Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at http://www.springer.com/series/7592

Bernhard Reus

Limits of Computation

From a Programming Perspective

Springer

Bernhard Reus
Department of Informatics
School of Engineering and Informatics
University of Sussex
Brighton
UK

Series editor
Ian Mackie

Advisory Board
Samson Abramsky, University of Oxford, Oxford, UK
Karin Breitman, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
Chris Hankin, Imperial College London, London, UK
Dexter Kozen, Cornell University, Ithaca, USA
Andrew Pitts, University of Cambridge, Cambridge, UK
Hanne Riis Nielson, Technical University of Denmark, Kongens Lyngby, Denmark
Steven Skiena, Stony Brook University, Stony Brook, USA
Iain Stewart, University of Durham, Durham, UK

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-27887-2 ISBN 978-3-319-27889-6 (eBook)
DOI 10.1007/978-3-319-27889-6

Library of Congress Control Number: 2015960818

© Springer International Publishing Switzerland 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by SpringerNature
The registered company is Springer International Publishing AG Switzerland

Foreword

Computer Science centers on questions about computational problems, and computer *programs* to solve problems:

- *What* is a computational problem, what is an algorithm, what does it mean to have solved a problem, are some problems unsolvable by any computing device, are there problems intrinsically difficult or impossible to solve automatically, are there problems intrinsically easier to solve than others, …?
- *How* can a machine solve a problem, how to build such a machine, how to specify an algorithm for computer execution, how to design good algorithms, which "language" can a human use to direct the computer, how to design and "debug" programs for computer execution, how to build good programs, …?

Good news: rapid progress has been made in both areas; we stand on the shoulders of giants in both theory and practice. The questions above have many and various answers. The first questions led in mathematical directions to foundational studies: the theories of computability, recursive functions, automata theory and more. The second led in engineering directions: computer architectures, the architectures of programming languages, the art or discipline of programming, software engineering and more.

Since the 1930s our understanding of both areas has developed hand in hand, led by theoreticians such as Kleene, Church, and Gödel; by hardware and software inventors such as Babbage, Von Neumann, and McCarthy; and by Alan Turing's genius at the borderline between the two areas.

This book focuses on the first question area by an approach near the borderline: the theory of computability and complexity (C&C for short) is presented by using a simple programming language. In this language one is able to perform the (many) program constructions needed for the theory. This is done abstractly enough to reveal the great breadth and depth of C&C. Further, it is done concretely and precisely enough to satisfy practice-oriented readers about the constructions' feasibility. Effect: a reader can see the relative efficiency of the constructions and understand the efficiency of what is constructed.

My 1997 C&C book was a step in this direction, but suffers from several problems: a scope too great for a single-semester university course; sections that (without warning) require more mathematical maturity than others; too few exercises; and too few historical and current references to research contexts.

Bernhard Reus has succeeded very well in overcoming these problems, and writing a deep, interesting, up-to-date, and even entertaining book on computability and complexity. I recommend it highly, both for systematic study and for spot reading.

Neil D. Jones
DIKU, University of Copenhagen

Preface

About 12 years ago a student[1] asked me after one of my lectures in *Computability and Complexity* why he had to write tedious Turing machine programs, given that everyone programmed in languages like Java. He had a point. What is the best way to teach a *Computability and Complexity* module in the twenty-first century to a cohort of students who are used to programming in high level languages with modern tools and libraries; to students who live in a world of smart phones and 24h connectivity, and, even more importantly maybe, who have not been exposed to very much formal reasoning and mathematics?

Turn the clock back only two or three decades. Then, a first year in a computer science Bachelor degree mainly consisted of mathematics (analysis and linear algebra, later discrete maths, numerical analysis, and basic probability theory). When computability and complexity was taught in the second or third year, students were already acquainted with a formal and very mathematical language of discourse, maybe because computer science lecturers in the 80s were usually mathematicians by trade. Things have changed significantly. Curriculum designers for Bachelor degrees are under pressure to push more and more new exciting material into a three-year degree program that should prepare students for their lives as working IT professionals. Any new module moved into the curriculum necessarily forces another one out. Often, allegedly "unpopular" modules, including formal theory and mathematics, are the victims. As a consequence, computer science students have to a degree lost the skills to digest material presented in an extremely formal and symbolic fashion while, at the same time, they are prolific programmers and quite knowledgeable in the use of tools.

So, *Computability and Complexity*, do they really have to be taught using Turing machines or μ-recursive functions? Do they have to be presented in the style a logician or mathematician would prefer? Seeking for alternatives, I eventually stumbled across Neil Jones' fantastic book *Computability and Complexity—From a Programming Perspective*. The subtitle already gives away the book's philosophy.

[1] Alexis Petrounias.

The leitmotif of Neil's textbook is to present the most important results in computability and complexity theory "using programming techniques and motivated by programming language theory" as well as "using a novel model of computation, differing from traditional ones in crucial aspects".[2] The latter, `WHILE`, is a simple imperative language with one datatype of lists (s-expressions) à la LISP. Admittedly, this language is not Java, but it has the hallmarks of a modern high level language and is infinitely more comfortable to program in than Turing machines or Gödel numbers. Java, or any similar powerful language, would be impractical for our purposes "since proofs about them would be too complex to be easily understood".[3]

So when rebranding the module under the name *Limits of Computation* in 2008, I adopted Neil's book as course textbook. Delivering an introductory, one semester final-year module, I picked the most important and appealing chapters. This was easy as the design of the book was exactly made to mix and match.[4] Soon, however, it turned out that students found Neil's book tough going. In fact, this became more apparent as the years went past. There were several factors. First of all, Neil's students would have had ML, a functional language with built in list type, as a first programming language, whereas our students were raised on Java. Yet, the datatype of `WHILE` is a functional one, and this caused more problems to the students than anticipated. Second, and more importantly, I had not put enough attention to the prerequisites. Neil expected readers of his book to be senior undergraduate students "with good mathematical maturity."[5] It turned out that not all the third-year students had this maturity (given the heterogeneity of backgrounds and reduction of maths teaching in the undergraduate years one and two).

As a response to mitigate the issues above, I started writing explicit notes to accompany my slides, intended as additional comments and explanations for the selected book chapters. I ended up adding more and more new material and rearranging it. The results of this effort are the 23 individual lectures of this book. A (British) semester is 12 weeks long, which usually requires 24 lectures to be delivered. The shortfall of one lecture is intentional, it acts as a buffer (in case things take more time) and also allows for extra events like invited talks or in-class tests.

This book was heavily influenced by Neil Jones's textbook, which is clearly visible in some chapters. To pay homage to his book, its telling subtitle "From a Programming Perspective" has been adopted.

Brighton Bernhard Reus
November 2015

[2]Preface of Neil's book, page x.

[3]The book "Understanding Computation From Simple Machines to Impossible Programs" by Tom Stuart, published by O'Reilly in 2013, appears to follow the same idea and philosophy, using Ruby as programming language. It does not deal with complexity however.

[4]Neil's Preface, page xii.

[5]Neil's Preface, page xiii.

For Tutors

The students using this book are expected to be senior undergraduates who can master at least one imperative programming language. They are supposed to know arithmetic, Boolean algebra, graphs and some basic graph algorithms. Similarly, knowledge of basic set theory, function, and relations is needed, but Chap. 2 contains a short summary of basic definitions used in the book. Some exposition to formal reasoning would be helpful as well, but is not strictly required. To understand the probabilistic complexity classes in Chap. 21 some basic knowledge of probability theory will be needed.

This book is divided into 23 chapters with the intention that each chapter corresponds to one lecture. If not all chapters can be delivered, the following chapters can be omitted without interrupting the natural narrative:

- Chapter 10, *Self-referencing Programs*, but the self-producing program is a brain teaser which turns out to be very popular with students;
- Chapter 21, *How to Solve* ***NP****-complete Problems*, which may, however, be the chapter that has the most impact on students' future projects;
- The last two chapters *Molecular Computing* (Chap. 22) and *Quantum Computing* (Chap. 23), but students find this material particularly exciting.

A few chapters are significantly longer than others. These are Chap. 11, *The Church-Turing Thesis*, Chap. 16, *Famous Problems in* ***P***, Chap. 20 *Complete Problems*, Chap. 21 *How Solve* ***NP****-complete Problems?*, and Chap. 22 *Molecular Computing*. The longer chapters allow the tutor to pick some of the sections and present them in the lecture, leaving the remaining ones for self-study or exercises.

Acknowledgements

This book would not have been possible without the help and influence of so many people. They all deserve a big thank you.

First of all, I wish to thank Neil Jones for his book on *Computability and Complexity*, which has been inspirational. The results of many brilliant researchers have been reported in this textbook. Many of them have been mentioned and referenced but, this being an introductory textbook, some of them may have been left out. My sincerest apologies to those who were omitted.

I am grateful for the feedback I have received over the years from my teaching assistants Matthew Wall, Jan Schwinghammer, Cristiano Solarino, Billiejoe Charlton, Ben Horsfall, and Shinya Sato.

I enjoyed and benefited from talking to many students on the *Limits of Computation* course I taught at Sussex. Thanks go in particular to Alexis Petrounias, Marek Es, Thomas Weedon Hume, Alex Jeffery, Susan Coleman, Sarah Aspery, Benjamin Hayward, Jordan Hobday, and Lucas Rijllart. The latter five also gave feedback on early versions of various chapters.

I also benefited from discussions with Thomas Streicher and my colleagues at Sussex: thanks go to Martin Berger, Matthew Hennessy, Julian Rathke, Des Watson, and particularly, George Parisis. Des and George also provided most welcome proof-reading services.

Luca Cardelli was kind enough to discuss current topics in molecular computing and gave valuable pointers regarding Chap. 22. Neil Jones and Matthew Hennessy provided comments on an entire draft of this book which helped improve the presentation. All remaining errors are of course solely mine.

I would also like to thank series editor Ian Mackie for his encouragement, Helen Desmond from Springer for her continuous support and Divya Meiyazhagan from the production team for all the last minute edits. The wonderful *LaTeX* typesetting and the Ti*k*Z (PGF) vector graph drawing systems have been used. Thanks to all those people who contributed to their development.

My sister's family and my friends deserve acknowledgment for moral support, and for being there for me when it counted.

Finally, I would like to dedicate this book to the memory of my parents and my brother. His interest in the sciences and in computing aroused my curiosity already at a young age.

Contents

Part II Complexity