Basic Proof Skills of Computer Science Students

Pieter H. Hartel[1], Bert van Es[1], Dick Tromp[2]

[1] Faculty of Mathematics and Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, {pieter,vanes}@fwi.uva.nl
[2] Formerly at SCO-KIOO (Foundation Centre for Education Research at the University of Amsterdam)

Abstract. Computer science students need mathematical proof skills. At our University, these skills are being taught as part of various mathematics and computer science courses. To test the skills of our students, we have asked them to work out a number of exercises. We found that our students are not as well trained in basic proof skills as we would have hoped. The main reason is that proof skills are not emphasized enough. Our findings are the result of a small experiment using a longitudinal measurement of skills. This method gives better insight in the skills of students than more traditional exam-based testing methods. Longitudinal measurement does not allow the students to specifically prepare themselves for particular questions. The measurements thus relate to skills that are retained for a longer period of time.
In our Department, fierce debates have been held in the past discussing such issues as "what proof skills do our students have?". An important aspect of our work is that it tries to find evidence to help answer to such questions. Research such as ours is rare in the field of teaching mathematics and computer science.

1 Introduction

Computer scientists must be able to study the foundations of their discipline. Discrete mathematics and mathematical logic are foundations of the core of Computer Science (CS) [14]. A computer scientist must therefore be skilled in the use of basic tools and techniques from discrete mathematics and logic. Continuous mathematics and other branches of mathematics are used more in applied CS subjects such as robotics and scientific computing. Mathematical tools and techniques from such areas are also important but perhaps not quite as fundamental. We will therefore concentrate on discrete mathematics and logic.

The development of skills in mathematical manipulation should also be an integral part of the programming activity. Gries, on the occasion of receiving the annual SIGCSE award for outstanding contributions to CS education, made it clear that such skills are essential to be able to manipulate large complex structures [3]. Parnas, another well known computer scientist subscribes to this view: "CS programs must return to a classical engineering approach that emphasizes fundamentals" [10].

It is thus rather unfortunate that mathematics is mostly taught to CS students as a separate activity. When the necessary skills are being taught, they are not always perceived as related to programming. The powerful interpretation of 'proofs as programs' is then not seen as natural. Many algorithms can be viewed as a slightly different rendering of a proof; see Asch [15] for a number of examples.

For CS, mathematics is best taught as part of an integrated curriculum in which the relationships between programming and mathematics are exploited [20]. Declarative programming styles (functional and logic programming) facilitate this integration. Teaching mathematics and declarative programming in an integrated fashion has received attention in the literature [7, 19, 6], and many CS departments are considering such issues [4].

In the CS department at the Universiteit van Amsterdam (UvA) we have also been discussing these issues. Our present curriculum begins with a first programming course in Pascal using methodical problem solving methods [13] and separate courses in logic and discrete mathematics. Functional programming is offered as the option 'functional languages and architectures' during the 3rd/4th year. As of the academic year 1995/1996, the first programming language will be a functional language. The relationship between the logic course, the discrete mathematics course and the functional programming course will be strengthened: all three subjects will be taught in parallel, thus providing the opportunity for integrated teaching.

As a preparation for the transition, we were interested in the present level of proof skills that our students have, and also in the question: Does the study of functional programming have an effect on the acquisition of such skills? Here we look at specific but essential aspects of discrete mathematics as a first step towards exploring the full set of mathematical skills of our CS students.

The aspects that we consider are basic equational reasoning and induction. This represents a choice out of a vast range of mathematical tools and techniques. Equational reasoning is the basis of all program transformation methods and as such an essential tool for the computer scientist. Induction is the only tool that supports the manipulation of potentially infinite structures, such as often encountered in CS. Induction and equational reasoning are therefore important.

It is unusual to find experimental data relating the skills of CS students to what their teachers would have expected, other than straightforward examination results. In a previous study we looked at all issues of the ACM SIGCSE bulletin over the past seven years. This study revealed that precious little hard evidence can be found for statements about CS education [5]. A notable exception to this rule is formed by the CS department of the Technical University at Twente. A recent Ph.D. thesis contains a comprehensive investigation into the effects that various decisions about the CS curriculum have on student performance [17]. Our experiment is of course small compared to the Twente experiment. It is interesting to note that both experiments relate to functional programming.

In the next section we discuss the context of the exploration into the mathematical skills of our students. Section 3 describes the experiment that has been carried out to measure the skills. The exercises and the evaluation criteria used to mark the exercises are described in Sect. 4. Section 5 presents the experimental results. The final section gives the conclusions.

2 Curriculum

The CS curriculum at the UvA nominally takes 4 years. A year has three terms of 14 weeks each. During a term 2–4 different subjects are taught. The amount of effort required to study a subject is rated using a point system. Each point corresponds to one week (that is 38 hours) of effort. The total number of points available during one term is thus 14. Subjects may be rated at 3, 4 or 7 points, depending on the amount of effort required by a hypothetical 'average' student. The point rating of each subject should cover all activities related to mastering that particular subject, including lectures, laboratories, homework and the preparation of the test.

2.1 Tests

Students normally take tests in a subject at the end of the term. There is no obligation to take a test directly after the course; tests are scheduled regularly. Students are also allowed to take a test a number of times until the test has been passed.

During the first 7 terms, the UvA CS curriculum has a common programme. Then, a choice must be made out of three specialisations, each of which lasts for another 5 terms. The specialisations on offer are theoretical CS (emphasis on mathematical logic, complexity theory, data base theory and natural languages), programming methodology (emphasis on algebraic formal methods) and technical CS (emphasis on robotics, image processing and parallel computing). The last 6 months of the specialisation are devoted to the final year project. There is a large degree of freedom in selecting topics for the specialisation. Students are encouraged to take courses from the different specialisations on offer. They are also allowed to choose some subjects from other disciplines.

The organisation of the curriculum offers the student considerable freedom in planning the programme of study. Firstly, a student chooses a number of subjects and an order in which they are studied. The dependencies specified by the prerequisites of each course constrain the freedom somewhat. Secondly, once a course has been taken, the student may delay taking the test until sufficient knowledge, experience and confidence has been acquired.

The system makes it difficult both for the student and for the staff to have accurate information about the progress that is being made. The data that are presently available record the number of points earned and the selection of tests passed. In theory, this should be a good predictor for the progress made. In practice, this is not always the case, as the points are awarded on the basis of how much work an 'average' student is supposed to spend studying a subject.

Table 1. Subjects taught during the first 6 terms of the CS curriculum. The numbers represent the point rating of each subject. One point corresponds to one week of full time study.

term I		term II		term III	
Introduction CS	3			Relational data bases	4
Logic	4	Discrete Mathematics	4	Graph Theory	3
		Continuous Mathematics	3		
Programming	7	Data Structures	7	Computer organisation	7
term IV		term V		term VI	
				CS ethics; presentation skills	7
Automata &	7	Linear Algebra	4	Calculus	4
complexity theory		Probability & statistics	3	Algebraic Structures	3
Operating systems	7	Programming Environments	7		

Many students spend more time, especially on the more theoretically-oriented subjects.

As a consequence, few students manage to complete their studies within 4 years even though a delay has severe financial implications. There are also other factors that cause delays, such as the need for many students to supplement their income through part-time employment.

The main problem with the present system of student performance assessment is that it does not give clear early warnings. A student who feels ill at ease with a particular subject will perhaps postpone a thorough study of the subject and also delay taking the test. For the staff this is not easy to detect, as one would have to monitor what students are *not* doing. For the student, the implications of postponing study of a particular subject may not be obvious: it is often the case that an insufficient background makes the study of new subjects more difficult, even though the newly-experienced difficulties are not directly traceable to the earlier, explicit or implicit choice to delay the previous subject.

A system of progress tests, such as operated by the University of Limburg Medical school in Maastricht [12] might be considered as an alternative to the present assessment system or as a means to supplement the present system. The progress test as used in Maastricht uses a set of questions that is fixed for the entire programme of study. The set is very large so that students cannot prepare specifically for the tests.

2.2 Proof Skills

The present paper reports on a small experiment with a test designed to identify progress in the acquisition of skills of the CS students. The small scale of the experiment required us to concentrate on a few aspects of the curriculum that we find essential. These are the skill in giving a simple proof by equational reasoning, the skill in constructing a proof by induction and the skill in creating an inductive definition.

Table 1 gives an overview of the subjects taught during the first two years together with their point rating. The contents of most courses will probably be obvious; the programming environment course includes as one of its components the subject of compiler construction.

The basic proof skills required are explicitly taught during the first term and reinforced during later terms. Explicit teaching of a concept means that the concept is taught for its own sake. Most teaching activities involve several concepts of which one is taught explicitly. The other concepts involved will then be taught more or less implicitly.

The following subjects contribute explicitly to a mastery of basic proof skills:

Logic – term I As part of the Introductory course on Logic equational reasoning and the principle of induction are taught. These techniques are applied to inductive proofs and definitions. On page 15 of the text book [16], the concept of an inductive definition is first explained. This is followed by several examples (the first example is actually the same as Exercise 3 of Round 1 of our experiments). The students practice giving inductive definitions during the tutorials. The test includes an exercise that requires an inductive definition.

Proofs by induction are the main subject of chapter 5 (page 69–86) of the text [16]. Here properties of formulae are proved by structural induction over inductively defined formulae. The tutorials include several exercises. However, the test is designed such that students may pass if they decide to skip the inductive proof. Students often do not appreciate the power of the principle of induction and prefer simple arguments or even a ‘proof by example’.

Discrete Mathematics – term II Both set theory and induction are taught as part of the discrete mathematics course. Many proofs are shown as examples. Students are advised to try some of the proofs at home but do not always follow the advice. The test includes an exercise that requires an inductive proof. The test results for this exercise show that there are two main categories of students: those that do well and those that fail; there are not many intermediate scores. In the discrete mathematics course, inductive proofs are used mostly to prove equalities. Students find it harder to prove an inequality instead. This indicates that at this stage of the curriculum, the full generality of the principle of induction is not appreciated by all students.

Graph Theory – term III The graph theory course makes use of the principle of induction, both for the purpose of giving inductive definitions and for proving properties of graphs and graph algorithms. The course is aimed specifically at CS students and therefore emphasizes algorithmic aspects more than proofs. The exam does not require the construction of inductive proofs.

Automata and Complexity theory – term IV Set theory is used in the course on automata and complexity theory. Inductive proofs are used here often. The students are given exercises that require inductive proofs; the exercises are discussed during tutorials. The test also includes an exercise that requires an inductive proof.

Algebraic structures – term V Set theory is heavily used in the course on algebraic structures, inductive proofs are rare in this course.

Towards the end of the first year, we would expect students to have acquired a reasonable level of proof skills. However, it is possible that students progress to this point without actually understanding the principle of induction. During the second year, the students should improve their basic proof skills to the point where one would expect all students to be able to carry out at least a simple inductive proof.

Inductive definitions do not play quite such an important role in the courses described, so one might expect the students to have difficulties creating an inductive definition.

3 Experiment

The aims of the experiment were, firstly, to investigate to what extent UvA CS students are able to construct simple proofs and, secondly, to investigate at what stage of the curriculum basic proof skills were mastered. More specifically, the second aim has been to measure progress from one year to another, and to measure progress within each year.

The experiment consisted of two rounds. During both rounds we asked the participants to work out a set of three exercises. The two sets were different, but comparable: they were designed to be of the same level of difficulty.

For the first-year students the exercises might have been hard; for the final year students they might have been easy; but all participants have been taught how to work out such exercises. Any difference in skills should thus be attributed to the experience that participants may have gained during their programme of study.

The first round was held at the beginning of term III of the academic year 1994/1995. The second round was held two months later, at the end of that same term. The participants were asked to supply their registration numbers, so that the progress made by participants who would cooperate on both occasions could be recorded accurately.

To keep the conditions during which the experiments were carried out the same, the exercises were completed during regular lecture or laboratory hours. On both rounds we allowed 20 minutes in total to allow for a reasonable amount of time to work out the exercises and at the same time to disrupt regular teaching as little as possible. The participants were not prepared in any way for the first round. After that they knew that the second round was coming, but not when it would come. The results or model answers were not communicated to the participants.

During the first round we asked 77 students to participate, of which only one refused. Five students refused to cooperate during the second round when we asked 60 students to take part. The participants were told in advance that the results would only be used for research purposes. They would thus not be disadvantaged by participating. As a small reward, six (CS) books were distributed amongst the participants.

The cohorts 1990 ... 1994 of UvA CS students all contain roughly the same number of students. For our two samples to be representative for the total population we would thus expect the numbers of students from these cohorts in the samples to be similar. This was found to be the case with one exception: the 1992 cohort on Round 1 contains twice as many students as expected. This gives a slight bias towards third year students on Round 1.

We have no indication that the skills of the students who did participate are not representative for the skills of the population as a whole. Both very able students and less able students sometimes do not go to class. The experiments were carried out towards the end of the year, when the student population that goes to class is relatively stable. Most students who drop out do so towards the beginning of the year. There is thus no indication that we may have worked with particularly skilled or particularly unskilled students. We could have investigated this matter further by using the exam-based results of our students. We decided against this to guarantee students that participating in the experiment would not be connected with their exam-based results.

Table 2. A fragment of the algebra of sets: A, B en C are subsets of a universal set U, the complement of a set A is written as A^c.

1a.	$A \cup B = B \cup A$	Commutativity
b.	$A \cap B = B \cap A$	
2a.	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	Distributivity
b.	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	
3a.	$A \cup \emptyset = A$	Identity
b.	$A \cup U = U$	
c.	$A \cap \emptyset = \emptyset$	
d.	$A \cap U = A$	
4a.	$A \cup A^c = U$	Complement
b.	$A \cap A^c = \emptyset$	
c.	$(A^c)^c = A$	
5a.	$A \cap B \subseteq A$	Inclusion
b.	$A \subseteq A \cup B$	

4 Exercises

In both rounds, the first exercise tested equational reasoning, the second tested the skill in constructing an inductive proof and the third exercise required the construction of an inductive definition.

We have tried to make all execises of the same level of difficulty. During the design of the exercises we consulted with a number of colleagues to make sure that the exercises would represent a good test of basic proof skills.

4.1 Exercise 1: Equational Reasoning

Equational reasoning is the basis of many proof systems. Set theory is an important part of discrete mathematics.

The first round required the participants to prove the inequality below, while using the axioms of Table 2:

$$(A \cup B) \cap A^c \subseteq B$$

The second round required the participants to prove the inequality:

$$B \subseteq (A^c \cap B) \cup A$$

These exercises are of the same level of difficulty. We give the model answer to the first:

$$\begin{array}{lllr}
 & (A \cup B) \cap A^c & = A^c \cap (A \cup B) & \{1b\} \\
 & & = (A^c \cap A) \cup (A^c \cap B) & \{2b\} \\
 & & = (A \cap A^c) \cup (A^c \cap B) & \{1b\} \\
 & & = \emptyset \cup (A^c \cap B) & \{4b\} \\
 & & = (A^c \cap B) \cup \emptyset & \{1a\} \\
 & & = (A^c \cap B) & \{3a\} \\
 & & = (B \cap A^c) & \{1b\} \\
\text{and:} & (B \cap A^c) & \subseteq B & \{5a\} \\
\text{therefore:} & (A \cup B) \cap A^c & \subseteq B &
\end{array}$$

The model answer for Exercise 1 of the second round follows the same lines, taking basically the same steps in a slightly different order.

To construct the above proof, the student needs to be able to instantiate an axiom and to use the transitivity of equality. This tests only the very essentials of equational reasoning. The choice of the particular proof steps must be driven by intuition. All students should be thoroughly familiar with the axioms of set theory that we are using here. All should therefore have sufficient intuition to choose the appropriate proof steps. The danger of choosing a familiar domain to test basic equational reasoning skills is that some of the axioms may be viewed as trivial. The commutativity axioms are obvious examples. We decided to choose familiar axioms, as using unfamiliar ones would have disabled the intuition of the student. This would have made the exercise too difficult.

The individual steps in the proof above have been labelled with the number of the axiom used. This makes the proof easier to read. Annotating proof steps should therefore be considered good practice. This view is not universally held. Many mathematics texts will give long sequences of proof steps without annotations. Examples of computing books that do annotate proof steps and derivation steps are the books by Morgan [9] and Bird and Wadler [1].

All exercises were marked on a scale from 0=poor to 10=excellent. A pass mark is at least 5.5. A general criterion and some specific criteria were used to calculate the marks. The general criterion looks at whether the question has been answered at all and, if so, whether the answer is complete.

For Exercise 1, the full set of criteria and the percentages of the mark awarded are:

connectedness: 20% Have the individual proof steps been connected properly? Some participants write a number of formulae without a hint of how they are connected, so that there is no apparent logic in the reasoning.

explicitness: 40% Have all steps been made explicit? Many participants forget to note the use of the commutativity axiom. Most other axioms were used explicitly but not by every participant. Each of the seven steps above contributes 1/8 of 40%.

annotations: 20% Has each step been labelled with the name or the number of the axiom applied? Many participants leave it to the reader to guess which axioms have been applied. Each of the seven steps above contributes 1/8 of 20%.

general: 20% The general criterion for Exercise 1 accounts for 20% of the mark.

Some criteria are awarded 20% of the full mark. This indicates that an otherwise perfect answer that completely fails on just one such criterion would still result in a good mark. Completely failing on a criterion that is awarded with 40% yields a mark that is just sufficient. Explicitness falls into the 40% category as this criterion essentially captures whether students are able to instantiate the axioms properly.

4.2 Exercise 2 : Inductive Proof

The properties of some formal systems can be proved by simple induction over the natural numbers. Properties of many more formal systems can be proved by structural induction. To test the skill in proving a property by induction we have chosen to work in the domain that is most familiar to the students; the natural numbers. Other domains such as formal languages would have been unsuitable for the first-year students.

The first round required the participants to prove by induction that for all positive natural numbers n the following equality holds:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

The second round required the participants to prove that for all positive natural numbers n the following property is true:

$$n^3 - 4n + 6 \quad \text{is divisible by} \quad 3$$

Both proofs involve elementary algebra using cubic polynomials.

The hypothesis $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$ is used to prove the first equation by induction over n:

$$\begin{array}{lll}
\text{Case 1:} & \sum_{k=1}^{1} k^2 & = 1 \\
 & & = \frac{1(1+1)(2\times 1+1)}{6} \\
\\
\text{Case (n+1):} & \sum_{k=1}^{n+1} k^2 & = \sum_{k=1}^{n} k^2 + (n+1)^2 \\
 & & = \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \qquad \text{(hypothesis)} \\
 & & = \frac{n(n+1)(2n+1)+6(n+1)^2}{6} \\
 & & = \frac{(n+1)(n(2n+1)+6(n+1))}{6} \\
 & & = \frac{(n+1)(2n^2+n+6n+6)}{6} \\
 & & = \frac{(n+1)(n+2)(2n+3)}{6} \\
 & & = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}
\end{array}$$

The proof of Exercise 2 from the second round follows the same lines.

The evaluation criteria for Exercise 2 are:

connectedness: 20% Have the individual proof steps been connected properly? (See under Exercise 1).

base case: 10% Is the base case present and has it been worked out properly? Some participants prove the base case for $n = 0$ instead of $n = 1$. The exercise explicitly states that the proof should apply to the *positive* natural numbers.

inductive case: 30% Is the inductive case properly worked out? Some participants start to work with $\sum_{k=1}^{n}(k+1)^2$.

annotations: 10% Has the use of the induction hypothesis been annotated? A useful sanity check when giving an inductive proof is to verify that the induction hypothesis has been used. Some participants leave it up to the reader to find out when the hypothesis has been used.

algebra: 10% Is the elementary high school algebra a problem? Many mistakes were made with the elementary algebra.

general: 20% The general criterion for Exercise 2 accounts for 20% of the mark.

Some criteria are awarded 10% of the mark. These represent relatively minor issues or elements of the proof that require little work. The inductive case represents a relatively large amount of work, which justifies its relatively large contribution to the mark.

4.3 Exercise 3 : Inductive Definition

Compositionality is the key to reasoning about complex structures in terms of their simpler components. An inductive definition can be given for a vast number of complex structures. The skill in producing such an inductive definition was the target of our third and last exercise.

Constructing inductive definitions is taught explicitly as part of the course on logic during the first term.

The first round required the participants to give an inductive definition of the formulae of propositional logic using the connectives $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ and $\neg$ and using as basis elements the variables p, q, r.

The model answer is:

Base case The variables p, q en r are formulae.

Inductive case Let P and Q be formulae, then $(P \vee Q)$, $(P \wedge Q)$, $(P \rightarrow Q)$, $(P \leftrightarrow Q)$ and $\neg P$ are formulae.

Closure No other terms than the ones mentioned under Base and Inductive case above are formulae.

The second round required the participants to give an inductive definition of the formulae of arithmetic, using the connectives $+$, $\times$, $>$, $=$, $-$ and the numbers 0, 1, 2 ... as basis elements.

Strictly speaking, this exercise does not test a proof skill, but rather a 'definition skill'. A more interesting test would have been to construct an inductive definition and its induction principle. Then, some property of the inductively defined structure could have been proved. Unfortunately, such an exercise would have been too time-consuming for the present constrained experiment.

The evaluation criteria for Exercise 3 are:

Base case: 20% Is the base case properly identified? Many participants forget to state which formulae are the basic elements.

Inductive case: 60% Has the inductive case been formulated? Participants either describe the inductive case properly or reproduce something completely different, such as the laws of boolean algebra (Round 1) or the Peano axioms (Round 2).

Closure: 20% Have other terms been explicitly excluded? Participants often forget to make it explicit that only the smallest class of formula is relevant. (The 'no junk' rule).

For this exercise the general criterion is subsumed by the inductive case. The inductive case has a weight of 60% as without it, the answer would be insufficient. A correct base case gives 20% of the mark, such that the ratio base case : inductive case = 1 : 3. This is the same ratio as for Exercise 2.

Table 3. Various sub-groups of the participants of Rounds 1 and 2. Here n= the number of participants; $\bar{x}$ = average mark (on a scale from 0=poor – 10=excellent; a pass mark is at least 5.5), $m.o.e.$= margin of error, s= standard deviation.

Sub-groups	round 1				round 2			
	n	$\bar{x}$	s	$m.o.e.$	n	$\bar{x}$	s	$m.o.e.$
difference	34	-0.6	1.5	0.5	34	+0.6	1.5	0.5
total	76	6.0	2.2	0.5	55	6.4	2.7	0.7

(a) Exercise 1 results of all participants

Sub-groups	round 1				round 2			
	n	$\bar{x}$	s	$m.o.e.$	n	$\bar{x}$	s	$m.o.e.$
difference	34	-0.4	3.9	1.3	34	+0.4	3.9	1.3
total	76	6.1	3.3	0.8	55	6.8	3.5	0.9

(b) Exercise 2 results of all participants

Sub-groups	round 1				round 2			
	n	$\bar{x}$	s	$m.o.e.$	n	$\bar{x}$	s	$m.o.e.$
difference	34	-1.4	3.4	1.1	34	+1.4	3.4	1.1
total	76	2.1	3.7	0.8	55	3.0	3.9	1.0

(c) Exercise 3 results of all participants

Sub-groups	round 1				round 2			
	n	$\bar{x}$	s	$m.o.e.$	n	$\bar{x}$	s	$m.o.e.$
1–7 common	21	4.1	2.0	0.8	15	4.9	2.5	1.2
8–14 common	22	5.0	1.4	0.6	14	4.7	1.8	0.9
common+theory	16	5.3	1.1	0.5	12	6.4	1.8	1.0
common+programming	20	5.6	1.8	0.8	17	6.8	1.8	0.9
common+technical	36	5.2	1.8	0.6	33	6.4	1.9	0.7
once	42	4.5	2.0	0.6	21	4.8	2.4	1.0
twice	34	5.0	1.9	0.6	34	5.8	2.2	0.7
twice (functional)	11	6.3	1.6	0.9	11	7.2	1.9	1.1
VU	14	5.4	1.1	0.6	7	6.3	2.0	1.5
UvA	62	4.6	2.1	0.5	48	5.3	2.4	0.7
total	76	4.7	2.0	0.4	55	5.4	2.3	0.6

(d) Overall results of all participants based on the average of Exercises 1, 2 and 3

5 Results

The total number of registered UvA CS students during the academic year 1994/1995 is 163. A number of these students do not go to class, in particular when they are working on their final year project. We could thus not reasonably expect the entire student population to participate. During the first round, 76 students took part and during the second round there were 55 participants. 34 participants cooperated in both rounds. We specifically handed out the exercises during laboratories and lectures scheduled for UvA CS students, but a fraction of the participants were not UvA CS students. On the first round 51 (of 76) were UvA CS students and on the second round 35 (of 55) were UvA CS students.

5.1 Student Groups

Table 3 presents various breakdowns of the group. Table 3-a, 3-b and 3-c apply to Exercises 1, 2 and 3 respectively. Table 3-d applies to the total test, based on the average of the three exercises. The exercises have equal weight.

The rows within the table correspond to certain sub-groups of the group of participants. Each row gives results obtained during the first and the second round of the experiment. The results are the number of participants (n), their average mark ($\bar{x}$), the margin of error of the average ($m.o.e.$) and the standard deviation of the average (s). (The margin of error is defined as $m.o.e. = 1.96s/n$). The number n varies from row to row because not all participants are part of each sub-group.

The sub-groups in Table 3 have been chosen partly such that progress in the acquisition of basic proof skills of the group as a whole is visible; partly to investigate whether students interested in one specialisation/subject have better proof skills than others.

The sub-group *1–7 common* has studied between 1 and 7 subjects from the list given by Table 1. These students may or may not have taken the tests for these first subjects. This sub-group has studied at most about one third of the common programme. The sub-group *8–14 common* has studied between 8 and 14 subjects (two thirds of the common programme). The sub-group *common+theory* has studied all or most of the common programme and at least one theoretical CS subject, which indicates that they may perhaps be more interested in theoretical issues than other students. Similarly, the sub-group *common+programming* has studied at least one programming methodology subject and the sub-group *common+technical* has studied at least one technical CS subject. Many students will study subjects from the different specialisations. There is thus some overlap between these five sub-groups.

The sub-group *once* has participated either in Round 1 or in Round 2 but not in both. The sub-group *twice* has participated in both Rounds 1 and 2. There is therefore no overlap between the sub-groups *once* and *twice.* The sub-group *difference* is the same as the sub-group *twice* except that for each student in the sub-group the difference between the marks awarded in Rounds 1 and 2 is calculated. The statistics given apply to these differences. A positive average in the column for Round 1 indicates that the marks in Round 1 were higher, a negative average in the same column indicates that higher marks were obtained in Round 2. The sub-group *twice (functional)* applies to students who participated on both rounds and who took the third year optional course functional languages and architectures.

An agreement between the UvA and the nearby Vrije Universiteit (VU) of Amsterdam makes it easy for CS students to take part in the courses of the neighbouring university. The sub-groups *UvA* and *VU* represent participants from the two universities in Amsterdam. These sub-groups do not overlap.

5.2 Student Performance

Table 3 shows that the average marks for Exercises 1 and 2 are sufficient. The marks for Round 1 are 6.0 and 6.1 for Exercises 1 and 2 respectively; for Round 2 they are 6.4 and 6.8. The marks cannot be qualified as good, which would require at least an 8. The participants are thus able to perform basic equational reasoning and to give a simple inductive proof, but one would suspect that more demanding proofs would give problems to many students.

The average marks for Exercise 3 (Table 3-c) are insufficient (2.1 for Round 1 and 3.0 for Round 2). About 10% of the participants noted explicitly on their forms that they had no idea what an 'inductive definition' might be. About 25% of the participants indicated that they needed more time for the three exercises. Exercise 3 was the last exercise, which makes it likely that time pressure has had a negative influence on the result for Exercise 3.

The low average marks for Exercise 3 apply to all participants. We had hoped that UvA CS students would do better than they did but this was not the case. Their averages are about the same as those found for the whole group of participants. This applies to all exercises, not just to Exercise 3. Clearly the concept of a inductive definition is being taught but not used often enough for all students to make it operational.

The rows marked *difference* for Exercises 1, 2 and 3 show an improvement in performance from Round 1 to Round 2 of +0.6, +0.4 and +1.4 respectively. The margins of error for Exercises 1 and 3 are just small enough (0.5 and 1.1 respectively) to suggest that the improvements may be significant. The margin of error is meaningful only if the data follow a normal distribution. This is not always the case here. We have applied Wilcoxon's non-parametric signed rank-test for paired data [8] to investigate this matter more closely. This test confirms that the participants have learned something that has helped them to improve their performance on Exercise 1. From the same test we could not conclude that something has been learned to improve the performance on the other two exercises.

We have also investigated what caused this improvement: Is it the learning effect that taking the tests on Round 1 has had on the results for Round 2? Or is it the result of the education during the two months that separate the two rounds? A Wilcoxon two sample test shows that the effect of participating on Round 1 on the performance during Round 2 is not significant. We have not handed out the model answers or discussed the results, so the learning effect of taking part in one of the rounds is minimal.

The variation in the performance of students at different stages of their programme of study is presented in Table 3-d. The rising averages from sub-group *1–7 common*, via *8–14 common* to either of the three sub-groups *common+*... suggest that students acquire proof skills as they progress towards the end of their programme of study. This is what one would expect. The averages for the sub-group *common+theory*, *common+programming* and *common+technical* are close, with largely overlapping margins of error. Again the distributions are not always normal, so reverting to a Wilcoxon-test we found that on Round 1 the improvement from *1–7 common* to *8–14 common* 1 is significant and that on Round 2 the improvement from *8–14 common* to any of *common+*... is significant.

The VU students seem to perform better than the UvA students: on round one the VU average mark was 5.4, the UvA mark 4.6; on round two the average marks were 6.3 and 5.4 respectively. These marks apply to relatively few VU students so it is not sensible to investigate this further on the basis of the available data. A firm conclusion can thus not be drawn. It would be reasonable to expect VU students to do better than UvA students: students who take courses at the neighbouring university might be more strongly motivated, have more initiative and are generally more resourceful than other students. The initiative is needed to overcome the problem that one university uses a semester system, whereas the other uses a trimester system. The time-tables for students who visit the neighbouring university are therefore complex.

The option functional languages and architectures is being taught during the third term, as a 7 point subject. The course consists of two parts. The first part follows Bird and Wadler [1] closely, with a strong emphasis on program transformation and on proving properties of functions using induction. The second part of the course uses Peyton Jones [11] to teach the principles of implementing functional languages. The emphasis is on the lambda calculus and various abstract machine models. Here equational reasoning and inductive proofs are also used, but to a lesser extent than in part 1. The functional languages and architectures course is accompanied by a laboratory where the students build a combinator parser [18] and a type checker [2, Ch. 7] for a simple functional language. See table 5 for a summary of the details of the course.

We found no indication that the study of functional programming as a separate subject has an effect on the acquisition of basic proof skills. Of the participants who took the option functional languages and architectures, a small group of 11 students participated on both rounds. Their results are shown in the row marked *twice (functional)* in Table 3-d. The improvement of their basic proof skills from Round 1 to Round 2 was found to be +0.9, which is basically the same as that found for the entire group of 26 UvA CS students, who participated in both rounds. The average mark of the participants who took functional languages and architectures is rather higher than the average mark of all other participant groups. This is the case both on Round 1 (average mark 6.3) and on Round 2 (average mark 7.2). We think that this indicates only that students who choose functional languages and architectures as an option are attracted to the more fundamental approach to programming.

Table 4. Opinions of the participants of Rounds 1 and 2. n= the number of participants; $\bar{x}$ = average mark (on a scale from 0=poor – 10=excellent; a pass mark is at least 5.5), $m.o.e.$= margin of error, s= standard deviation.

	round 1				round 2			
	n	$\bar{x}$	s	$m.o.e.$	n	$\bar{x}$	s	$m.o.e.$
I found this easy	29	5.9	1.6	0.6	13	7.0	1.7	0.9
I enjoyed doing this	20	6.2	1.6	0.7	11	6.9	2.2	1.3
I practised proofs too long ago	26	4.1	1.9	0.7	13	4.2	2.2	1.2
I needed more time	35	4.4	1.9	0.6	10	4.3	1.4	0.8
I could do this if I had a book	25	4.0	1.8	0.7	13	3.7	2.2	1.2

Table 5. Summary description of the course on functional languages and architectures.

Title of the course:	Functional languages and architectures						
Aims of the course:	1) To acquire functional programming skills. 2) To acquire basic insights in the theory of functional programming. 3) To gain a thorough understanding of how functional languages are implemented.						
Audience:	Mandatory for 3rd/4th year technical CS students, optional for other CS students.						
Prerequisites:	Thorough knowledge of imperative programming and computer architecture.						
Texts:	Textbooks [1, 11] and assignment booklet.						
Duration:	10 weeks teaching plus 4 weeks exam preparation						
		times per week		hours per session		total hours per week	
Time table:	lecture hours	2	×	1.5	=	3	
	laboratories	1	×	2	=	2	
	home work					15	
Assessment:	Two written examinations and six laboratory assignments.						

5.3 Student Opinions

The exercises were accompanied by a number of questionnaire items which enabled the participants to express their opinion on the exercises. There was a list of 21 options, with a blank space for the students to write their own. Table 4 shows which opinions were indicated as most appropriate, with the average mark, margin of error and standard deviation as calculated for the sub-group expressing that particular opinion.

Participants who had positive feelings ("I found this easy" and "I enjoyed doing this") did reasonably well. Participants who had negative feelings about their skills ("I practised proofs too long ago") did not so well, like the participants who felt that circumstantial effects were important ("I needed more time" and "I could do this if I had a book"). The option "I will never learn this" was not selected as appropriate by a single participant.

6 Conclusions

Our students are not as well trained in basic proof skills as we hoped. This is unfortunate, as the UvA CS curriculum does provide opportunity for training these skills: one of the first three subjects being taught in the curriculum is a course in mathematical logic. This should provide for a good start, but clearly not all students appreciate the importance of proofs. During the first two years of the curriculum proof techniques are not emphasized enough. Students may get the impression that proof skills are not as important as programming skills. After these two years, the students choose a specialisation which offers the possibility to choose either subjects that strongly emphasize proof techniques or subjects that do not. As a result, it is possible for CS students to graduate with limited proof skills.

From the experimental results we conclude that:

- A clear relation between the study of functional programming as a separate subject (i.e. not integrated with the mathematics teaching) and the acquisition of basic proof skills could not be found.
- There is some improvement in basic proof skills as the students progress from the first to the second and to later years.
- There is some improvement in basic proof skills as the students progress from the beginning to the end of the third term.
- The basic proof skills of the students should generally be improved.
- Students should not be permitted to skip exam questions that test proof skills. This gives them the impression that proof skills are unimportant.
- The training of basic proof skills should not be confined to a small set of mathematically-oriented subjects. Such concepts should appear throughout the curriculum.

We believe that the situation could be improved by teaching proof skills as an integral part of the programming subjects. At the UvA, the coming academic year will see a better integration of subjects during the first term. This represents a first and important step towards a more integrated approach. We are hopeful that this may soon be followed by more steps.

The methodology that we have used could be a first step towards a more comprehensive longitudinal measurement of skill. The results highlight certain problems that are not so evident from the results of the more traditional testing scheme.

7 Acknowledgements

We thank Natasha Alechina, Marcel Beemster, Johan van Benthem, Jan Bergstra, Mark van den Brand, Ben Bruidegom, Kees Doets, Edo Dooijes, Peter van Emde Boas, Theo Janssen, Paul Klint, Hugh McEvoy, Hans van der Meer, Jon Mountjoy, Cora Smit, Leen Torenvliet and Rob Veldman for their help with the experiments and for their comments on draft versions of the paper. The comments of the anonymous referees are gratefully acknowledged. The willingness of many of our students to participate was greatly appreciated.

References

1. R. S. Bird and P. L. Wadler. *Introduction to functional programming.* Prentice Hall, New York, 1988.
2. A. J. Field and P. G. Harrison. *Functional programming.* Addison Wesley, Reading, Massachusetts, 1988.
3. D. Gries. Improving the curriculum through the teaching of calculation and discrimination. *Education and computing*, 7(1,2), 1991.
4. R. Harrison. The use of functional programming languages in teaching computer science. *J. functional programming*, 3(1):67–75, Jan 1993.
5. P. H. Hartel and L. O. Hertzberger. Paradigms and laboratories in the core computer science curriculum: An overview. *ACM SIGCSE bulletin*, 27(4):13–20, Dec 1995.
6. J. L. Hein. A declarative laboratory approach for discrete structures, logic and computability. *ACM SIGCSE bulletin*, 25(3):19–24, Sep 1993.
7. P. B. Henderson and F. J. Romero. Teaching recursion as a problem-solving tool using standard ML. In R. A. Barrett and M. J. Mansfield, editors, *20th Computer science education*, pages 27–31, Louisville, Kentucky, Feb 1989. ACM SIGCSE bulletin, 21(1).
8. E. L. Lehman. *Nonparametrics: Statistical methods based on ranks.* Holden & Day, San Francisco, California, 1975.
9. C. Morgan. *Programming from specifications.* Prentice Hall, Hemel Hempstead, England, 1990.
10. D. L. Parnas. Education for computing professionals. *IEEE Computer*, 23(1):17–22, Jan 1990.
11. S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.
12. E. S. Tan. *A stochastic growth model for the longitudinal measurement of ability.* PhD thesis, Dept. of Maths. and Comp. Sys, Univ. of Amsterdam, Dec 1994.
13. Th. J. M. (Dick) Tromp. *The acquisition of expertise in computer programming.* PhD thesis, Dept. of Psychology, Univ. of Amsterdam, Sep 1989.

14. A. J. Turner. A summary of the ACM/IEEE-CS joint curriculum task force report: Computing curricula 1991. *CACM*, 34(6):69–84, Jun 1991.
15. A. G. van Asch. To prove, why and how? *Int. J. Mathematical education in science and technology*, 24(2):301–313, Mar 1993.
16. J. F. A. K. van Benthem, H. P. van Ditmarsch, J. Ketting, and W. P. M. Meyer-Viol. *Logica voor Informatici.* Addison-Wesley Nederland, Amsterdam, 1991.
17. K. van den Berg. *Software measurement and functional programming.* PhD thesis, Twente technical Univ., Jun 1995.
18. P. L. Wadler. How to replace failure by a list of successes, a method for exception handling, backtracking, and pattern matching in lazy functional languages. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 113–128, Nancy, France, Sep 1985. Springer-Verlag, Berlin.
19. R. L. Wainwright. Introducing functional programming in discrete mathematics. In M. J. Mansfield, C. M. White, and J. Hartman, editors, *23rd Computer science education*, pages 147–152, Kansas, Missouri, Mar 1992. ACM SIGCSE bulletin, 24(1).
20. U. Wolz and E. Conjura. Integrating mathematics and programming into a three tiered model for computer science education. In D. Joyce, editor, *25th Computer science education*, pages 223–227, Phoenix, Arizona, Mar 1994. ACM SIGCSE bulletin, 26(1).