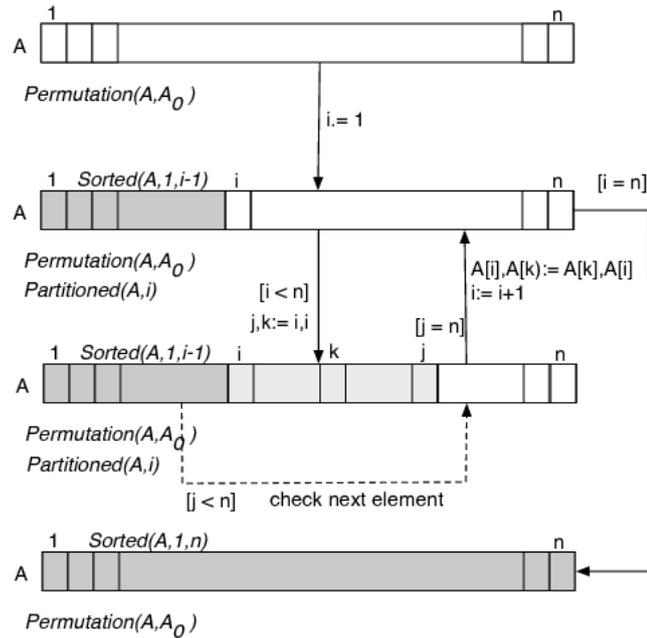


Figure 5: Complete algorithm at work

For each transition that we add to the diagram, we check that it preserves the

situations as follows: assume that we initiate execution in the source situation of a transition and that all the constraints hold for the starting state. Also assume that we reach some target situation after executing the statements for the transition (there may be more than one possible target situation). Then all the constraints of the target situation must hold for the final state. We say that a program is *consistent* if this is true for all transitions in the diagram.

We still need one more loop to find the smallest element in the remainder of the array. Again, we use figures as a tool to help us get an idea of how the algorithm works (Figure 5). The corresponding invariant diagram (Figure 6) represents our final program.

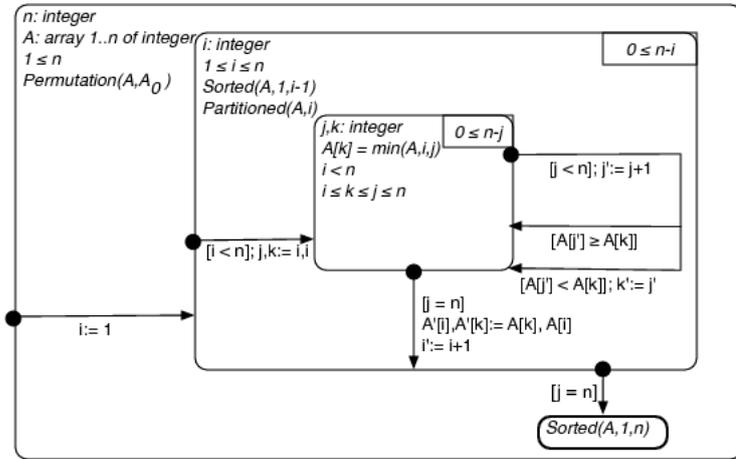


Figure 6: Final invariant diagram

When all situations and transitions have been added to the diagram, we still have to check that no infinite loops exist, i.e. that the program *terminates*. We introduce a termination function for each intermediate situation. A termination function is an integer function that is bounded from below and whose value is decreased before re-entering the situation. The termination functions are written in the right upper hand corner of the respective invariants (Figure 6).

Finally, we must check that the program is *live*, i.e. that termination only occurs in final situations. In practice, this means that we must make sure that for all situations, except for final ones, there is at least one exit transition that terminates normally.

An invariant based program is correct if it satisfies the three criteria above, i.e. 1) is consistent, 2) terminates and 3) is live. For a more in-depth presentation of IBP as a method, see the articles by Back [6, 7, 8].

3 Study Design

The study presented in this paper is part of a larger developmental research project [28, 32], in which we empirically evaluate the use of three methods for teaching introductory CS courses. The empirical evaluations act as a feedback mechanism for making improvements to the methods or changes to the ways in which the methods are used in education.

3.1 Data Collection

A course covering IBP was introduced at the Department of Information Technologies at Åbo Akademi University in Turku, Finland in spring 2007. The data for the study reported on in this paper have been collected during 2007-2008, when 23 students completed the course.¹ Half of the students were on their first or second study year, whereas the other half had studied for three years or more. Most students were CS or software engineering majors.

The course includes 34 hours in class and has been given in an interactive lab lesson format. Approximately a third of the time has been used for hands-on training. During these exercise sessions, students solve assignments which are later handed in for grading and feedback. At the end of the course, all participants take a final exam. The exercise sessions and the exam include different types of assignments:

- “Look at this invariant diagram and explain what it does.” (*reading*)
- “Modify this invariant diagram so that the program does X instead of Y. Prove that the resulting program is correct.” (*reading and modification*)
- “Find the corresponding imperative program for this invariant based one.” (*reading*)
- “Construct an invariant based program that does X. Prove that the program is correct.” (*construction*)

In this study, we focus on evaluating students’ difficulties when constructing programs, and will thus only consider the construction part of the final type of assignment. All in all, we have analyzed solutions to six assignments for 23 students. Two of the solutions were solved at the beginning of the course (Exercise set 1), two towards the end (Exercise set 2) and the final two on the exam (Exam). Nine solutions were missing, giving us a total of 129 analyzed solutions (out of 138).

In the following, we briefly describe and exemplify the types of assignments included in the exercise sets and the exam. The assignments were of increasing difficulty level, i.e. those included in the first exercise set were the easiest, while the ones in the second set and on the exam were more difficult.

¹In this context, completing the course stands for “handing in at least 50 % of all assignments and participating in the final exam”.

Exercise set 1: In these assignments, students were provided with readily defined predicates and there was no need to use quantified expressions. The following is an example of this type of exercise.

Construct a correct invariant based program² that checks if an integer ($n > 0$, given) is odd and returns the result as a boolean value (True if the integer is odd, False otherwise). You are only allowed to use addition. To simplify your work you can define the predicate $isOdd(x) = x \bmod 2 \neq 0$ to describe what it means that an integer is odd.

Exercise set 2: In these assignments, students needed to define predicates themselves and also use quantified expressions, for instance as in the following:

Construct a program that substitutes the number one (1) for all odd numbers in an array. The even numbers should be kept unchanged. Use the mod operator ($x \bmod n$) to decide if a number is odd or even.

Exam: In these assignments, students needed to define predicates and use quantified expressions.

*Construct a program that calculates the sum of every m^{th} integer between 0 and n . Assume that n and m are given integers ($n \geq 0$, $m > 0$). I.e. if $m = 3$ and $n = 10$, the following integers should be summed up: 0 1 **2** 3 4 **5** 6 7 **8** 9 10 (sum = 15).*

One of these assignments also included nested loops:

A polynomial is an expression of the format $a_0 + a_1x^1 + a_2x^2 + \dots + a_{n-1}x^{n-1} + a_nx^n$. Construct a program that calculates the value of the polynomial given a non-negative integer n , the value of x and an array a containing the coefficients a_0, a_1, \dots, a_n ($a[0] = a_0, a[1] = a_1, \dots, a[n] = a_n$).

If we had the precondition $n = 3 \wedge x = 4 \wedge a = [2, 3, 0, 5]$, the results of the calculation should be 334 ($2 + 3 \cdot 4^1 + 0 \cdot 4^2 + 5 \cdot 4^3$). Note! You do not have access to the exponential function in the calculations, so you will need to come up with another way to calculate the powers (e.g. in a nested loop).

²In the following assignment specifications the word “program” refers to a correct invariant based program.

3.2 Method

When analyzing student solutions, we are dealing with rich data. In order to be able to interpret such data, it first needs to be reduced. In this study, a content-analytical approach was chosen for this purpose.

The basic idea of content analysis is to take textual material and analyze, reduce and summarize it using emergent themes. These themes can then be quantified, and as such, content analysis is suitable for transforming rich data into a form which can be statistically handled and analyzed. [12]

The content analysis was done in two phases. First, a subset of the student programs were analyzed one at a time by comparing these to the problem specification. Each solution could contain no, one or several error types. During this initial analysis, all errors were listed, resulting in 14 different error types. In order to further organize and reduce the data, detailed types were combined into higher level error categories. This resulted in eight (8) categories.

Based on these categories, a coding scheme was created. This coding scheme was then used in the second phase to analyze all assignments, resulting in an overview of the errors each student made in the different assignments.

The results of this second round analysis indicated a need for making a more thorough analysis of the problems related to the invariant. Therefore, all solutions, in which an invariant related error had been found, were revisited. A process similar to the one described above was initiated to find the different types of errors made by the students when attempting to find and express the invariant. This resulted in a new scheme containing nine (9) error types for invariants.

Whereas questions looking to describe a phenomenon are best answered using a qualitative approach, quantitative methods are better at addressing more factual questions [12]. Hence, when the initial (quite qualitative) analysis had been completed, a quantitative approach was taken in order to present and statistically analyze the data. We also wanted to find out whether there was any difference between the exam performance of novices (students on their first or second study year) and students having studied for a longer period of time. The *Kolmogorov-Smirnov* test showed that the exam data were not normally distributed, and hence the non-parametric *Mann-Whitney U* test was used in the analysis.

4 Results

4.1 General Error Types

In this subsection, we address the first two research questions: "What kind of errors do novices make when using IBP?" and "Do these change as the course progresses, and in that case how?"

The analysis revealed eight main error types, which are presented and exemplified in the following.

- **Updates:** Missing or redundant update. The student may, for instance, have forgotten to update the variable containing the result when finishing execution and moving to the final situation. Redundant updates do not make the program erroneous, but were still considered an error as they are unnecessary.
- **Guards:** Missing guard or incorrect bounds.
- **IBP notation:** Issues with the IBP syntax. The student may, for instance, have left out some brackets surrounding guards, set boundaries or a transition arrow. Another version of this error was the use of a *Java*-like syntax for expressing transition actions.
- **Logical notation:** The student had problems expressing situations correctly using logical notation. Note that this error type does not include quantifier related errors found in the invariant (these are included in the “invariant” error).
- **Invariant:** The analysis revealed the invariant related error type to be a multifaceted one; hence a more thorough analysis of these errors was conducted. The results from this analysis are discussed in section 4.2 below.
- **Postcondition:** Erroneous or imprecise. The student may have used an incorrect postcondition or expressed it carelessly.
- **Terminating function:** Missing or erroneous.
- **Algorithm:** The student may have constructed a program solving another problem than the one intended or taken shortcuts in the algorithm (e.g. not storing the result of a calculation in a variable, but instead assuming it was kept in memory).

The frequency of error types in the two exercise sets and on the exam is illustrated in figure 7. As the assignments were different, it is naturally not possible to make an absolute comparison of the error frequencies in them. The error frequencies do show the trend of how the errors made by the students developed as the course progressed and the assignments became more difficult. Evidently, the frequency of all error types decreased from the beginning to the end.

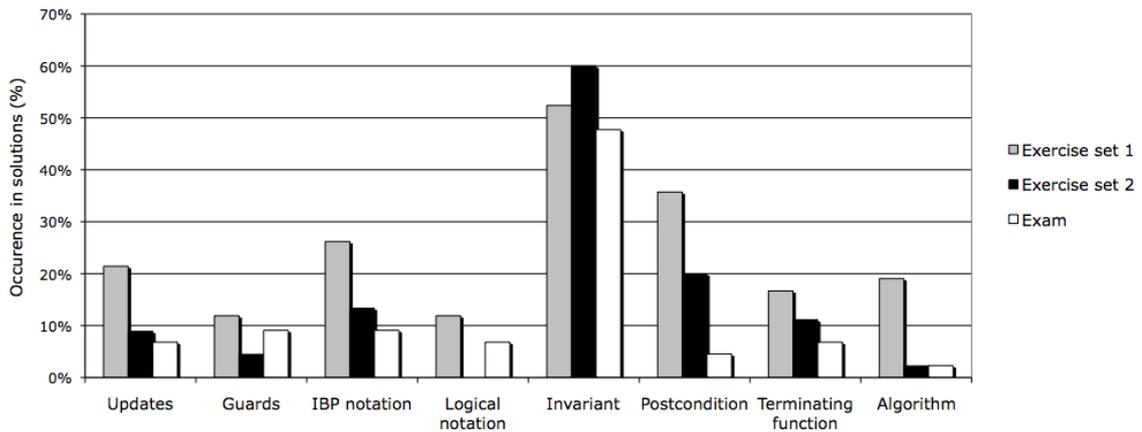


Figure 7: Occurrence of general error types in the two exercise sets and on the exam

4.2 Invariant Related Error Types

As the diagram in Figure 7 shows, invariant related errors were most common throughout the course, and as mentioned above, this called for a deeper analysis in order to investigate where the main difficulties lie.

The results from this analysis revealed that the invariant related errors were indeed quite multifaceted, as nine error types were found.

- **Quantifier syntax:** Malformed quantified expression.
- **Quantifier bounds:** Incorrect bounds in a quantified expression.
- **Variable bounds:** Missing, incorrect (“off-by-one”) or incomplete (e.g. lacking upper bound) bounds for a declared variable.
- **Variable declaration:** Use of an undeclared variable.
- **Logical notation:** Imprecise logical notation not related to quantifiers.
- **Incomplete:** An essential part of the invariant is missing.
- **Missing relationship:** The invariant includes all essential information, but lacks the relationship between a variable and a describing predicate. For instance, stating only $sorted(A, 0, k)$ instead of $isSorted = sorted(A, 0, k)$.
- **Strong:** A part of the invariant is too strong, for instance, stating that a variable is a constant (e.g. $isSorted = False$) instead of expressing it in terms of a predicate.
- **Erroneous:** Where the student had written a program solving the wrong problem, the invariant naturally also was incorrect. A variation of this error

was demonstrated in solutions where the student apparently had not been able to come up with a sensible invariant.

As in the previous subsection, we illustrate the occurrence of these error types for the exercise sets and the exam separately using a diagram (Figure 8) .

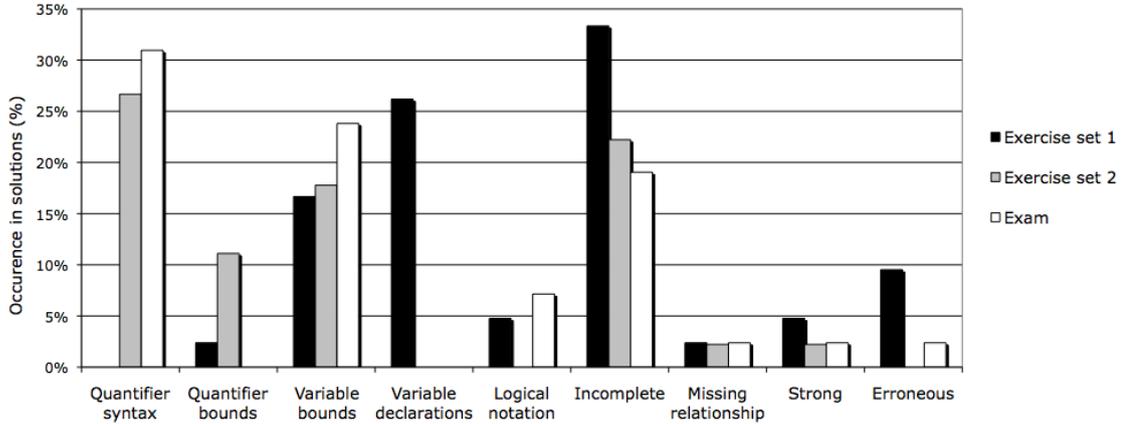


Figure 8: Occurrence of invariant related error types in the two exercise sets and on the exam

These nine error types can further be classified as *severe* or *less severe*. For instance, one could argue that using an incorrect syntax in a quantified expression is not a severe error, as long as it is possible for the human reader to interpret what the student has meant and this underlying meaning is sound. Using a 100% correct syntax when using the quantifier is not necessary to accomplish this. Similarly, one could claim that variable declarations are easily overlooked when writing programs by hand. This type of errors would not necessarily get caught when writing formal proofs by hand.

Incomplete, too strong or erroneous invariants on the other hand can be classified as severe, as these are directly related to the student’s ability to find a suitable invariant. Likewise, bound errors indicate that the student has not done a good job at verifying the program, because such misses would easily have been found during the verification process. Based on this distinction, we get the following classification:

- **Severe errors:** related to the invariant (incomplete, strong, erroneous, quantifier bounds, variable bounds)
- **Less severe errors:** notational (quantifier syntax, logical notation, missing relationship) or careless (missing variable declarations)

As is shown in the diagram, the frequency of most severe errors decreased as the course progressed. The only exception is for errors related to variable bounds,

which showed an increasing trend. A similar development can be seen for the less severe “notational” errors, where the increase in errors related to quantifier syntax is especially eye-catching. No careless errors (missing variable declarations) were found after the first exercise set.

Of the 23 students, 12 demonstrated a decrease in invariant related errors when comparing their solutions in the first exercise set to those written on the exam. For seven (7) students, the number of invariant related errors increased, whereas it remained unchanged for the remaining four (4). Unfortunately, one of the assignments in Exercise set 2 contained a subtlety resulting in many students’ invariants being incomplete. In order to get a sufficiently strong invariant, an unchanged property that was not easy to “see” needed to be stated. If these errors had not been taken into account, the percentage of incomplete invariants for the second exercise set would have been 9% instead of 22%. On the exam, all occurrences of incomplete invariants except one were found in an assignment involving nested loops; apparently, either invariant tends to become sloppily expressed in such programs.

Finally, we also wanted to examine how the proportion of completely correct invariants would change if only considering the severe errors. If looking at all errors (both the severe and the less severe ones), totally correct invariants were found in 31% of student programs in the first exercise set, with a slight increase in the second set and on the exam. If, however, only considering the severe errors, correct invariants were found in 45%, 53% and 60% for the exercise sets and the exam respectively.

4.3 Exam Performance

Our final research question aimed at investigating whether “older” students performed any differently in the course compared to the novices (students on their first or second study year).

The maximum score on the exam was 30, out of which 15 points were required to pass the exam. The average score was 21.2 (std dev = 4.82), while one student failed (13 points). The Mann-Whitney U test indicated no difference in exam performance ($U=50$, $Z=-0.992$, $p < 0.05$). Similarly, the content analysis showed no difference in error frequencies or error types between the two groups.

5 Discussion

The results indicated a decreasing trend for all general error types comparing the situation at the beginning of the course to that on the exam. Although the invariant related errors seemed to be a problem throughout the course, the more detailed analysis of these resulted in some interesting findings. When dividing the invariant related errors into two groups (severe and less severe), we found that the

number of most severe errors decreased as the course progressed. The decrease in invariant related errors implies that students became more proficient at finding the invariant, even if the problems to solve became more difficult. This is an encouraging result, as one of our initial suspicions when introducing the approach in education was that the largest difficulties would be related to identifying the invariant.

With this general trend for the severe errors, the increase in errors related to the variable bounds seems somewhat counterintuitive. Why would students become better at more difficult things (finding the invariant) while at the same time start doing worse on aspects that are quite easily checked? Given the way in which the programs were written, one could argue that the bound errors should be seen as careless errors. After all, there was no interpreter or compiler checking if a variable had been declared or if it had been given the correct bounds. Under pressure, these are “smaller details” that may be easily overlooked. This is especially true for the exam, where the students did not have the time to do proper proofs.

Looking at how the number of solutions with correct invariants would change if only considering the severe errors, it became clear that the less severe errors made up a substantial part of the errors. Although these are not as serious as the other errors, their occurrence accentuates a need to further stress the importance of going through and proving each transition separately, if not by writing a formal proof, at least informally by checking that all properties hold. Without explicit checks, formal or informal, the invariant approach is no more “safe” than the traditional “trial-and-error” approach to program development.

In addition, the results indicated that many students are weak at formalizing informal statements using logic. To be able to, for instance, describe situations precisely and correctly, students need to be able to move confidently between informal (e.g. situations illustrated in the figures) and formal representations (the corresponding logical expressions). Apparently, predicate logic and quantifiers are particularly difficult to students. This finding also serves as a partial explanation to why students find proofs difficult [9]. How could a student construct a correct proof if he or she does not know how to express and manipulate the verification conditions formally and precisely? Consequently, it seems crucial that CS students were given more training in translating informal statements into symbolic form.

A tool could help in addressing both the issue of careless errors and that of logical notation and syntax. *SOCOS*³ [10] is a graphical programming environment developed within our research group at Åbo Akademi University for the construction and verification of invariant based programs. It analyzes invariant diagrams semantically, and generates correctness conditions which are sent to external proof tools. Using *SOCOS*, trivial verification conditions can thus be proved or simplified automatically. In addition, it also compiles invariant diagrams to executable Python code.

³<http://mde.abo.fi/confluence/display/SOCOS>

A beta version of *SOCOS* was used in the course, and based on the initial teaching experiences and student feedback, the tool is currently being further developed to better suit the needs and skill levels of novice CS students. Nevertheless, we still believe that it is essential to introduce the approach using pen and paper only. Learning to build invariant based programs and constructing the corresponding proofs by hand is important in order for students to become familiar with the approach. The hands-on experience also makes explicit the link between mathematics and programming and shows that proving programs manually is a tedious and time demanding process; knowing the effort that goes into these activities, students can appreciate tool support to a larger extent later on.

The results also indicated that the IBP notation does not pose a problem to students, as only few such errors were found. Most of the notational errors found were due to students using a *Java* like syntax when expressing transitions and invariants. Given that the students already “knew” how to program, they already had an “ingrained” syntax. Thus, it is understandable that such students may face initial difficulties in abandoning their old approach [13]. Choosing to use a programming language like notation may also be closely related to the problems with using logical notation discussed above; the programming language syntax may be the only formal notation they feel confident in using.

Finally, the findings presented in the previous section indicate that IBP is just as suitable for novices as for students who have studied for a longer period of time.

6 Concluding Remarks

In this paper, we have investigated the difficulties that students face when learning IBP. Although inventing the invariant was not found to be trivial, the main difficulty seems to be related to a lack of skills in formalizing expressions and interpreting expressions written using logical notation. Problems related to constructing program correctness proofs also appear, at least to some extent, explainable by the lack of these very same skills. In order to successfully use the invariant based approach, more attention thus needs to be put on appropriate training aimed at developing these skills – as early as possible.

Apart from the problems with logical notation, we have not found any indication that IBP would be “too advanced” for CS students during their first or second study year. Although the invariant can be tricky to come up with, our experience shows that this is not beyond the capability levels of novices. Taken together with the positive attitudes among students towards the approach presented in [9], IBP seems to be a worthwhile alternative method for introducing a more formal approach to program development early in CS education.

References

- [1] Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, 2001.
- [2] David Arnow. Teaching programming to liberal arts students: using loop invariants. *SIGCSE Bull.*, 26(1):141–144, 1994.
- [3] Owen Astrachan. Pictures as invariants. In *Proc. of the 22nd SIGCSE symposium*, pages 112–118, New York, NY, USA, 1991. ACM Press.
- [4] Ralph-Johan Back. Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.
- [5] Ralph-Johan Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.
- [6] Ralph-Johan Back. Invariant based programming revisited. Technical Report 661, TUCS - Turku Centre for Computer Science, Turku, Finland, 2005.
- [7] Ralph-Johan Back. Invariant based programming. In S. Donatelli and P. S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006*, pages 1–18, June 2006.
- [8] Ralph-Johan Back. Invariant based programming: basic approach and teaching experiences. *Form. Asp. Comput.*, 21(3):227–244, 2009.
- [9] Ralph-Johan Back, Johannes Eriksson, and Linda Mannila. Teaching the construction of correct programs using invariant based programming. In *Proc. of the 3rd South-East European Workshop on Formal Methods*, Thessaloniki, Greece, 2007.
- [10] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Verifying invariant based programs in the socos environment. In *BCS-FACS Workshop on Teaching Formal Methods: Practice and Learning Experience*, London, UK, December 2006.
- [11] Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, (15):95–114, 2001.
- [12] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research Methods in Education*. Routledge: New York, 6th edition, 2007.

- [13] Richard Denman, David A. Naumann, Walter Potter, and Gary Richter. Derivation of programs for freshmen. In *Proc. of the 25th SIGCSE symposium*, pages 116–120, New York, NY, USA, 1994. ACM Press.
- [14] Peter J. Denning. A debate on teaching computing science. *Commun. ACM*, 32(12):1397–1414, 1989.
- [15] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [16] Edsger W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, Dec 1989.
- [17] David Evans and Michael Peck. Inculcating invariants in introductory courses. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 673–678, New York, NY, USA, 2006. ACM Press.
- [18] Robert Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
- [19] David Ginat. Loop invariants and mathematical games. *SIGCSE Bulletin*, 27(1):263–267, 1995.
- [20] David Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [22] Joint Task Force on Computing Curricula. Computing Curricula 2001: Computer Science, December 2001.
- [23] Andrew McGettrick, Roger Boyle, Roland Ibbett, John Loyd, Gillian Lovegrove, and Keith Mander. Grand challenges in computing education. Technical report, BCS - The British Computer Society, 2004.
- [24] Kirby McMaster, Nicole Anderson, and Brian Rague. Discrete math with programming: better together. In *Proc. of the 38th SIGCSE symposium*, pages 100–104. ACM Press, 2007.
- [25] Peter Naur. Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4):310–316, July 1966.
- [26] Joy N. Reed and Jane E. Sinclair. Motivating study of formal methods in the classroom. In *TFM 2004*, pages 32–46. Springer-Verlag Berlin Heidelberg, 2004.
- [27] J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.

- [28] Rita C. Richey and James D. Klein. Developmental research methods: Creating knowledge from instructional design and development practice. *Journal of Computing in Higher Education*, 16(2):23–38, 2005.
- [29] Geoffrey G Roy. Designing and explaining programs with a literate pseudocode. *J. Educ. Resour. Comput.*, 6(1):1, 2006.
- [30] Abhik Roychoudhury. Introducing model checking to undergraduates. In *Formal Methods Education Workshop*, pages 9–15, 2006.
- [31] Wing C. Tam. Teaching loop invariants to beginners by examples. In *Proc. of the 23rd SIGCSE symposium*, pages 92–96, New York, NY, USA, 1992. ACM Press.
- [32] Jan van den Akker. *Design Approaches and Tools in Education and Training*, chapter Chapter I: Principles and Methods of Development Research. Kluwer Academic Publishers, 1999.
- [33] M. H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, SE-5(2):148–159, 1979.

Turku Centre for Computer Science

TUCS Dissertations

90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Sääntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Methods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation

TURKU
CENTRE *for*
COMPUTER
SCIENCE

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Information Technologies



Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2364-8

ISSN 1239-1883

Linda Mannila (Grandell)

Teaching Mathematics and Programming – New Approaches with Empirical Evaluation