Systematic Program Design

From Clarity to Efficiency

A systematic program design method can help developers ensure the correctness and performance of programs while minimizing the development cost. This book describes a method that starts with a clear specification of a computation and derives an efficient implementation by step-wise program analysis and transformations. The method applies to problems specified in imperative, database, functional, logic, and object-oriented programming languages with different data, control, and module abstractions.

Designed for courses or self-study, this book includes numerous exercises and examples that require minimal computer science background, making it accessible to novices. Experienced practitioners and researchers will appreciate the detailed examples in a wide range of application areas including hardware design, image processing, access control, query optimization, and program analysis. The last section of the book points out directions for future studies.

Yanhong Annie Liu is a Professor of Computer Science at Stony Brook University. She received her BS from Peking University, MEng from Tsinghua University, and PhD from Cornell University. Her primary research has focused on general and systematic methods for program development, algorithm design, and problem solving. She has published in many top journals and conferences, served more than fifty conference chair or committee roles, and been awarded more than twenty research grants in her areas of expertise. She has taught more than twenty different courses in a wide range of Computer Science areas and presented close to a hundred research talks and invited talks at international conferences, universities, and research institutes. She received a State University of New York Chancellor's Award for Excellence in Scholarship and Creative Activities in 2010.

SYSTEMATIC PROGRAM DESIGN

From Clarity to Efficiency

Yanhong Annie Liu
Stony Brook University, State University of New York

CAMBRIDGE UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314-321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi - 110025, India

103 Penang Road, #05-06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107036604

First published 2013

A catalogue record for this publication is available from the British Library

Library of Congress Cataloging in Publication data
Liu, Yanhong Annie, 1965–
Systematic program design : from clarity to efficiency / Yanhong Annie Liu, Stony Brook University, State University of New York.
pages cm
Includes bibliographical references and index.
ISBN 978-1-107-03660-4 (hardback) – ISBN 978-1-107-61079-8 (paperback)
1. Computer programming. 2. System design. I. Title.
QA76.6.L578 2013
005.1–dc23 2012047527

ISBN 978-1-107-03660-4 Hardback
ISBN 978-1-107-61079-8 Paperback

To all my loving teachers,
especially my parents,
my Scott, Sylvi, and Serene,
and many of my colleagues and students.

Contents

Preface

Design may refer to both the process of creating a plan, a scheme, or generally an organization of elements, for accomplishing a goal, and the result of that process. Wikipedia states that design is usually considered in the context of applied arts, engineering, architecture, and other creative endeavors, and normally requires considering aesthetic, functional, and many other aspects of an object or a process [319]. In the context of this book in the computing world, design refers to the creation of computer programs, including algorithmic steps and data representations, that satisfy given requirements.

Design can be exciting because it is linked to problem solving, creation, accomplishments, and so on. It may also be frustrating because it is also linked to details, restrictions, retries, and the like. In the computing world, the creation of a computer program to accomplish a computation task clearly requires problem solving; the sense of excitement in it is easy to perceive by anyone who ever did it. At the same time, one needs to mind computation details and obey given restrictions in often repeated trials; the sense of frustration in the process is also hard to miss.

Systematic design refers to step-by-step processes to go from problem descriptions to desired results, in contrast to ad hoc techniques. For program design, it refers to step-wise procedures to go from specifications prescribing *what* to compute to implementations realizing *how* to compute. The systematic nature is important for reproducing, automating, and enhancing the creation or development processes. Clarity of the specifications is important for understanding, deploying, and evolving the programs. Efficiency of the implementations is important for their acceptance, usage, and survival.

Overall, a systematic program design method that takes clear specifications into efficient implementations helps ensure the correctness and performance of the programs developed and at the same time minimize the development cost. In terms of human adventure and discovery, it allows us to be free of tedious and error-prone aspects of design, avoid repeatedly reinventing the wheel, and devote ourselves to truly creative endeavors. It is with these motivations in mind that this book was written, to give a unified account of a systematic method that was developed based on significant prior work by many researchers.

The systematic program design method described in this book applies to large classes of problems of many different kinds; it does not yet do the magic of generating efficient implementations from clear specifications for *all* computation problems, if such a magic method will ever exist. For example, the method can derive dynamic programming algorithms from recursive functions, produce appropriate indexing for efficient evaluation of relational database queries, and generate efficient algorithms and implementations from Datalog rules; however, it cannot yet derive a linear-time algorithm for computing strongly connected components of graphs. It is, of course, not the only method for program design.

The method described in this book consists of step-wise analysis and transformations based on the languages and cost models for specifying the problems. The key steps are to (1) make computation proceed iteratively on small input increments to arrive at the desired output, (2) compute values incrementally in each iteration, and (3) represent the values for efficient access on the underlying machine. These steps are called Step Iterate, Step Incrementalize, and Step Implement, respectively. The central step, Step Incrementalize, is the core of the method. You might find it interesting that making computations iterative and incremental is the analogue of integration and differentiation in calculus. Steps Iterate and Incrementalize are essentially algorithm design, and Step Implement is essentially data representation design.

Overview

This book has seven chapters, including an introduction and a conclusion. The five middle chapters cover the design method for problems specified using loop commands, set expressions, recursive functions, logic rules, and objects, respectively. Loops are essential in giving commands to computers, sets provide data abstraction, recursion provides control abstraction, rules provide both data and control abstractions, and objects provide module abstraction.

Chapter 1 motivates the need for a general and systematic design method in computer programming, algorithm design, and problem solving in general; introduces an incrementalization-based method that consists of three steps: Iterate, Incrementalize, and Implement; explains languages, cost models, as well as terminology and notations used throughout the book; and provides historical and bibliographical notes about the method.

Chapter 2 explains the core step of the method, Step Incrementalize, as it is applied to optimizing expensive primitive and array computations in loops. The basic ideas are about maintaining invariants incrementally with respect to loop increment. Because loops are already iterative, and primitives and arrays are easily implemented on machines, there is little to do for Step Iterate and Step Implement. The method is further illustrated on two examples, in hardware design and image processing. Finally, the need for higher-level data and control abstractions is discussed.

Chapter 3 presents Step Incrementalize followed by Step Implement, as they are used to obtain efficient implementations of set expressions. If a set expression involves a fixed-point operation, Step Iterate easily transforms the operation into a loop. We focus on composing incremental maintenance code in Step Incrementalize and designing linked data structures for sets in Step Implement. The method is applied to two additional examples, in access control and query optimization. The chapter ends by discussing the need for control abstraction in the form of recursive functions, which are optimized in Chapter 4.

Chapter 4 studies Step Incrementalize preceded by Step Iterate, as they are applied in optimization of recursive functions. We concentrate on determining minimum increments and transforming recursion to iteration in Step Iterate, and deriving incremental functions and achieving dynamic programming in Step Incrementalize. Step Implement easily selects the use of recursive versus indexed data structures when necessary. Additional examples are described, in combinatorial optimization and in math and puzzles. We end by discussing the need for data abstraction in the form of sets, which are handled in Chapter 3.

Chapter 5 describes Step Incrementalize preceded by Step Iterate and followed by Step Implement, as they are used together to generate efficient implementations from logic rules. Step Iterate transforms fixed-point semantics of rules into loops. Step Incrementalize maintains auxiliary maps extensively for incremental computation over sets and relations. Step Implement designs a combination of linked and indexed data structures for implementing sets and relations. The method gives time and space complexity guarantees for the generated implementation. We present two example applications, in program analysis and trust management. Finally, we discuss the need for module abstraction in building large applications.

Chapter 6 studies incrementalization across module abstraction, as the method is applied to programs that use objects and classes. Object abstraction allows specification and implementation of scaled-up applications. We discuss how it also makes obvious the conflict between clarity and efficiency. We describe a language for specifying incrementalization declaratively, as incrementalization rules, and a framework for applying these rules automatically. We also describe two example applications, in electronic health records and in game programming. At the end, we show how to use incrementalization rules for invariant-driven transformations in general, and we present a powerful language for querying complex object graphs that is easier to use than set expressions, recursive functions, and logic rules for a large class of common queries.

Chapter 7 takes a deeper look at incrementalization, illustrates the ideas on three sorting examples, describes how program design requires both building up and breaking through abstractions, discusses issues with implementations and experiments for the method, and points out limitations of the method and directions for future studies.

How to use this book

This book can be used for both self-study and course study. It is a dense book, but it is intended for both readers with a minimal computer science background and experienced computer science researchers and practitioners. For course study, the book is intended to suit upper-level undergraduate students and beginning graduate students, but selected parts with simpler examples can be taught to lower-level undergraduate students, and full coverage with all examples can be taught to advanced graduate students.

Each of the five middle chapters is relatively independent of the others, except for some of the language constructs introduced in earlier chapters. Nevertheless, studying the materials in order will help one better understand the design method through preview and review of each chapter.

Each of the middle chapters is organized as follows. First, it introduces the problem and a running example and describes the language constructs handled in that chapter. Then, it presents the ideas and steps of the method as applied to the language constructs handled and illustrates them on the running example and other smaller examples. Next, it gives two or more examples to show either additional aspects or certain interesting consequences of the method. Finally, it puts the chapter in the context of the book to motivate the subsequent chapter. Each chapter ends with bibliographic notes.

Exercises are given at the end of each section, to help readers learn the method discussed. Each exercise is given one of two levels of difficulty: purely for practicing or partly for discovery. Exercises of level one are simple examples for programming or for following the method presented in that section. Exercises of level two can lead to discovery of aspects of programming or of the method not discussed in that section. Exercises of level two are indicated with an asterisk (*).

An index at the end of the book lists the terminology and names used in the book. A boldface number following a term denotes the page where the term is defined, and other numbers indicate the pages where the term is used.

Acknowledgments

It is impossible to thank everyone, in an appropriate order, who helped me work on things that contributed to this book, but I will try.

First of all, I would like to thank Anil Nerode. His enlightening comments and encouragement, ever since my years at Cornell University, with his deep insight from mathematics and logic, open mind on hard practical problems, as well as rich experience working with people, are invaluable for the work that led to this book and beyond. All of these, poured on me during the long hours at each of my visits to him, and unfailingly shown through his instant response to each of my email inquiries and visit requests, makes him to me like a master to a disciple seeking some ultimate truth, not to mention that it was while taking his logic class that I met a classmate to be my love in life.

It was with extreme luck that I went to Cornell University for my PhD, took stimulating classes not only from Anil, but also Dexter Kozen, Bard Bloom, Keshav Pingali, Keith Marzullo, and others, and did my dissertation work with Tim Teitelbaum. Tim challenged me to find general principles underlying incremental computation. He provided me with generous advice and knowledge, especially on how to value the importance of research in terms of both principles and practices. Bob Constable showed great enthusiasm for my work and gave excellent suggestions. David Gries gracefully helped polish my dissertation and offered marvelous humor as an outstanding educator.

Since my dissertation work, I have received many helpful comments and great encouragement at the meetings of IFIP WG 2.1—International Federation for Information Processing, Working Group on Algorithmic Languages and Calculi. Bob Paige and Doug Smith, whose papers I had read with great interest before then, were instrumental in discussing their work in detail with me. How I wish that Bob lived to continue his marvelous work. Michel Sintzoff, Cordell Green, Lambert Meertens, Robert Dewar, Richard Bird, Alberto Pettorossi, Peter Pepper, Dave Wile, Martin Feather, Charles Simonyi, Jeremy Gibbons, Rick Hehner, Oege de Moor, Ernie Cohen, Roland Backhouse, and many others showed me a diverse range of other exciting work. Michel's work on designing optimal control systems and games provides, I believe, a direction for studying extensions to our method to handle concurrent systems.

Many colleagues at Stony Brook University and before that at Indiana University were a precious source of support and encouragement. At Stony Brook, Michael Kifer taught me tremendously, not only about deductive and object-oriented database and semantic web, but also other things to strive for excellence in research; David Warren enthusiastically gave stimulating answers to my many questions on tabled logic programming; Leo Bachmair, Tzi-cker Chiueh, Rance Cleaveland, Radu Grosu, Ari Kaufman, Ker-I Ko, C.R. Ramakrishnan, I.V. Ramakrishnan, R. Sekar, Steve Skiena, Scott Smolka, Yuanyuan Yang, Erez Zadok, and others helped and collaborated in many ways. At Indiana, Jon Barwise exemplified an amazing advisor and person as my mentor; Steve Johnson enthusiastically applied incrementalization to hardware design; Randy Bramley, Mike Dunn, Kent Dybvig, Dan Friedmen, Dennis Gannon, Daniel Leivant, Larry Moss, Paul Purdom, David Wise, and others helped in many ways.

I also benefited greatly from interactions with many other colleagues, including many who visited me or hosted my visits and acquainted me with fascinating works and results: Bob Balzer, Allen Brown, Gord Cormack, Patrick Cousot, Olivier Danvy, John Field, Deepak Goyal, Rick Hehner, Nevin Heintze, Connie Heitmeyer, Fritz Henglein, Daniel Jackson, Neil Jones, Ming Li, Huimin Lin, Zuoquan Lin, David McAllester, Torben Mogensen, Chet Murthy, Bill Pugh, Zongyan Qiu, G. Ramalingam, John Reppy, Tom Reps, Jack Schwartz, Mary Lou Soffa, Sreedhar Vugranam, Thomas Weigert, Reinhard Wilhelm, Andy Yao, Bo Zhang, and others. Neil's work on partial evaluation initially motivated me to do derivation of incremental programs via program transformation. Many other friends in Stony Brook and old friends in Beijing, Ithaca, and Bloomington have helped make life more colorful.

I especially thank colleagues who have given me helpful comments on drafts of the book: Deepak Goyal, David Gries, Rick Hehner, Neil Jones, Ming Li, Alberto Pettorossi, Zongyan Qiu, Jack Schwartz, Michel Sintzoff, Steve Skiena, Scott Stoller, Reinhard Wilhelm, and others who I might have forgotten. Jack Schwartz's comments and encouragement left me with overwhelming shock and sadness upon learning that he passed away soon after we last spoke on the phone. Anil Nerode wrote an enlightening note from which I took the quote for the most important future research direction at the end of the book.

Many graduate and undergraduate students who took my classes helped improve the presentation and the materials: Ning Li, Gustavo Gomez, Leena Unikrishnann, Todd Veldhuizen, Yu Ma, Joshua Goldberg, Tom Rothamel, Gayathri Priyalakshmi, Katia Hristova, Michael Gorbovitski, Chen Wang, Jing Zhang, Tuncay Tekle, Andrew Gaun, Jon Brandvein, Bo Lin, and others. I especially thank Tom for picking the name III for the method out of a combination of choices I had, accepting nothing without being thoroughly convinced, and making excellent contributions to incrementalization of queries in object-oriented programs. Students in my Spring 2008 Advanced Programming Languages class marked up the first draft of this book: Simona Boboila, Ahmad Esmaili, Andrew Gaun, Navid Azimi, Sangwoo Im, George Iordache, Yury Puzis, Anu Singh, Tuncay Tekle, and Kristov Widak.

Scott Stoller deserves special thanks, as a colleague, before that a classmate and then an officemate, and as my husband. He has usually been the first person to hear what I have been working on. He has given me immense help in making my ideas more precise and my writing more succinct, and he has answered countless questions I had while writing this book. He has been a wonderful collaborator and a fabulous consultant. Finally, I thank my parents for designing me, preparing me for both high points and low points in my endeavors, and, perhaps, for unyieldingly persuading me to go to Peking University to study computer science. I thank my two daughters for being so lovely, helping me better understand the need for clear specifications and efficient implementations, and, perhaps, for fighting with my designs from time to time. I especially thank my daughter Sylvi for reading the last draft of this book and giving me excellent suggestions. I thank my daughter Serene for her infinite creativity in keeping herself busy while waiting for me.

Much research that led to this book was supported by the Office of Naval Research under grants N00014-92-J-1973, N00014-99-1-0132, N00014-01-1-0109, N00014-04-1-0722, and N00014-09-1-0651; the National Science Foundation under grants CCR-9711253, CCR-0204280, CCR-0306399, CCR-0311512, CNS-0509230, CCF-0613913, and CCF-0964196; industry grants and gifts; and other sources. Many thanks to my editor at Cambridge University Press, Lauren Cowles, for her wonderful support and advice during the publication process of this first book of mine.