D.E. Rydeheard
R.M. Burstall

Computational Category Theory

PRENTICE HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR

Computational Category Theory

DAVID E. RYDEHEARD
University of Manchester

ROD M. BURSTALL
University of Edinburgh

PRENTICE HALL
NEW YORK LONDON TORONTO SYDNEY TOKYO

First published 1988 by
Prentice Hall International (UK) Ltd,
66 Wood Lane End, Hemel Hempstead,
Hertfordshire, HP2 4RG
A division of
Simon & Schuster International Group

Printed and bound in Great Britain by
A. Wheaton & Co. Ltd, Exeter.

Library of Congress Cataloging-in-Publication Data

Rydeheard, D. E. (David E.)
Computational category theory.
(Prentice Hall international series
in computer science)
Bibliography: p. Includes index.
1. Categories (Mathematics) – Data processing.
2. Electronic digital computers – Programming.
I. Burstall, R. M. II. Title. III. Series.
QA169.R93 1988 511.3 88-4232
ISBN 0-13-162736-8

British Library Cataloguing in Publication Data

Rydeheard, D. E., *1954–*
Computational category theory.
1. Computer systems. Applications of category theory
I. Title II. Burstall, R. M., *1934–*
004'.01'51255
ISBN 0-13-162736-8

1 2 3 4 5 92 91 90 89 88

ISBN 0-13-162736-8

Computational Category Theory

Foreword
John W. Gray

Why should there be a book with such a strange title as this one? Isn't category theory supposed to be a subject in which mathematical structures are analyzed on such a high level of generality that computations are neither desirable nor possible? Historically, category theory arose in algebraic topology as a way to explain in what sense the passages from geometry to algebra in that field are 'natural' in the sense of reflecting underlying geometric reality rather than particular representations in that reality. The success of this endeavor led to many similar studies of geometric and algebraic interrelationships in other parts of mathematics until, at present, there is a large body of work in category theory ranging from purely categorical studies to applications of categorical principles in almost every field of mathematics. This work has usually been presented in a form that emphasizes its conceptual aspects, so that category theory has come to be viewed as a theory whose purpose is to provide a certain kind of conceptual clarity.

What can all of this have to do with computation? The fact of the matter is that category theory is an intensely computational subject, as all its practitioners well know. Categories themselves are the models of an essentially algebraic theory and nearly all the derived concepts are finitary and algorithmic in nature. One of the main virtues of this book is the unrelenting way in which it proceeds from algorithm to algorithm until all of elementary category theory is laid out in precise *computational* form. This of course cannot be the whole story because there are some deep and important results in category theory that are non-constructive and that cannot therefore be captured by any algorithm. However, for many purposes, the constructive aspects are central to the whole subject.

This is important for several reasons. First of all, one of the most important features of category theory is that it is a *guide to computation.* The conceptual clarity gained from a categorical understanding of some particular circumstance in mathematics enables one to see how a computation of relevant entities can be carried out for special cases. When the special case is itself very complex, as frequently is the case, then it is a tremendous advantage to know exactly what one is trying to do and in principle how to carry out the computation. The idea of mechanizing such computations is very intriguing. The present book, of course, does not enable one to do this, but it can be viewed as an essential precursor of developments that will lead to such mechanization. Categories themselves must be present in the computer as well as many particular examples of them before mechanical computation of categorical entities can be carried out.

Secondly, the fact that category theory is essentially algebraic means that it can be learned by learning these basic constructions. It comes as something of a shock to realize that one aspect of category theory is that it is 'just' a collection of ML-algorithms. However, it is particularly important for computer scientists and students of computer science that there is such a programming language representation of the subject. Because mathematicians have accumulated geometric and algebraic intuitions, many things can be elided in presenting category theory to them. But computer scientists generally lack these intuitions, so these elisions can present a great difficulty for them. Computer code does not permit such elisions and thus presents the basic material in a form that reassures computer scientists and allows them to use their intuitions for and understanding of programs to gain an advantage similar to the mathematicians' advantage from their knowledge of geometry and algebra.

Of course, all of this is beside the point unless there is a reason for computer scientists to need to learn category theory. However, the reasons are easily found by looking into almost any issue of a journal in theoretical computer science. Either the category theory is explicitly there or should be there and is missing only at the expense of devious circumlocutions. It really cannot be avoided in discussing the semantics of programming languages. The most dramatic instance of this arises in the semantics of the polymorphic lambda calculus which underlies ML. It really is an engaging thought that one needs category theory to explain ML, while in turn ML is a vehicle for explaining category theory.

That brings up the last point. There is another audience for this book; namely, category theorists who want to understand theoretical computer science so that they can participate in the exciting interactions that are taking place between these two fields. One very important entry point into the problems of theoretical computer science is just to examine computer programs and to wonder what they mean. There probably is no final answer to this question, but along the way, this book can serve as an invaluable stimulus to further research.

Preface

This is an account of a project we have undertaken in which basic constructions of category theory are expressed as computer programs. The programs are written in a functional programming language, called ML, and have been executed on examples. We have used these programs to develop algorithms for the unification of terms and to implement a categorical semantics.

This book should be helpful to computer scientists wishing to understand the computational significance of theorems in category theory and the constructions carried out in their proofs. Specialists in programming languages should be interested in the use of a functional programming language in this novel domain of application, particularly in the way in which the structure of programs is inherited from that of the mathematics. It should also be of interest to mathematicians familiar with category theory – they may not be aware of the computational significance of the constructions arising in categorical proofs.

In general, we are engaged in a bridge-building exercise between category theory and computer programming. Our efforts are a first attempt at connecting the abstract mathematics with concrete programs, whereas others have applied categorical ideas to the *theory* of computation.

The original motivation for embarking on the exercise of programming categorical constructions was a desire to get a better grip on categorical ideas, making use of a programmer's intuition. The abstractness of category theory makes it difficult for many computer scientists to master it; writing code seemed a good way to bring it down to earth. Someone with a computing background who wishes to learn category theory should have recourse to standard texts, some of which are listed later, but could well find this book a helpful companion text. Mathematicians who have learned a little programming, perhaps in conventional languages like Pascal, may profit from seeing how the functional programming style can embody abstract mathematics and do it in a way not too far from mathematical notation.

In preparing this book, we would especially like to thank John Gray for contributing a foreword. His enthusiasm for this project will be evident. Tony Hoare and the referees gave detailed comments for improving the book. Mike Spivey carefully read the manuscript and gave some useful comments. Anne Rydeheard and John Stell undertook some proofreading for which we are grateful. Ma Qing Ming and Don Sannella pointed out some errors in an early draft. Finally, we are indebted to LaTeX2 and Microsoft Word 3, two document preparation systems used for the book.

Contents

some internal cohesion and *raison d'être*, instead of a bundle of functions which the modularly-minded programmer has forced into uneasy proximity.

Another reason why computer scientists might be interested in category theory is that it is largely constructive. Theorems asserting the existence of objects are proven by explicit construction. This means that we can view category theory as a collection of algorithms. These algorithms have a generality beyond that normally encountered in programming in that they are parameterized over an arbitrary category and so can be specialized to different data structures.

We have expressed categorical algorithms in ML, a functional programming language. Functional languages are closer to mathematical notation than are imperative languages like Basic or Pascal. One writes expressions to denote mathematical entities rather than defining the transitions of an abstract machine. ML also provides types which make a program much more intelligible and prevent some programming mistakes. ML has polymorphic types which allow us to express in programs something of the generality of category theory. However, the type system of ML is not sufficiently sophisticated to prevent the illegal composition of two arrows whose respective source and target do not match. This requires a computation of equality on objects. It is an open question whether a programming language with dependent types or a subtype mechanism can do better.

The relationship of the mathematics to the ML code is as follows: (1) categorical concepts are represented as types in ML, and (2) constructive proofs of theorems in category theory become ML programs. For instance, the theorem that if a category has an initial object and pushouts then it has all finite colimits yields an iterative algorithm for constructing the colimiting cocone of a finite diagram, starting with the initial object and using the pushout at each iteration.

We should make it clear that we have not invented a new programming language or a new specification language. We simply used an existing functional language, ML, to write a novel kind of program of unusual generality. Tatsuya Hagino has indeed invented such a new language for programming and specification, based on adjoints. It turns out to be very like ML, almost identical in its expressive power, but using fewer primitive notions and hence having a more rational structure, a sort of natural mathematical unfolding of the main language concepts as opposed to a computer science evolution of them by trial and error of language designers. We say a little about Hagino's work in Chapter 10.

It has been clear for a long time that the many of the proofs in category theory are constructive and hence could be translated into algorithms; so in a mathematical sense we have just spelled out the obvious. However, from a programming point of view, there is considerable interest in seeing carefully worked out programs to represent the essence of the categorical proofs and to notice that these programs have a certain elegance and pleasing structure. We went to considerable trouble through various formulations to embody as much of the elegance of the categorical approach as possible in our programs. For example, having written a certain function which we needed, we noticed that it formed the object part of a functor and that the arrow would be helpful later on. Seeing these two functions as part of the same functor is a good example of categorical thinking imposing mathematical structure on a program. The Nuprl system [Constable et al. 85] is a proof development system based on constructive logic which automatically extracts a program from a proof. It would be interesting to see how such automatically generated programs compare with our hand-coded ones. Probably in Nuprl one could obtain elegant programs by creating a proper organization of the proof, but the question is as yet unexplored. Unlike the Nuprl formulation, our algorithms only represent part of the information in a proof; they embody the construction; the remaining information in the proof corresponds to the verification showing that the construction produces the required result.

In programming category theory, we are confronted at the outset by the problem: how do we represent a category? Do we use a list of objects and a list of arrows? This would mean we represent only finite categories. Instead we use a functional representation in which the class of objects and that of arrows are types in ML. This allows us to represent infinite categories. Another representation problem arises with the ubiquitous universal properties of category theory. Again we make use of functions, in this case higher order functions. The programs derived from categorical constructions are parameterized on categories. In order to apply the programs to a range of categories, we need systematic ways of constructing categories rather than explicitly encoding them. Goguen suggested we use comma categories for computations on structures such as graphs. We have also made use of functor categories. Another aspect of category theory that is used in the programming is duality. Duality is a fundamental principle in category theory arising from the invariance of the theory under the reversal of arrows. We use it, for instance, to convert programs computing colimits to those computing limits.

In a final chapter we discuss other approaches to computational representation of category theory, notably those of Dyckhoff and Goguen, which are similar in spirit to ours, and that of Hagino, which differs rather radically and interestingly.

We have discovered that applications of our categorical approach to specific computing problems are not easily developed. You have to really understand a task to abstract it in a categorical framework. However, we have two quite interesting applications, a general unification algorithm using coequalizers, which specializes to known unification algorithms, and a categorical implementation of the specification constructing operations in the language Clear.

Since the early 1970s there has been an increasing amount of interest in using category theory to explicate aspects of the theory of computation, in particular, the semantics of programming languages. This is somewhat outside the scope of this book although we try to indicate where categorical concepts are relevant to programming. The range of applications of category theory in computation may be judged from the proceedings of two conferences published as Lecture Notes in Computer Science, nos. 240 (1986) and 283 (1987), Springer-Verlag.

1.1 The contents

In the succeeding chapters, we describe the techniques used in the programming of category theory.

In Chapter 2, we describe the functional programming language Standard ML. We cover all the features of ML that we use later in the book, using illustrative examples. This is meant as a tutorial and a series of exercises is included. Answers to these exercises may be found in an appendix to the book. Those with knowledge of ML can safely omit this chapter. Those with some experience of functional languages may wish to browse through the chapter to acquaint themselves with the syntax of ML. Others ought to read the chapter so as to be able to understand the subsequent programming. In Appendix A there is an index of ML keywords. This may be used as a reference for reading ML programs. Programming in ML is often a rewarding experience and we encourage the reader to get hold of an ML system to practice on.

Chapters 3 to 7 lay out basic category theory. We describe the mathematical concepts and constructions and the corresponding ML programs. We choose illustrative examples which are relevant to programming rather than those drawn from areas of abstract mathematics. In