The Teaching of TCS at the University of Bremen

Hans-Jörg Kreowski

This report on the teaching of theoretical computer science (TCS) at the University of Bremen concerns mainly the undergraduate phase. There will be eventually a second part completing the picture.

Curricular Requirements

As everywhere, TCS is a constitutional part of the computer science curriculum at the University of Bremen. While most students enjoy their studies of computer science for about 6 years, they run officially for 4 1/2 years. The first 2 years form the undergraduate phase which ends with the "Vordiplom". The graduate phase of 2 1/2 years follows. The last half year of it is devoted to the writing of a diploma thesis. If all this is passed successfully, the "Diplom" is conferred. A study year is split into 2 semesters. The teaching period of the summer term (from April to September) lasts 13 weeks, the teaching period of the winter term (from October to March next year) lasts 15 weeks.

The course programme of the undergraduate phase sums up to 84 semester week hours (SWH's), i.e. 21 hours per week and semester in average. This is devided into 5 areas listed with the number of SWH's and the number of courses in brackets:

Mathematics	22	(4)
TCS	8	(2)
Practical Computer Science	22	(4)
Technical Computer Science	12	(3)
Applied Computer Science (including Computer & Society)	20	(4)

The whole programme is obligatory. Only with respect to one of the courses in applied computer science, the students can choose among various application areas. The "Vordiplom" requires the successful participation in the course programme and an oral examination in each of the 5 areas. Under certain conditions, the oral examinations can be replaced by written exercises.

The structure of the graduate phase is more complicate and cannot be quantified in terms of SWH's properly. The diploma requires – in addition to the thesis – the participation in a students' project, 4 oral examinations and 5 course certificates (confirming the successful participation usually based on written exersises or essays). A students'

project runs for 2 years and consists of 15 to 30 students usually supervised by a professor and an assistant. A typical goal of a project is the development of a software system in some practical or applied context. Together with some introductory courses, the project covers about a third of the graduate phase. The rest consists of courses in theoretical, practical and applied computer science. Usually 2 such courses provide the subject of an oral examination. None of the courses is obligatory, but the subjects of the oral examinations and of the certified courses are required to stem from pairwise different subareas. One of the oral examinations and one of the course certifications are requested for TCS, the same for applied computer science and twice the same for practical computer science. The fifth certified course can belong to any area. In TCS, the subjects can be chosen out of the 4 subareas *algorithms and complexity, formal languages, theory of programming,* and *others* (like Petri net, theory of concurrency, theory of rule-based systems, syntactic methods of picture generation, etc. depending on the courses on offer).

The Aims of the TCS Courses for Undergraduates

The 2 TCS courses in the undergraduate studies are meant to introduce the foundations of computability, complexity and formal languages. They are presented in the 2nd and 3rd semesters and consist each of lectures and exercises (both 2 hours per week). Exercises take place in small groups of about 15 students. According to the regulation of computer science studies, the goals of the TCS courses are:

- examplary knowledge and insights that are typical for the field,
- knowledge of basic facts, which are useful in various contexts and shed some light an the relationship of TCS to other areas,
- understanding of the mathematical foundations and basic questions of computer science, in particular, of sense and nature of exactness, precise conception and proving,
- understanding of the interrelation between intuition and firm insights.

Although I was involved in the formulations of these aims and I still believe that they are reasonable, my personal aims are more modest whenever I teach the courses (what I do quite regularly). Otherwise the job would be too frustrating. The number of participants is about 100. Unfortunately, one cannot expect that most of them are well-motivated and enthusiastic about TCS. On the contrary, many are indifferent and ignorant, some are even horrified, and only a very few are really interested. But all of them may change their mind during the courses. Hence I try very hard to keep the interested interested and to convince the others that TCS is not so bad. Hence I try very hard to present the material in a motivating, attractive and entertaining way (as far as the topics and my abilities permit). My intention is that all students or at least as many as possible learn the fundamental facts of TCS and get a good idea of their value.

I dare say, however, that I lose more and more my confidence in the effectiveness of these efforts. The results of the written exercises and the oral examinations are acceptable and the students' attitudes towards TCS seem to be improved after the courses. But whenever I meet the same students later in the graduate courses, the have lost nearly all remembrance of the learnt notions, results and fundamental ideas.

The Contents of the TCS Courses for Undergraduates

In the first course, conplexity and computability are introduced and discussed based on the equational specification language CE-S, which is particularly tailored for the purposes of this course, and on the small imperative programming language designed by Kfoury, Moll and Arbib for their book on *Programming Approach to Computability* (published 1982 by Springer). This provides an intuitive approach to the issues of computability and complexity as well as to fundamental aspects of formal semantics and correctness and allows to relate the considerations with the students' experiences in every-day programming.

Kfoury's, Moll's and Arbib's imperative language has got a state transition semantics that can be extended into a proper denotational and fixed-point semantics and yields the notion of computable functions on (non-negative) integers. The language is well-suited to handle the basic questions of computability including the halting problem and the existence of universal functions and to explain the role of Church's thesis.

The language CE-S is based on the data types of strings, integers and Boolean values and allows to specify operations on these types by means of conditional equations. Using term rewriting, one gets an interpreter for CE-S that illustrates the concept of operational semantics. If one combines the equational calculus with the inductive definition of strings, one gets a proof technique to show the correctness of specified functions with respect to requirements that are also given in the form of equations. Moreover, by re-interpretation of the defining equations, one obtains recurrence equations for the time complexity of the specified functions in terms of numbers of applications of equations. In nice cases, the recurrence equations allow to deduce proper upper bounds for the time complexity. This approach is quite intuitive because the recurrence equations for the complexity functions reflect the work of the interpreter. It allows to analyse complexity in nontrivial cases explicitly within the introduced framework. The results coincide with classical notions of complexity whenever the steps of the intepreter can be performed in constant time.

A typical example of a specification in CE-S is the following specification of sorting by merging:

mergesort

uses: **merge**,**left**,**right**
opns: $sort : A^* \longrightarrow A^*$
vars: $x \in A,\ u \in A^*$
eqns: $sort(\lambda) = \lambda$
$sort(x) = x$
$sort(u) = merge(sort(left(u)), sort(right(u)))$ if $length(u) \geq 2$

where *merge*, *left* and *right* are specified elsewhere. The operation *merge* merges 2 strings respecting the order of the alphabet. The operations *left* and *right* partition a string in left and right halves with $u = left(u)right(u)$ and $length(left(u)) = n$ if u has *length* $2n$ or $2n + 1$. Using this, the last equation induces the recurrence

$$T_{sort}(2n) = T_{merge}(2n) + T_{left}(2n) + T_{right}(2n) + 2\ T_{sort}(n) + 1$$

where $T_{op}(m)$ denotes the complexity function of the operation *op* applied to the sum of the lengths of the string arguments of *op*. By a straightforward induction, one can show that T_{sort} is of order $n \cdot log\ n$ (proving in advance in the same way that T_{merge}, T_{left} and T_{right} are of linear order).

Moreover, matrix multiplication including Strassen's algorithm is analysed to address complexity issues with respect to a famous example as encountered in the literature. And again with the aim to relate the presentation of the course with a more traditional approach, the notion of Turing machines is introduced and shown to be equivalent to CE-S specifications.

As the following table of contents shows, the second course is organised in the traditional way of formal language theory using the problem of syntax analysis as the thread and picture generation by means of chain code picture languages as a second source of motivation:

1. syntax of programming languages and syntax analysis
2. Chomsky grammars
3. examples
4. recursive enumerability of typ 0 languages
5. word problem of monotone languages
6. finite automata
7. rightlinear grammars
8. regular languages
9. push-down automata
10. translation of contextfree grammars into push-down automata
11. leftmost derivations
12. contextfreeness lemma
13. Cocke-Younger-Kasami algorithm
14. derivation trees
15. pumping lemma
16. undecidable problem for contextfree grammars

For both courses, the students are provided with lecture notes in printed form, but which are also available via access to my homepage. The first chapter is devoted to the motivation of TCS including a series of scenarios. Each scenario sketches a fictive situation in which some data processing problem occurs. And it ends with the question *Is this possible?* or *How can this be done?*

Instead of a Conclusion

As I mentioned above, I am not fully content with the outcome of teaching the undergraduate TCS courses. I think that the conception is OK and mirrors my understanding of TCS properly. Nevertheless, something is wrong. The simple reason may be that I spend much more time to prepare my lectures than each student to get the facts and essence of the discussed issues. I am interested in TCS, they doubt its usefulness. I love it, they rather fear and dislike it. I am not willing to accept the students' ignorance because TCS offers quite important and helpful knowledge and methods. But what more can be done about it?