HtDP and DMdA in the Battlefield

A Case Study in First-year Programming Instruction

Annette Bieniusa
Markus Degen
Phillip Heidegger
Peter Thiemann
Stefan Wehr
Albert-Ludwigs-Universität Freiburg
{bieniusa,degen,heidegger,thiemann,wehr}
@informatik.uni-freiburg.de

Martin Gasbichler *
Zühlke Engineering AG
martin.gasbichler@zuehlke.com

Michael Sperber
DeinProgramm
sperber@deinprogramm.de

Marcus Crestani
Herbert Klaeren
Eric Knauel
Eberhard-Karls-Universität Tübingen
{knauel,crestani,klaeren}
@informatik.uni-tuebingen.de

Abstract

Teaching the introductory course on programming is hard, even with well-proven didactic methods and material. This is a report on the first-year programming course taught at Tübingen and Freiburg universities. The course builds on the well-developed systematic approaches using functional programming, pioneered by the PLT group. In recent years, we have introduced novel approaches to the teaching process itself. In particular, assisted programming sessions gave the students a solid basis for developing their programming skills. In this paper we trace the development of our approach. Furthermore, we have collected information on how well our course had worked, and how the results together with our experience gained over years have lead to substantial, measurable improvements.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer and Information Science Education—Computer science education; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Experimentation, Languages

Keywords Introductory Programming Course, Assisted Programming, Scheme, TeachScheme!

1. Introduction

Each year thousands of students start studying computer science at universities. Most universities offer or even require an introductory course where students are taught basic programming skills. These courses (*intro courses* for short) usually teach programming in a particular programming language, the usage of programming environments and tools, proper documentation of code, testing and debugging, and so on. Their intention is to build up a proper general foundation in programming, as well as to prepare for subsequent courses.

However, teaching the intro course well is exceedingly difficult: Almost every teacher who has taught an intro course can recite a long litany of frustrating experiences. Even if students produce running programs, many of these programs lack simplicity, systematic construction, and correctness.

Functional programmers often think that these problems can easily be remedied by simply changing the programming language to a functional one. But this is not enough. The TeachScheme! project [12] has done pioneering work on developing a radically new and effective approach to the teaching of programming. They identify crucial items for developing systematic programming abilities across a wide range of student backgrounds. However, we have found that simply using the TeachScheme! methodology in a traditional (German) course still leaves room for improvement.

While a certain percentage of students always seems to do well, a large proportion just does not seem to grasp the essence of programming. Moreover, students often tend to dislike a course that emphasizes more systematic approaches to programming: Those with little or no experience in programming are easily discouraged by problems they encounter when doing their homework. These problems rank from using the programming language environment and simple syntax errors to actual problems of understanding the material. It is crucial to motivate these students with doable (read: easy) exercises in the first few weeks of the course. However, students who have previous "programming experience" often do not feel sufficiently challenged by the early homework exercises. Those students tend to solve them quickly and superficially by tinkering, missing the chance to acquire a systematic, clear and concise coding style. Between these two extremes there are many students who feel comfortable solving well-posed homework exercises. However, they are unable to solve problems that differ from the examples given in the lectures. They often run into trouble when trying to solve less structured problems.

Is there a way to teach students the required skills with neither boring nor discouraging them? Is it possible to teach them how to properly design programs and to maintain readable code? How can we enforce that they follow the design principles of clean coding?

We have adopted the tools of the trade developed by the TeachScheme! project: functional programming rather than object-oriented programming, Scheme as the programming language, and the DrScheme IDE for quick, test-driven development tailored towards students of the intro course. In addition, the most important new building block of our courses is *assisted programming*. In assisted programming sessions, we supervise our students closely on a weekly basis. Additional tutorial sessions serve to discuss selected topics in more depth.

Although many critics and sceptics did not believe we would succeed, our experience with this approach was stunningly good. Not only did the students perform well on exams, they were also able to transfer their knowledge to other programming languages

* Partially supported by Universität Tübingen.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FDPE'08, September 21, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-068-5/08/09. . . $5.00

(Java) and IDEs (Eclipse). Furthermore, their ability to think in a structured way helped them to perform well also in following courses.

In this paper we report on the development of the course, our experiences, and the results.

Overview Section 2 describes the common curricular circumstances of the intro course in Tübingen and Freiburg and specifies the material we cover in our intro courses. Section 3 gives a brief overview of specific differences between the Freiburg and Tübingen courses. Section 4 explains the development of the course concept in Tübingen. We then present in Section 5 an analysis of the results in the final exams and of a course evaluation conducted among students. Section 6 briefly describes follow-up courses and their interaction with the intro course. Section 7 concludes with a discussion of what we have achieved so far, and what we think can still be improved.

2. General Setting

The curricular environments in which the intro courses in Freiburg and Tübingen take place are quite similar, as are the goals of the intro courses described in this paper. This section describes the commonalities between the two courses in form and content and gives a short overview of rationale for the choice of material.

2.1 Goals

After finishing the course, students should know basic programming techniques as well as elementary data structures and algorithms. Furthermore, students should be aware of the fundamental concepts of programming languages, so that they can learn new languages on their own. Finally, the intro course should prepare students for a follow-up course using Java.

2.2 Course Outline

The contents of the course roughly follows the book "Die Macht der Abstraktion" [20] (in German, DMdA, engl.: *The Force of Abstraction*) which, in turn, is inspired by the book "How to Design Programs" [10] (HtDP). DMdA compresses the syllabus of HtDP by slightly increasing the "height" of the individual conceptual steps taken (read: it omits some of the fine-grained intermediate steps) and by omitting some of the advanced patterns for designing functions. Thus, DMdA is best suited (and laid out) as a supporting textbook for a course that reinforces the material by practical coursework. In addition, DMdA covers some mathematical background material that parallels the programming constructs just taught. This material is intended to instill in students the idea that programming is a mathematical activity and demonstrate some of its elegance.

2.2.1 Programming Methodology

The programming methodology taught by the course is that of systematic programming using data-driven and test-driven top-down design (following DMdA and HtDP). It requires students to follow a rigid process under strict supervision when doing their coursework. This process is enforced by assisted programming sessions as described in section 2.3.

Here is a very brief summary of this methodology: Initially, students define the data structures underlying their problem. They learn to build and document these structures from the standard types using primitive types, composition (product types), alternatives (variant types), and recursive types. Once the data structures are fixed, they define a function by first giving a one-line problem statement and then writing down a contract, which is initially just a type signature. The next step consists of writing down test cases for the function. Finally, the students implement the function. This activity is driven by the preceding analysis of the input and output types of the function. The DMdA/HtDP methodology provides a number of *design recipes* that make the construction of the function bodies to some extent a mechanical effort driven almost entirely by the types.

The point here is to constructively teach the students the design rules followed by successful and effective programmers. This is in sharp contrast to most traditional curriculi, which teach mostly by example, leading to "programming-by-tinkering". In our approach, they learn to systematically deconstruct a problem and then to construct the solution just as systematically. For details on this approach, consult HtDP or DMdA.

2.2.2 Programming Language and Environment

The course is based on a progression of *language levels* derived from the Scheme programming language [25] as provided by the DrScheme programming environment [7, 9, 13]. (The language levels were developed especially for DMdA. They are different from the HtDP language levels, but based on the same principles.) The progression of languages enables the teacher to adapt the expressiveness of the language to the current ability of the students, thus unleashing additional expressiveness as the students progress.

Using a Scheme dialect for teaching programming has several advantages.

- Programming instruction can make use of DrScheme, an integrated program development environment with documentation, stepper, debugger, support for testing, and some analysis. DrScheme familiarizes the students with the concepts of an IDE without overwhelming them with a plethora of features meant for professional programmers (as would be the case with Eclipse, for example, [2, 16, 24]).
- DrScheme is designed to support exactly the programming methodology explained in Sec. 2.2.1.
- DrScheme implements the language levels as well as the functionality to implement further levels of this kind.
- There is well-developed teaching material for the approach taken with several books being available. The students appreciate that DMdA is in German. A wide variety of books on advanced Scheme programming and on teaching programming concepts using the Scheme language [23, 8, 14, 27] show students that Scheme is not a dead end.
- The strong, dynamic typing regime of Scheme provides ultimate liberty in writing types and contracts: if needed, very complicated constrained types (as with Java generics or Haskell's type classes) or even dependent types can be written. In this aspect, Scheme is close to today's popular scripting languages, but without the drawback of weak typing and automatic type conversion that prevail in scripting.
- The dynamic typing regime also avoids frustrating struggles with type checkers and type inference algorithms (although DrScheme performs some static analysis and rejects some programs before they get to run).
- Scheme imposes no particular programming paradigm. We teach functional, imperative, object-oriented, and event-driven programming styles in the course.

2.2.3 Overview of Material Covered

Here is the progression of topics treated in the course.

- introduction emphasizing the need for contracts with an example suffering from underspecification
- functions consuming and producing values of primitive datatypes (numbers, strings, booleans)

- a formal execution model: the substitution model
- scope and lexical binding
- function definition by case distinction and branching
- record types with constructors, selectors, tests (the Scheme dialect has explicit declarations for these types)
- variant types
- functions consuming and producing records and variants
- case study: game of Nim
- polymorphic contracts for pairs to prepare for polymorphic lists
- pairs and lists (recursive types)
- functions consuming and producing lists
- recursion on (natural) numbers
- properties of evaluation: nontermination, recursive and iterative evaluation, tail recursion
- mathematical excursion: relations, ordering, and inductive definitions; induction on numbers and lists
- case study: towers of Hanoi
- functional programming: functions as data, higher-order functions
- abstracting over functions (`filter`, `fold`), anonymous functions
- function composition, iterated function composition, currying
- time-dependent models: animation, model, and view
- visual recursion: simple fractals (not in DMdA)
- recursive data: binary trees
- consuming and producing binary trees
- search trees
- case study: Huffman trees
- mathematical excursion: trees, terms, term induction, Σ algebras, basic properties
- abstract datatypes, encapsulating functions with data (*e.g.*, packaging the ordering with a search tree)
- different implementations of an ADT signature
- computability, halting problem, complexity (by term induction), sorting lists and its complexity (Freiburg only)
- assignments and state, `set!`, encapsulated state
- extended substitution model with a store (Freiburg), environment model (Tübingen)
- excursion: parallel execution with assignment to shared state
- object-oriented programming: message passing, inheritance, single vs. multiple inheritance, overriding methods, mixins (Freiburg only)
- λ calculus
- interpretation: BNF, symbols and quote, representation of syntax trees, closures, definitions (Freiburg only)

2.3 Assisted Programming

During an assisted programming sessions students solve a set of programming exercises under the supervision of a doctoral student assisted by one or two teaching assistants (TAs). The assistance provided by the supervisors is geared to design recipes as they provide a valuable orientation for quickly gauging how far a student had gotten solving the problem. The supervisors ensure that the students follow the design recipes. The visible traces of the design recipes such as test cases, contracts, and short descriptions make it easier to understand the student's problem and provide useful help. For example, a wrong understanding of the problem statement often leads to wrong test cases (that must be specified before writing the code). This may be discovered and discussed with the student just by inspecting the test case—a useless and time-consuming inspection of the source code becomes superfluous.

Assisted programming sessions take place in the department's computer lab. Each student (20-25 per session) works on her own machine. We use a dedicated environment on the lab machines that allows no network traffic (except for connections to a special server from which lecture material is available), and through which students can start only the programming system (DrScheme in our case), a web browser, and a PDF viewer. Moreover, the environment enforces a strict time limit of 90 minutes (Freiburg) or 120 minutes (Tübingen). The dedicated environment has two main advantages:

- Students cannot do anything except working on their programming problems: they cannot surf the web, they cannot check their emails, they cannot chat and so on.
- Students cannot cheat easily by exchanging their solutions with other students. This is especially important if the different sessions happen on different days of the week.

The programming problems posed in the assisted programming sessions are closely connected with the topics of the lecture (Section 2.2). For the theoretical parts of the lecture (mathematical basics, terms and Σ algebras, λ calculus), we either pose problems that tie together different aspects of previous sessions, or that put theory into practice. For example, when dealing with the λ calculus in the lecture, we implement an interpreter for λ terms in the assisted programming session.

3. Local Specifics

There are several differences between instances of the course in Freiburg and Tübingen.

3.1 Freiburg

In this section, we describe the intro course as it took place in the winter semester 2007/2008 at the University of Freiburg. The course was worth 9 ECTS credits, which corresponds to 225-270 hours of work in the whole semester.[1] The teaching period encompassed 15 weeks, excluding a mid-term break of about two weeks. Participants of the course were bachelor students but also students who do computer science only as their minor subject.

The students' workload consisted of the following parts:

Lectures There were two 90-minute lectures per week. See Section 2.2 for a description of the material covered.

Exercise sheets and exercise sessions Each week, we published an exercise sheet which the students had to solve within one week. To be admitted to the final exam, each student had to reach a score of 50% of the maximum before and after the mid-term break. Discussion of the solutions to the exercise sheets took place during a 90-minutes exercise session among 15-20 students, guided by a teaching assistant.

Assisted programming Students had to participate in a weekly assisted programming session of 90 minutes. As for the exercise sheets, each student had to reach a score of 50% before and after the mid-term break to be admitted to the final exam.

System design project In cooperation with the Department of Microsystems Engineering, we offered a system design project, in which students worked together in groups of four to design and build an autonomous robot using LEGO mindstorms [21]. The

[1] Note that two credits in the UK are equivalent to one ECTS credit. For more information see `http://ec.europa.eu/education/programmes/socrates/ects/index_en.html`.

main goal of this project was to stimulate team spirit and team work among the students.

Exams There were three exams: one practice exam before the mid-term break, another practice exam at the end of the teaching period, and the final exam six weeks after the teaching period. Students needed to pass at least one of the practice exams to be admitted to the final exams. Moreover, if a student managed to pass both practice exams, the final exam became optional. The final grade was calculated as the weighted average of the grades in the final exam (90%) and the system design project (10%) or, if the student passed both practice exams, the grades in the practice exams (45% each) and the system design project (10%), whichever gave the better result.

We used a semi-automatic tool for marking the students' submissions to programming problems. The tool parses the input file, verifies that the code meets the required language level, and then performs several checks [4]:

- Does the function and record definitions asked for in the problem formulation exist?
- Does the student provide enough test cases? For every problem, we specified how many test cases were needed.
- Do the student's functions satisfy the student's test cases? This checks that the student's solution is coherent.
- Does our sample solution satisfy the student's test cases? This checks that the student's test cases are not biased towards the student's solution.
- Does the student's function satisfy our test cases? This checks whether the student's solution is, in some sense, correct.

Based on the outcome of these checks, the tool computes a preliminary score. The score is zero if the input file has syntax errors or does not meet the required language level. The teaching assistant then examines the solution to ensure that it meets certain conditions that are hard to check automatically:

- Is the code properly documented?
- Is every function equipped with a type signature?
- Does the code meet our coding conventions?

After this visual inspection, the teaching assistant assigns the final score.

Using a semi-automatic tool for marking the submissions had several advantages. First, it reduced the workload of the teaching assistants significantly. Second, it forced the students to work more precisely because the tool enforced our requirements on syntax and functionality of the solution more rigorously than teaching assistants would when marking completely by hand.

The semi-automatic tool had also some disadvantages. First, it required more time to work out the programming problems because we also needed to write the specifications for our tool. Second, we often had to slightly overspecify the problems to allow for a sensible specification. For example, we often had to fix the name of some function or the ordering of the fields of some record.

3.2 Tübingen

This section describes the first year programming course of the winter semester 2006/2007 at the University of Tübingen.

The attendees of this course were diploma and bachelor students. All students were required to pass the exam at the end of the course. Another requirement was to get credit for either the first- or second-semester course. For the bachelor students, the course was worth 8 ECTS credits. The length of the teaching period and the mid-term break were the same as in Freiburg.

The students' workload consisted of the following parts:

Lectures As in Freiburg, there were two 90-minutes lectures per week.

Exercise sheets and exercise sessions As in Freiburg, we published weekly exercise sheets, which the students had to solve within one week. To get the credit, each student had to reach 60% of the maximum score before and after the mid-term break. The students were allowed and encouraged to solve the exercises in a group of two students. As in Freiburg, there were weekly exercise sessions guided by a TA. Furthermore, to receive credit, the students were obliged to attend the exercise sessions and each students had to present her solution in front of the group twice during the teaching period.

Two of the exercise sheets where *mandatory*: They did not contribute to the score. Instead, each student had to present the solution separately to the TA, and the TA needed to be "satisfied" with the solution. In theory, failing to do so meant that credit was denied but in practice the student would often get a second chance. The first mandatory exercise sheet took place very early in the semester and was intended as an opportunity for the TA to get to know each student individually. The second mandatory sheet was the last sheet given out and covered a larger project. The students had two weeks to solve the second mandatory sheet.

Assisted programming As in Freiburg, we conducted weekly assisted programming sessions. However, ours took 120 minutes. As for the exercise sheets, each student had to reach 60% of the maximum score before and after the mid-term break to get the credit.

Exams and grades There was a single exam about four weeks after the teaching period. For the diploma students, passing this exam was necessary for the intermediate diploma which in turn was a requirement for the diploma itself. However, the grades from the intermediate diploma did not contribute to the grades of the diploma. For the bachelor students, passing the exam was necessary as well and the grade of the exam was relevant for their degree.

The credit itself had a separate grade but this grade did not contribute to the intermediate diploma or the bachelor. Thus the grade was relevant only for students applying for scholarships or the like.

We did not use a tool for marking the students' submissions. Instead the teaching assistant marked each solution to an exercise sheet or assisted programming session manually. (We are considering using the tool for the next iteration, however.)

4. Developing the Course

The current course is based on a development that essentially started with the 1999/2000 Tübingen course: This course marked the switch to functional programming and Scheme, as well as DrScheme as the programming environment. Much of the programming content was modelled on *SICP* [1] and *Concrete Abstractions* [17]. (Note that HtDP was not available in print yet.)

While the use of Scheme enabled us to cover much more material in the course than in previous years, we observed that many of the didactic approaches suggested by *SICP* and, to a lesser extent, *Concrete Abstractions*, did not work as well as we had expected. (Of course, the TeachScheme! project had made that observation earlier [11], but we were not aware of these results.)

4.1 The 2004/2005 Course

As a result of the 1999/2000 course, we started the slow process of improving the course over time. We published a book [19] describing the course material. When HtDP finally came out, we adopted the design recipes and made them a central element of the 2004/2005 course, which was the next intro course the authors taught. We structured the course according to the book but augmented the material in the lecture with design recipes. Of course, the examples in the book were not constructed using design recipes and thus had no contracts or test cases. Thus students saw the application of design recipes and their artifacts in the lecture only.

We also used contracts with type variables quite extensively, even while introducing higher-order functions. Type variables were explained in the lecture but again not covered in the book. As it turned out in the exercises and in the exam, the students did not grasp type variables very well.

Thus, augmenting our book with material from HtDP was not perceived as a coherent lecture by the students and many thus formed a negative attitude towards the material.

An initial indication of problems was the general observation that the noise level was quite high during the lecture: many students did obviously not follow the presentation. This was certainly especially harmful as only the lecture presented the material as intended by us. Also, towards the end of the course, many of the students had already reached the 60% limit and stopped to work on the exercises. At this point, the exercises covered the environment model. In the exam, the students performed badly on this topic.

While the course was running, we commissioned a term project to observe how students deal with the material of the course outside of the course and the tutorial sessions [26]. Two observations stood out:

- The students would freely "cooperate" on the exercises, plagiarizing other students' solutions. Networks of students formed which circulated solutions, in one observed case reaching the size of 54 students. This, of cause, voided the intended practice effect of the exercises.
- The students found the design recipes tedious, especially in the beginning when the problems were simple. This led to them ignoring the recipes when actually working on the problems, and adding the required elements (short description, contract, test cases, etc.) later. However, many students said they believed the design recipes would help them later when the problems would get more challenging.

The latter observation was especially galling in light of the results in the final exam: A fair number of students had not successfully solved the programming exercises, and of those who had, many had done so unsystematically, without the use of the design recipes: Even though many students had suspected the design recipes might come in handy when push comes to shove, they could not produce them when the going got tough.

Alarmed by the bad results in the programming exercises, we interviewed each teaching assistant separately and asked them about their experience with the students in their groups. Their statements can be summarized as follows:

1. The TAs did not notice many students having significant problems solving the programming exercises.
2. Many students did not use the design recipes. Instead they augmented their solution with contracts and test cases afterwards.
3. Students with little or no experience who used the design recipes found them useful and were able to solve the exercises with the help of the recipes.
4. There were quite a few students teams (which usually consist of two students) where one student focused on solving the programming exercises while the other one took over the ones related to theory.
5. The TAs did not have the impression that many students plagiarized.

In the light of the results in the exam, we concluded that many students had bluffed the TAs about their programming capabilities. However, we were very encouraged to continue to use design recipes as they obviously provided helpful for the students who used them.

4.2 The 2006/2007 Course: Improvements

The 2004/2005 course was a sobering lesson for us: While the results were formally acceptable, students had not, on the average, learned programming to the level of proficiency we had hoped. This was despite our conviction that we had taught the right material and structured the course appropriately, with proven didactic techniques. We saw two possible explanations for the outcome:

- We were simply expecting too much of the students, and the curse of knowledge was preventing us from seeing clearly what the students were capable of learning in a single semester.
- We had failed to do everything possible to help the students learn.

When it came to preparing the 2006/2007 course, we decided that we would act as though explanation #2 was correct. We already knew two concrete problems: plagiarism and insufficiently interesting examples and problems in the beginning. We also conjectured that many students plagiarized because they did not develop sufficient self-confidence in the first few weeks of the course.

We had the opportunity to revise the old textbook for the course, and rewrote the programming chapters to conform to the HtDP discipline we had already used in the 2004/2005 course: Proper introduction to design recipes, and strict adherence to them in the examples. Developing more interesting examples was quite easy: We adapted HtDP's `world.ss` teachpack for functionally-programmed reactive animations, and used it for introducing state-based models.

Moreover, we concluded that our "buddy approach", trying to encourage rather than force students to do the exercises themselves and ask questions when necessary had failed. We decided to actively fight plagiarism, and enforce a strict discipline both in form and content for the upcoming semester. In particular:

1. We made students sign a form at the beginning of the course declaring that they would not plagiarize, and were aware of the consequences. ("One warning, two strikes you're out.")
2. We would actually kick students out of the course after two strikes of plagiarism.
3. We set aside part of one TA's time to actively search for plagiarism in the students' solutions.
4. We decided to break out parts of the exercises and have the students solve them in the University's computer lab under supervision.[2] This lead to the introduction of assisted programming.
5. We would use the final exam as our measure of success, and differentiate between different subject areas to see how well the course had worked.

[2] The TeachScheme! project, of course, has long used supervised lab exercises to develop their curriculum. Their specific observations are not accessibly documented, however.

The fourth item deserves special attention: We would not only supervise the exercises in the sense of checking that there is no plagiarism, but also look over students' shoulders to actively enforce their adherence to the design recipes, and push them over the inevitable bumps in the road that leads to correct and working programs.

Note this was strictly an experiment at the time: We did not know whether our original explanation for the underperformance was correct, nor whether these new measures would help. In particular, relying on enforcement rather than encouragement for self-managed learning seemed a significant gamble, as self-managed learning and intrinsic motivation generally promise better results [6]. We also risked that we would become unpopular with the students, and that Scheme and/or functional programming would become similarly unpopular by association. We resolved not to care about these aspects for the 2005/2006 course. We did, however, develop these measures with the help of the student TAs, who fully supported them.

4.3 Assisted Programming

Despite our efforts to advertise lab sessions as a form of homework with assistance, most students initially regarded lab sessions as a kind of quiz. As a consequence, many students avoided asking the TAs for help during the session: They either expected that TAs—as with an official written exam—were not allowed to provide concrete help or they even believed that asking for help was a form of cheating. The perception of assisted programming slowly changed during the semester as the TAs not only provided their help upon request but also helped proactively as they noticed students having problems.

The time limit for assisted programming sessions provided additional help nudging the students towards using the design recipes: Some students with an initial resistance to the recipes would say something like "I'd rather do it myself" when offered help. But they quickly realized that they would run out of time using their preferred method of programming-by-tinkering, and so they tried the design recipes in the end. Many were shocked when they found out that the recipes worked.

In our experience, designing problem sets with a suitable amount and difficulty of problems is challenging: The problem should be new and interesting, solvable with the programming techniques taught in the lecture, and there should be enough time for students to make use of the assistance of the TAs. We realized that it is hard to design the exercises so that they are both challenging to the students and solvable within the restricted time frame of an assisted programming session. Typically, we conducted the first session of each week—which starts with a new problem set—ourselves: This way, confusing problem statements, mistakes, amount of work, and other unforeseen problems could be addressed immediately and fixed for the subsequent sessions.

Despite these tactics, we misjudged the complexity of some problem sets in the beginning for a simple reason: Typically, weaker students tend to schedule their lab session at the end of the week — hoping that they benefit from the experiences of other students that attended an earlier session. With hindsight we knew that many of our best students always chose the first lab session—which served as our comparison group to evaluate a problem set's appropriateness. This observation led to two insights: First, we have to take the above-average performance of our comparison group into account. Second, students attending sessions at the end of a week may take advantage of the other students' experiences with the problem set.

However, we discovered only few cases of cheating during assisted programming. The specialized computer environment already prevented that students take home their solutions: Submitted solutions were only accessible by TAs. This hindered plagiarism as solutions must be reproduced from memory. Some students procured such a solution, memorized it, and reproduced it during the lab session. Typically, students share such a solution verbatim, which makes it easy to identify the solution using our routine check for plagiarism. Additionally, we sometimes changed the problem set slightly on an irregular basis during a week.

A common problem consisted of the quantity and quality of assistance provided during a session. We had underestimated the amount of work necessary to attend a group of 25 students during a lab session without having students wait too long for help.

Initially, the introduction of assisted programming encountered some resistance among students. The university calendar announced the course with four hour of lectures per week plus two hours of exercise per week. Our approach, however, requires four hours of presence in exercises per week: two hours for the regular tutorial and two hours for assisted programming. The students' workload, however, is roughly the same as with other courses as we just moved programming exercises from regular exercises to assisted programming.

4.4 Results

Final exam We had determined at the beginning of the course that the final exam would be the primary indicator of success. The Tübingen final exam is one hour, and we posed a total of nine problems in the following subject areas: logical calculus, λ calculus, inductive proofs, ADTs, program proof, (recursive) programming with lists, higher-order programming, programming with state, environment model. While more detailed results are available in Section 5, the results had generally improved significantly over the 2004/2005 course. Moreover, the students' programming abilities were significantly improved. In particular, most students could apply the design recipes with ease.

Popularity Surprisingly, our much more formal approach to organizing the course had no significant impact on the popularity of the course with the students: The course-evaluation poll yielded comparable results to the 2004/2005 course across the board.

Comparing with the OO course The course described here was not taught every year. In particular, the 2005/2006 course was based using a traditional Java-based OO approach. The final exam of that course was comparable in the bureaucratic sense (one hour, same grading scale), but much different in content: six problems instead of nine, in categories: number representations, class relationships, various multiple-choice questions, (manual) tree traversals, recursion, class definition. Subjectively, this final exam was *much* easier than ours. This, and a conversation with the lecturer, led us to conclude that the expectation we had of our students was warranted to be significantly higher than with the OO approach.

The next course The 2007/2008 course in Tübingen was taught by a professor who had no prior exposure to functional programming, and never taught an introductory course before. He chose to follow the content and organization of the previous year quite closely. The exam results (not reproduced in this paper) are completely consistent with those of the 2006/2007 course. Thus, taking into account that several people had held the lecture in previous years, it appears the approach is robust across lecturers.

5. Analysis

In this section, we present some of the data we have gathered to gauge the effectiveness of our teaching methods. In particular, we have tried to measure if assisted programming indeed helped improve the course. We have also looked at the results of our exams to identify subject areas that worked well, as well as those

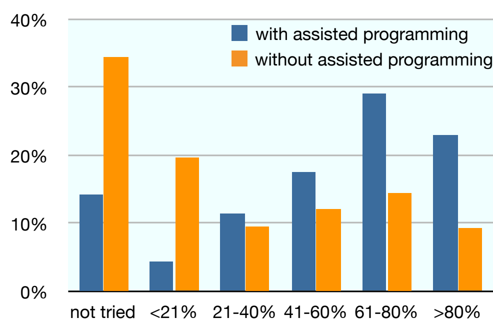

Figure 1. Performance of students in courses with and without assisted programming

that still need improvement. While these results are probably not statistically solid, they provide very useful qualitative information.

5.1 Assisted Programming

Figure 1 compares the students' performance in the programming assignments of the final exams between the 2004/2005 Tübingen course without assisted programming, and the 2006/2007 Tübingen course with assisted programming. The diagram shows how many students (y-axis) achieved what score (x-axis). The special category "not tried" collects students that did not even try to solve the programming assignments. The number of students and the score are given relative to the total number of students and the maximum score, respectively. It is obvious that the performance of the students who benefited from assisted programming is significantly higher than the performance of the students from the earlier course.

5.2 Exam Results

Figure 2 and Figure 3 display the students' performance in the practice and final exams, sorted by problem category. Figure 2 list the results for those categories that are common to Freiburg and Tübingen, whereas Figure 3 contains results for categories specific to Freiburg or Tübingen. The Freiburg results come from the course in 2007/2008, the Tübingen results from the course in 2006/2007.

Here are the most significant results:

- The students did well on the "standard" programming exercises that involved applying the design recipes.
- The students were quite proficient with the λ calculus, despite our expectation that they would perform poorly. (Theoretical, "boring" material.) Our conjecture (for the results in Tübingen) is that this was caused by making the λ calculus subject of one of the mandatory exercises, which was really a timing accident of the 2006/2007 course—one that we gladly repeated in the following year, with similar success.
- The Freiburg result for abstract data types (Figure 2) is probably misleading: The students had not performed well with ADTs during the semester, and such an improvement for the final exam would have been miraculous. The assignment was probably too easy. This insight, taken together with the Tübingen results, show that the treatment of ADTs is problematic and could be improved.
- The Freiburg assignment on terms (Figure 3) dealt with terms and their interpretation via Σ algebras. The students did not perform well in this assignment because they had not completely understood the difference between syntax and semantics.

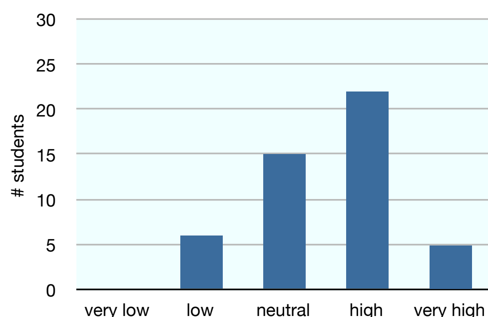

Figure 4. Requirements

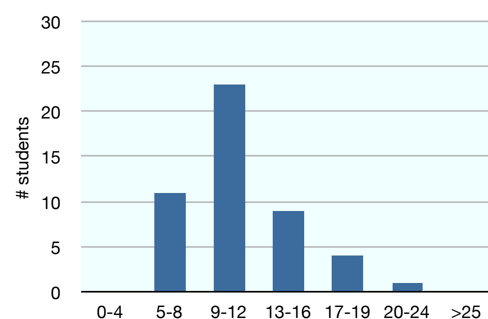

Figure 5. Overall Hours per Week

- For the Freiburg assignment on relations (Figure 3), we conjecture that students did not perform well because they considered this topic as "too mathematical".
- The results for the Tübingen assignment on the environment model (Figure 3) show that the environment model (despite the use of pictures) is not well suited for explaining mutable state.

5.3 Course Evaluation

We next present the results of an evaluation of the Freiburg 2007/2008 course, which was performed at the end of the teaching period. About 90 students attended the course, of which about 50 participated in the evaluation.

Most students judged the requirements of our introductory course as "neutral" or "high", only few students considered the requirements as "very high" (Figure 4). The overall numbers of hours spent per week for our course (including lecture, tutorial sessions, and assisted programming) ranged from 5 to 24 hours, but the majority of the students (about 70% of the evaluation's participants) spent only 12 hours or less (Figure 5).

According to Section 3.1, the course in Freiburg should require (in average) a total amount of 250 hours of work. Subtracting 40 hours for exam preparations, we are left with 14 hours per week. Hence, we conclude that we did not stress the students too much.

There were several questions which allowed a free-text answer. In the following, we summarize the answers to the most interesting questions.

- *What did you like about this course?* Students liked the practical aspects of the course, the assisted programming sessions, and the overall structure of the course. Another positive point men-

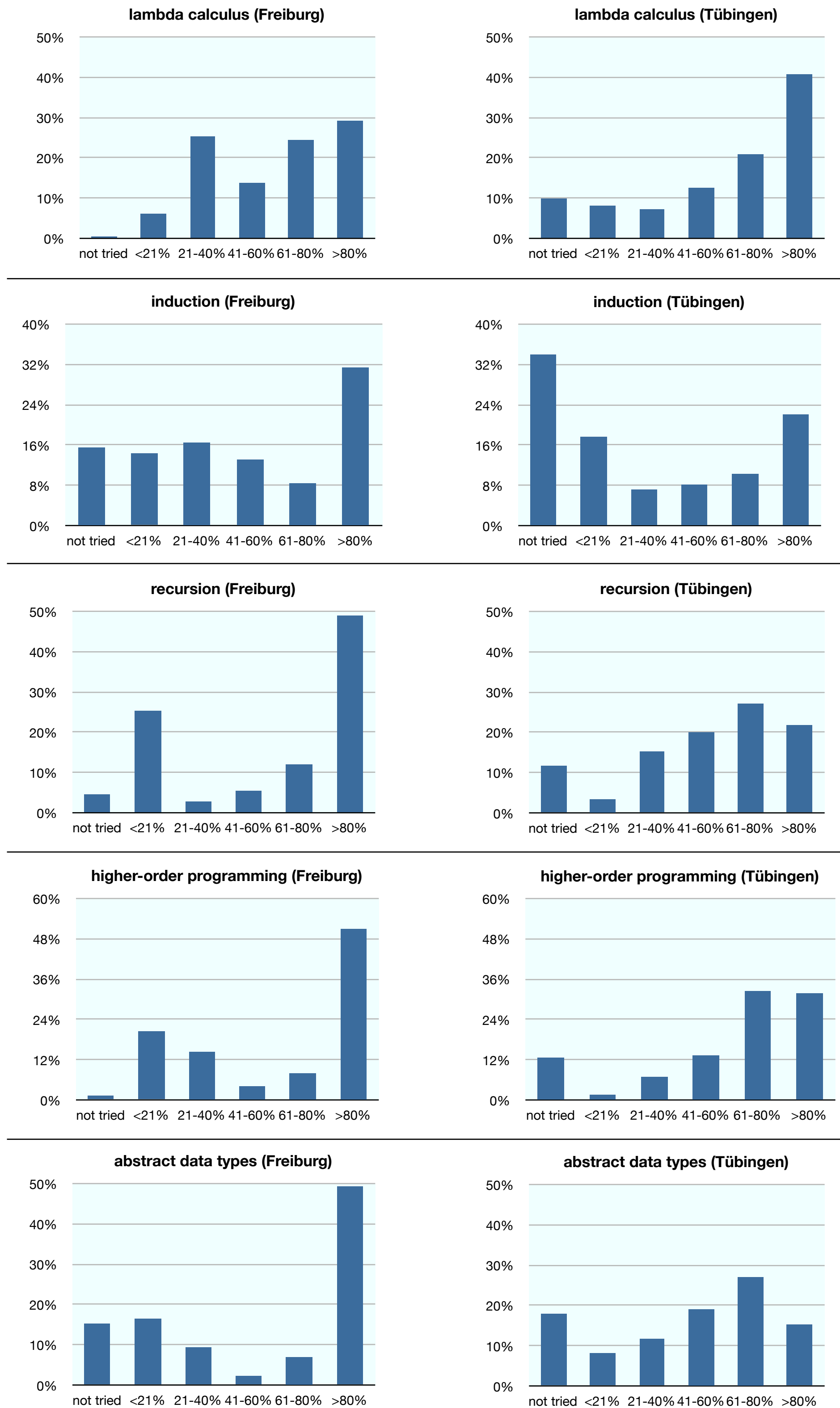

Figure 2. Exam results in conjoint exercise categories

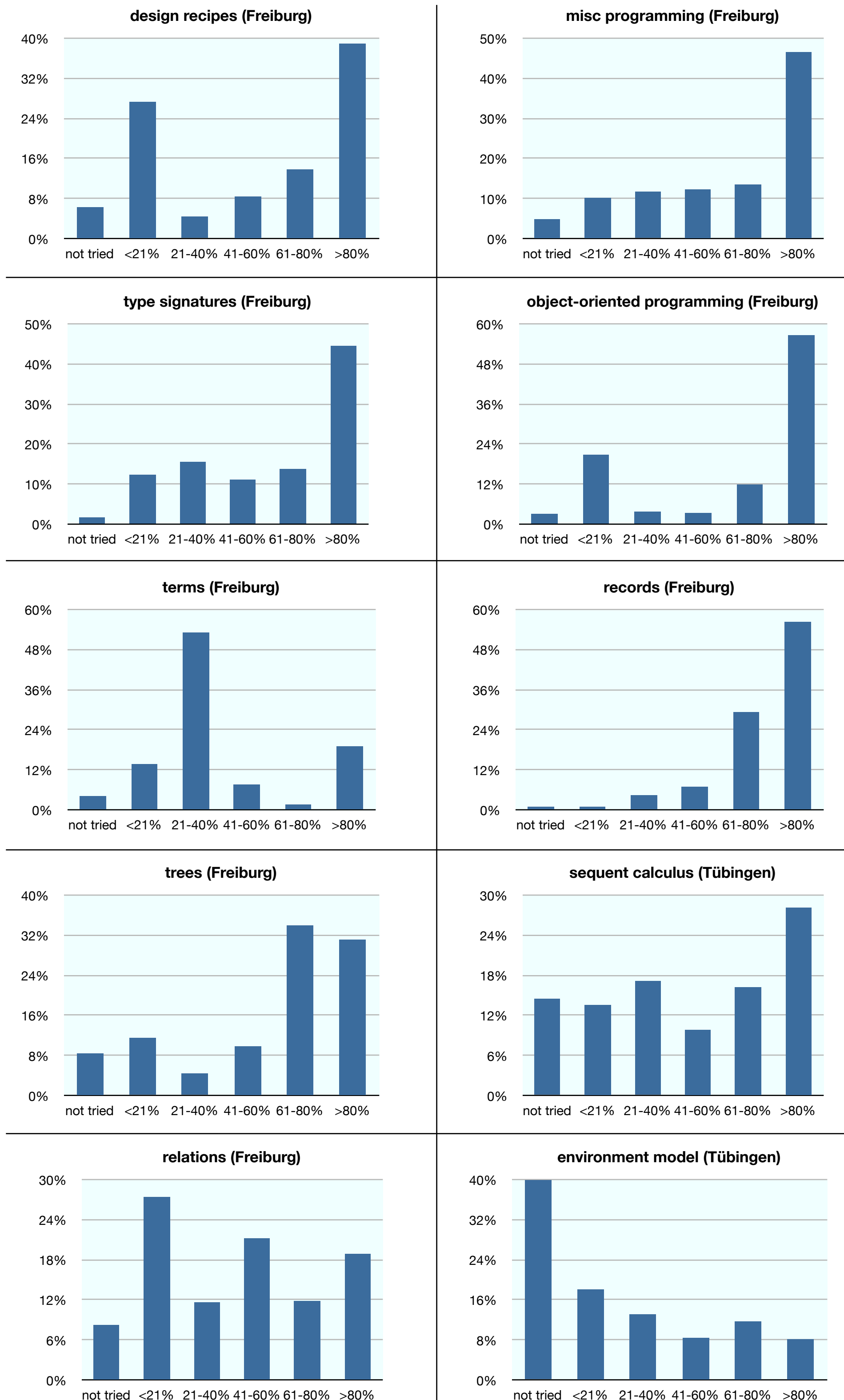

Figure 3. Exam results in disjoint exercise categories

tioned was that we recorded the lecture so that students could review it later.

- *What did you not like about this course?* The students did not like the mathematical excursion. They also complained about Scheme. They were unsure whether they could transfer their knowledge to a different programming language.
- *Did your programming skills improved because of the assisted programming sessions?* Most students stated that their programming skills had improved due to assisted programming. The students mentioned the following reasons for this improvement:
  - Assisted programming forced them to do a great amount of programming.
  - Assisted programming gave them the possibility to ask all kinds of questions.
- *For which other courses would assisted programming be helpful?* The students considered assisted programming helpful for all further practical courses.

The results from Freiburg were remarkably consistent with the results of a similar evaluation in Tübingen.

6. Follow-up Courses

The intro course should not only prepare the students for potential careers in software development, but also provide the necessary prerequisites for follow-up courses in the curriculum. The requirements posed by these courses are different in Freiburg and Tübingen—however, both intro courses have follow-up courses using Java. This section summarizes how we handled the transition.

6.1 Freiburg: Crash Course on OO and Java

In Freiburg, the second-semester course on algorithms and data structures requires students to be fluent in Java. To provide an interface to this course, we set up a one-week full-time adaptation course that translates their knowledge into basic Java proficiency. This crash course is based on the ideas underlying the upcoming book "How to Design Classes" (HTdC) by Felleisen and his collaborators (also documented by Proulx and others [22]), but adds some elements that are dear to the heart of an algorithms instructor. The important topics are, roughly: data modelling with classes, functional methods, abstraction, imperative methods, and mixed material preparing specifically for the algorithms course, with topics such as arrays and typical programming templates that deal with arrays.

The first four days rely on ProfessorJ [15], a teaching language resembling Java which is also implemented in the DrScheme environment with similar support for different language levels and interactive debugging as for Scheme. Day five prepares students for using Eclipse, which is used in the algorithms course.

Following this intensive phase, the course is tapering off at roughly two hours per week throughout the rest of the semester and in parallel to the algorithms course. It goes on to cover advanced topics like abstraction with generics, defining equalities and comparisons, enumerations, using the collection framework, using Swing, inner classes and anonymous classes, and so on.

In an evaluation of the crash course, about 84% of the students answered that they knew enough about Java to be able to follow the course an data structures and algorithms, and that no topic was missing. The remaining 16% missed arrays and hash tables. We discussed arrays in the crash course, but intentionally left out hash tables because this is a major topic of the algorithms and data structures course.

The results of the evaluation indicate that students of our introductory course learn Java (a new programming language) rather quickly. We do not have any hard facts whether this result generalizes to other programming languages, but we will observe the students within the next semesters to see how they perform with respect to this aspect.

6.2 Tübingen: Second-semester Course on OO and Java

In Tübingen, there is a second semester course on programming. We had the opportunity to teach and design this course ourselves. The faculty seems to agree that, by end of the second semester, students should either be fluent in Java or C++, as almost all subsequent courses presume knowledge of these languages. Therefore, we decided to devote the complete second semester to object-oriented programming and Java. The significant topics of the course were similar to the Freiburg course. We added testing with JUnit, GUI programming, the MVC model, debugging tools, and the SECD machine.

In contrast to the Freiburg course the Tübingen course proceeded at a much slower pace. Twelve lectures (90 minutes each) were devoted to HtDC and the remaining seven lectures covered the language features of full Java.

Feedback on this course was mixed: While many students—with and without prior knowledge of Java—actually appreciated the design rigor imposed by HtDC and the insights from "Effective Java" [3], requiring the use of ProfessorJ in the exercises was unpopular. Students with prior knowledge of Java complained about the limitations imposed by the various language levels. Moreover, some students feared that we would not teach "real" Java which would hinder them in future courses or even on the job market. However, students without prior knowledge clearly favored the gradual introduction with ProfessorJ into Java.

7. Conclusions

Teaching the intro course has consistently been a rewarding experience for us, despite the fact that many of our ideas of the past did not work in the classroom as well as we had hoped. However, teaching the course repeatedly and closely looking at the available documentation of the successes and failures can help to improve it significantly. This section describes our most import observations, impressions and conclusions. We start with a short discussion about the choice of Scheme and conclude with some lessons we have learned from the courses, including possible improvements and an outlook on further introductory courses.

7.1 Choice of Programming Language

The general methodology of the approach could also be expressed in other functional languages such as O'Caml or Haskell. For example, pattern matching and algebraic datatypes simplify some of the design recipes considerably. A pattern match clause is more robust than a `cond` expression with a sequence of type tests because the pattern match can be checked for exhaustiveness. Furthermore, it was confusing to the students that there was an explicit syntax for defining record types, whereas sum types had no syntactic counterpart. Their definition is entirely based on programming conventions, for Scheme it is just a comment. An argument against pattern matching is that the languages that students get to use in their later worklife do not have pattern matching, either.[3]

Type inference and type checking could, in principle, avoid frustrating test runs that repeatedly fail because of type errors. On the other hand, type inference is infamous for generating inscrutable error messages. This is known to be especially discouraging for

[3] Matthias Felleisen, personal communication

beginners. In the case of Haskell, the type inference engine may inadvertently reveal aspects of the language (such as type classes) that the instructor does not want to get into, yet. This problem has been recognized [18], but it is not clear that the programming environments for statically typed languages are up to the needs of beginning students.

It may be possible to address the problems of explicit sum typing and checking of conditionals on these types within the language themselves by changing or extending our DrScheme language levels. We intend to experiment with such an extension next semester.

7.2 Methodology vs. Content

One might conclude from the results reported here that our successes are mainly a result of assisted programming as well as other organizational measures we have taken, and that similar results may be achieved with traditional content, say the standard Java-based "OO course". While nobody seems to have actually tried this and documented the results, there are strong indications that this would not work as well as our approach:

- Even before introducing assisted programming, the Tübingen students performed quite well in comparison to students who had attended the Java-based course: In particular, our exam problems have always been significantly more challenging than those of the Java-based course, while our passing rate was as good or better.
- The particulars of Scheme occupy very little time in the lecture, which allows us to cover significantly more ground over the semester. In contrast, Java-based lectures need to spend significantly more time on mundane issues such as syntax or other specific properties of the language.
- Scheme programs are significantly shorter than, say, Java programs: This allows the students to solve more exercises in the same amount of time, again increasing the amount of material we can teach over a single semester.

7.3 Lessons Learned

Here are the most important lessons we have learned from developing the intro course over the last few years.

- Teaching the intro course well is very difficult. Some of the lecturers involved in the course have been teaching programming since the 1980s, and still find things that could be improved.
- Simply switching to a functional language does *not* automatically solve the problems of traditional approaches.
- It is necessary to closely observe learning success; evaluating a course's success by asking students how well they liked the course is not sufficient. Rampant plagiarism can mask teaching failures.
- The TeachScheme! approach is a great prerequisite for teaching a good intro course. However, it needs to be coupled with effective course organization to be successful.
- A programming environment tailored for beginning students is very important. However, many students require more direct help in assisted programming sessions to succeed writing their first programs.
- Students have trouble managing their own learning process. In particular, students often plagiarize instinctively, before having even tried to solve a given exercise. Moreover, many students have trouble properly gauging their own abilities. As a result, many students show an instinctual resistance to the "boring" approach of the design recipes even though they would fail on their own.
- In spite of initial concerns, a formal, enforcement-oriented approach is not unpopular with the students. In contrast, the students welcome the additional assisted programming lessons, even if this means having more workload for the course.
- Improving the intro course is enormously expensive; gaining a new round of experience requires an entire class, potentially using it as the guinea pig for new ideas. The work of the TeachScheme! project is an enormous help. Still, we need to find more effective ways to benefit from each others' experience. This process is uniformly hampered by ideological approaches to designing the course. ("Objects first", "real-world programming in the intro course", "functional programming is better", "static typing is better", etc.)

We realize that these observations may, in some instances be culture-specific. We have consistently observed them in Germany, however.

7.4 Possible Improvements

Here are some improvements we are considering for future iterations of the course:

Treatment of state and assignment Treating `set!` in the course is not a good idea because it complicates the semantics and confuses the students. The next iteration of the course will use records with mutable fields (similar to reference cells in ML) instead. The Freiburg group will continue their work on an extended substitution model with a store, while the Tübingen group will try an alternative approach based on the SECD machine. Experience will show which of these models is better suited for introducing students to assignment and mutable state.

Contracts in the language Contracts are already an essential part of the design recipes. However, they are only comments. A formal notation for contracts or types may improve the experience and practice of writing contracts. Our semi-automatic grading tool could check the consistency of the contracts, as well as checking the procedures against their contracts during testing. Moreover, an automatic testing tool could derive tests from the types following the ideas of QuickCheck [5].

ADTs The treatment of ADTs has not worked as we had hoped. This is possibly intrinsic—the first semester may be too early to introduce the difference between specification and implementation. Furthermore, the language levels do not provide the abstraction facilities necessary for effectively hiding implementation details.

7.5 Outlook

The authors of this paper will teach the intro course again in the 2008/2009 semester, both in Freiburg and Tübingen. We plan to make changes according to the insights described in this paper, as outlined in the preceding section. We will cooperate closely to achieve the best possible results, and reduce the amount of work needed to teach and organize the course. We will again evaluate how well our changes work. Moreover, we are waiting for impressions from the follow-up courses on how well our students perform in the future.

We hope that we can find other introductory courses that are willing to cooperate with us so as to improve the course even further, and to develop a body of material—software, sample exercises, instruction manuals—that help other teachers.

Acknowledgments

We thank Matthias Felleisen and the other members of the PLT team for developing the TeachScheme! approach, and providing help and advice time and again.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., second edition, 1996.

[2] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: a lightweight pedagogic environment for java. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 137–141, New York, NY, USA, 2002. ACM.

[3] Joshua Bloch. *Effective Java*. Prentice Hall, 1st edition, 2001.

[4] Anne Brygod, Titou Durand, Pascale Manoury, Christian Queinnec, and Michèle Soria. Experiment around a training engine. In *Proceedings of the IFIP 17th World Computer Congress*, pages 45–52, 2002.

[5] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Philip Wadler, editor, *Proceedings International Conference on Functional Programming 2000*, pages 268–279, Montreal, Canada, September 2000. ACM Press, New York.

[6] Richard deCharms. *Enhancing motivation: Change in the classroom*. Irvington Publishers, New York, 1976.

[7] DrScheme: PLT programming environment, 2008. `http://docs.plt-scheme.org/drscheme/index.html`.

[8] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 2nd edition, 1996.

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The DrScheme project: An overview. *SIGPLAN Notices*, 33(6):17–23, June 1998.

[10] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. In *Functional and Declarative Programming in Education (FDPE)*, pages 21–26, 2002.

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, March 2004.

[13] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In Hugh Glaser and Herbert Kuchen, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '97)*, number 1292 in Lecture Notes in Computer Science, Southampton, England, September 1997. Springer-Verlag.

[14] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[15] Kathryn E Gray and Matthew Flatt. ProfessorJ: A gradual intro to Java through language levels. In *OOPSLA Educators' Symposium*, Anaheim, California, October 2003.

[16] Dianne Hagan and Selby Markham. Teaching Java with the BlueJ environment. In *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference ASCILITE*, 2000.

[17] Max Hailperin, Barbara Kaiser, and Karl Knight. *Concrete Abstractions*. Brooks/Cole, 1999.

[18] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In Johan Jeuring, editor, *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop*, pages 62–71, Uppsala, Sweden, August 2003.

[19] Herbert Klaeren and Michael Sperber. *Vom Problem zum Programm*. Teubner, Stuttgart, 3rd edition, 2001.

[20] Herbert Klaeren and Michael Sperber. *Die Macht der Abstraktion*. Teubner Verlag, 1st edition, 2007.

[21] LEGO mindstorms. `http://mindstorms.lego.com/`.

[22] Viera K. Proulx and Kathryn E. Gray. Design of class hierarchies: An introduction to OO program design. *SIGCSE Bull.*, 38(1):288–292, 2006.

[23] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1994.

[24] Charles Reis and Robert Cartwright. Taming a professional ide for the classroom. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 156–160, New York, NY, USA, 2004. ACM.

[25] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/final/r6rs.pdf`, Sep 2007.

[26] Claudius Stempfle. Stoffvermittlung in der Informatik I, February 2006. Term project, University of Tübingen.

[27] Christian Wagenknecht. *Programmierparadigmen*. Teubner, 2004.