V. E. Wolfengagen

Combinatory Logic in Programming

Library "JurInfoR"
Founded in 1994

Series: Computer Science and Information Technologies

Project: *Applicative Computational Systems*

Project Leader, Dr.

L. Yu. Ismailova

Published on the occasion of 60th anniversary of Moscow Engineering Physics Institute

Institute for Contemporary Education
"JurInfoR-MSU"

Department of Advanced Computer Studies
and Information Technologies

V. E. Wolfengagen

COMBINATORY LOGIC in PROGRAMMING

Computations with objects through examples and exercises

2nd edition

Moscow
"Center JurInfoR" Ltd.
2003

LBC 32.97
UDC 004
B721

Library of "JurInfoR"
Founded in 1994

Series: Computer Science and Information Technologies

V. E. Wolfengagen

Combinatory logic in programming. Computations with objects through examples and exercises. — 2-nd ed. — Moscow.: Center "JurInfoR", 2003. — X+337 p.

ISBN 5-89158-101-9

The book is intended for computer science students, programmers and professionals who have already got acquainted with the basic courses and background on discrete mathematics. It may be used as a textbook for graduate course on theoretical computer science.

The book introduces a reader to the conceptual framework for thinking about computations with the objects. The several areas of theoretical computer science are covered, including the following: type free and typed λ-calculus and combinatory logic with applications, evaluation of expressions, computations in a category. The topics, covered in the book accumulated much experience in teaching these subjects in graduate computer science courses.

A rich set of examples and exercises, including solutions, has been prepared to stimulate the self studying and to make easier the job of instructor.

ISBN 5-89158-101-9

Center "JurInfoR"
Institute for Contemporary Education "JurInfoR-MSU"
Fax: +7 (095) 956-25-12. E-mail: vew@jmsuice.msk.ru

Contents

Preface of the editors of the series 1

Special Preface 3

The spectrum of problems 5

Preface to the first edition 7

Preface to the second edition 12

Introduction 18

1 Preliminaries 27
- 1.1 The spectrum of ideas 29
- 1.2 Layout of a chapter 30
- 1.3 State-of-the-art in an area 32
- 1.4 Typical task 35
- 1.5 Variants of task 37
- 1.6 A recommended order of solving the tasks 44

2 Derivation of Object 45
- 2.1 Principle of combinatory completeness 46
  - 2.1.1 Combinatory characteristic 46
  - 2.1.2 Systems of concepts 47

2.1.3 Combinatory completeness 47
2.1.4 Elementary combinatory logic 49
2.2 Deriving main combinators: tasks 51
2.3 Historical remark 62

3 Fixed Point **65**
3.1 Theoretical background. 65
3.1.1 Abstraction 66
3.1.2 Multiabstraction 66
3.1.3 Local recursion 67
3.2 Main tasks 68
Exercises 73

4 Extensionality **75**
4.1 Theoretical background 75
4.2 Tasks 77
Exercises 79

5 Numerals **81**
5.1 Numbers and numerals 81
5.2 Combinatory arithmetic 82
5.3 Tasks 87
Exercises 91

6 Typed combinators **93**
6.1 Notion of a type 93
6.1.1 Combinatory terms 96
6.1.2 λ-terms 97
6.2 Tasks 97

7 Basis I, K, S **113**
7.1 Theoretical background 113
7.2 Tasks 114
Exercises 115

8 Basis I, B, C, S **117**
8.1 Theoretical background 117
8.2 A property of being basic 118
8.3 Elementary examples 120
Exercises 121

9 Applications of fixed point combinator Y **123**
9.1 Fixed point theorem 123
9.2 Elements of recursive computations 124
9.3 Using the combinator Y 125
9.4 Evaluation of a function 127
Exercises 131

10 Function $list1$ **133**
10.1 Theoretical background 133
10.2 Tasks 134
10.3 Functor-as-object 137
Exercises 138

11 Isomorphism of c.c.c. and ACS **139**
11.1 Theoretical background 139
11.2 Tasks 141

12 Currying **143**
12.1 Theoretical background 143
12.1.1 Operators and functions 143
12.1.2 Comprehension 144
12.1.3 Connection between operators and functions . 144
12.2 Tasks 145
Exercises 147

13 Karoubi's shell **149**
13.1 Theoretical background 149
13.2 Tasks 150

Exercises 154

14 Products and projections **157**
14.1 Theoretical background 157
14.2 Task 157
14.3 Product and cartesian closed category 161

15 Embedding Lisp into ACS **163**
15.1 Theoretical background 163
15.2 A main task 164
15.3 Concluding remarks 170

16 Supercombinators **171**
16.1 Theoretical background 171
16.1.1 Notion of supercombinator 172
16.1.2 Process of compiling 174
16.1.3 Transformation to supercombinators 175
16.1.4 Eliminating redundant parameters 177
16.1.5 Ordering of the parameters 178
16.1.6 The lambda-lifting with a recursion 182
16.1.7 Execution of the lambda-lifting algorithm . . 185
16.1.8 Other ways of lambda-lifting 188
16.1.9 Full lazyness 190
16.1.10 Maximal free expressions 192
16.1.11 Lambda-lifting with MFE 194
16.1.12 Fully lazy lambda-lifting with *letrec* 196
16.1.13 Compound example 197
16.2 Task 200
16.3 Answers to exercises 202

17 Lazy implementation **211**
17.1 Tasks 211
Exercises 214

18 Permutation of parameters **215**
18.1 Task . . . 215
Exercises . . . 219
Test . . . 219

19 Immediate computations **221**
19.1 Task . . . 221
Exercises . . . 223
Test . . . 224

20 de Bruijn's encoding **225**
20.1 Tasks . . . 225
Exercises . . . 230

21 Abstract machine: CAM **233**
21.1 Theoretical background . . . 233
21.1.1 CAM structure . . . 233
21.1.2 Instructions . . . 235
21.2 Tasks . . . 239
Exercises . . . 240

22 Optimizing CAM-computations **243**
22.1 Task . . . 243
Exercises . . . 252
Test . . . 253

23 Variable objects **255**
23.1 Models . . . 255
23.1.1 Applicative structure . . . 256
23.1.2 Typed models . . . 259
23.1.3 Partial objects . . . 267
23.1.4 Data object models . . . 270
23.2 The main task . . . 274
23.2.1 Elementary types . . . 275

23.2.2 Typed variable objects 276
23.2.3 Computational models 278
23.2.4 Indexed objects 280
23.3 Interpretation of evaluating environment 288

Bibliography **289**

Index **309**

Glossary **313**

Practical work **329**

Dissertations **333**

About the Author **336**

Preface of the editors of the series

Computer science and information technologies have become omnipresent and continue to promise changes that more and more involve, practically speaking, all the spheres of our life. First of all, the new technologies make it easier to get access to diverse information and produce masses of information in electronic form, changing by this both the character of work and its results. Indeed, many products, being of great demand, are produced in the form of a sequence of bits, with an exceptionally high tempo of their changes.

Changes involve both separate professions, and the whole branches of industry and knowledge. In real state of things, the development of information technologies has led to the appearance of virtual reality. Contemporary society is just beginning to get adapted to virtual reality, the capabilities of which are being appropriated quickly enough.

While at its early stages of development programming was a kind of art with a programmer creating a program in order to solve a definite task and providing it with more or less detailed documentation. By now, a powerful industry of programming has been created, complete with accompanying software engineering. At present, in the research in the field of programming or in the sphere of computer sciences, as a rule, support is given to works in which a slight improvement is introduced in the solution of the already well-known problem. At the same time no attention is paid to really important and basic research, being the way to search for new computation concepts, while insufficient attention is given to accumulation of knowledge in the field of programming.

This series of volumes is meant as a continuing row of publications in the field of computer science, information technologies and programming, promoting the accumulation of knowledge in the above-mentioned fields. It is assumed that these publications may be used for presentation of separate courses and can also promote

scientific research. As a rule, the volumes of the series are meant to satisfy the needs of readers of various levels, starting with students that have the aim of getting initial familiarization with the subject itself, up to specialists in different branches of computer sciences.

Thus, the publications of the planned series are meant to reflect the current status of the given field and also to provide the basis for systematic study of different sections of computer sciences, information technologies and programming.

The Editors of The Series

Special Preface

One of the virtues of combinatory logic is that it provides a smart universe of discourse to computer scientists, programmers, applied theoreticians, practitioners. In practice, however, a sophisticated applied theoretician and an applied programmer must have a thorough grasp of what the system of objects does in response to an evaluation request. To make the most transparent use of the evaluation system, one must have insight of such things as what kind of thing is an object or data structure, how to optimize the execution plan of a program for given set of objects, to understand the impact of avoiding the bound variables on a program code, the importance of sub- and super-partitioning of an object set, etc. When used or studied, all these trends are interrelated and involve some self contained local universes of mathematical ideas.

This book covers the commonly used computational ideas related to combinatory logic.

Nevertheless, the lifting from undergraduate to graduate teaching of applicative computations topics initiates unusual didactic challenges which can be explained by the visible distance between the rigorous mathematical apparatus and applied research papers.

This volume addresses this problem by providing a graduate textbook which covers the principle topics for classroom teaching. In this book the gradual progression of topics is supported by examples and exercises. By solving them, the students are introduced to the research issues at the forefront of computer science. The book is written by the author who is a specialist in the covered areas, and who has accumulated a rich experience in teaching graduate courses in those areas.

To use this book a minimal knowledge of theoretical computer science is needed to reach the following aims:

- to provide an introduction to the principle areas of current research in computer science in order to give students the needed knowl-

edge and intuition for more advanced study and research;

- to provide both the practitioners and researchers with a conceptual background of computational ideas that are expected to be used in applied computer science in the coming years.

To achieve the second aim the author has used the general notations of mappings between the classes of objects, in order to bring in more independency from a particular formal system. Most of attention is paid to the main computational ideas and their implementation within a formal framework.

Nobody argues that λ-calculus is a theory of functions. This theory gives a philosophy and reasons to view the idealized entities as functions. It is more important that λ-calculus is related to other theories and the kind of these relations gives important insight into how to build different models of computation.

The spectrum of other theories usually starts with *set theory*. The question is how set theory provides a theory of functions. E.g., Zermelo's theory has a limited view of sets observing separate set A as extremely small with respect to the size of V, the *universe* of all sets. Any particular map $f : A \to B$ from one set A into other set B gives no information concerning maps from V to V, thus the classes of operations on all sets are hardly ever known within set theory. Usual assumption takes set A as an element of V, $A \in V$ in spite of *class* B which is subcollection of V, $B \subseteq V$. The connections between λ-calculus and class theory have been established and studied. This does not mean that λ-calculus depends on the set theory.

If we want a theory of functions be not derived from the set theory we need a pure theory of functions within which functions are observed as particular entities. A *category theory* gives a universe of discourse for selected functions, or better: functional entities. The middle way between pure category theory and a set theory is given by *cartesian closed categories*.

The spectrum of problems

> *"It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group [in Oxford University] as a teaching and research group has been to set up an atmosphere in which this separation cannot happen."*
>
> *Christopher Strachey*
>
> http://vmoc.museophile.com/pioneers/strachey.html

From the moment of their emergence, combinatory logic and lambda-calculus are referred to as "non-classical" by logicians. The point is that combinatory logic appeared in the 20-ies, while lambda-calculus, in the 40-ies, as a branch of mathematics with sufficiently clear-cut purpose of rendering foundation to mathematics. It means that, having constructed the required "applied" mathematical theory, i.e. object theory, reflecting processes and phenomena in real environment, it is possible to make use of a "pure" metatheory as a shell for finding out the capabilities and properties of the object theory.

Combinatory logic and lambda-calculus are such formal systems in which the central elaborated essence is the concept of an object. Within the framework of the first of them, i.e. the combinatory logic, the mechanism of binding variables is absent in its explicit form, while being present in the second one. The presence of this explicit mechanism of binding assumes both the presence of bound variables (however, in this case, free variables are also assumed), as well as mechanisms of replacement of formal parameters, i.e. bound variables, by actual parameters, that is substitution.

The initial purpose of combinatory logic was namely the analysis of this *substitution process*. In the capacity of its essences, it was

planned to use objects as *combinations of constants*. Lambda-calculus was assigned the role of means for specifying the notions of an *algorithm* and *computability*. As a consequence, combinatory logic provides a tool for analysis of the process of substitution. After a short period of time, it turned out that both the systems could be treated as *programming languages*.

Both the systems, in which objects are computed, are calculi or languages of *higher order*, i.e., there are means for describing mappings or operators, which are defined on a set of mappings or operators, while also generating mappings and operators as a result of it. The most important thing is that it is precisely *mapping* that is considered to be an object. This constitutes its basic difference from the whole variety of other systems, for which the notion of *set* and its elements is usually taken as their primary basics.

Currently, both these languages have become the basis for the whole domain of research in the field of computer science and are also widely used in the theory of programming. The development of computational power of computers has led to the automation of a considerable part of theoretical – both logical and mathematical, – knowledge, while combinatory logic together with lambda-calculus are recognized as a basis for considerations in terms of objects.

Without the mastering of their methods, it is impossible to fully develop the basic technique of computations with objects, since the set-theoretic style, still widely used in the object-oriented languages of programming and development, makes it difficult to avoid getting buried in masses of secondary details and, as a result, losing sight of the really important issues of the interaction of objects.

Preface to the first edition of the volume

Whom is the volume addressed to?

This volume has been written to assist those readers who study computer science or are engaged in this sphere of activity, who want to systemize their knowledge and have another look at the spectrum of ideas they have to deal with in their everyday work. The volume is meant to help in reading and studying the original research publications in the field of system and theoretical programming, as well as, if need be, to effect accurate mathematical analysis of newly created programming mechanisms and data models. The detailed explanations and a great number of analyzed examples and problems will assist the reader in getting the message without any considerable effort of making a good choice of the required literature and starting own research work in this interesting and promising field. That is promoted by a considerable amount of independence in the study of separate sections, which is reasonable in view of the specific character of the mathematical discipline itself, i.e. combinatory logic. A more experienced reader in the field of mathematics may be interested in the applied aspects of the theory.

Why should objects be calculated?

Work with computers provided with a shell, which can take over upon itself the control of software objects, lays the basis for the utmost state-of-the-art technique of programming. Presently, hundreds of applied programs, such as Windows, AutoCAD, Designer and many others deal with objects. On the other hand, the instrumental systems of programming, such as Small Talk, C++, Actor and some others require a programmer to exercise systematic speculations in terms of objects and relationship between them, which, in their turn, can be treated as objects. Programming in terms of objects requires the development and maintenance of their own mathematical culture that

provides the whole spectrum of stimulating ideas. In the process of solving a specific task, a programmer becomes a researcher, who is required to create his own language with its own capabilities. Such capabilities are not always intuitively self-evident and may require purely mathematical evaluation of their expressive abilities. Apart from that, very often it is not only necessary to create a certain program code but also to fulfill its optimization without losing the properties of its equivalence to the initial code. For doing careful and professional work, it is required to have its own "mathematical shell", supporting all significant and interesting mathematical applications.

The principal thing is the adequate way of thinking

It is a well-known fact that in the practice of programming, different approaches, developing in different directions, have taken shape. The most evident differences manifest themselves in the difference in comprehension and writing of programs. The major part of programmers is engaged in *procedure-oriented* programming. Apart from that, there exist *rule-oriented* programming, *logical* programming, *parallel* programming, *visual* programming, and programming in terms of dataflows. If need be, this list can be continued, but, evidently, it will be incomplete without also including *object-oriented* programming, displaying a vividly expressed tendency for growth.

Approaches and styles of programming. Approaches and styles of programming are numerous, which reflects the tendency of upgrading and propagating of still newer computer architectures. The emerging new architectures are oriented at new approaches to programming that are still taking shape in research laboratories.

The abundance and great variety of approaches to programming in computer science are reflected in the development and propagation of different approaches to mathematics. And indeed, surprisingly many mathematical theories have been built, each of them being

an absolutely unique language of communication for a comparatively narrow circle of specialists who understand each other perfectly well. At the same time, any attempt of an "uninitiated" person to understand practical usefulness and significance of the new mathematical language meets with obstacles. It turns out that, first of all, it is necessary to change one's own way of thinking so as to be able to have a different look at certain difficulties. Thus, the spread of object-oriented programming also requires the use of different ways of reasoning, which more often than not radically differ from the stereotypes of reasoning adopted in procedure-oriented programming.

Speculations in terms of objects. Similarly, only just few and comparatively young mathematical theories are oriented at speculation in terms of *objects* rather than in terms of *operators*, as goes from the experience in studying mathematical analysis at the majority of universities, including those technically or computer oriented. Unfortunately, a programmer has no opportunity to attend a university course that might lay down the basis of mathematical thinking in terms of objects. At best, only some information about purely mathematical results is given, which are obtained in the framework of *combinatory logic*, *lambda-calculus* or *category theory* and which are far from being easy to apply to practical programming, if you are not versed in theory.

It is possible to assert that combinatory logic has had a considerable influence on the contemporary status of programming. Coming into life as a science about the *nature of substitutions* in mathematical theories, it has then given birth to *functional* programming, programming in terms of *supercombinators*, as well as some other highly productive approaches to programming. In particular, only after one really understands the very spirit of combinatory logic, it is possible to fully comprehend and put into practice the system of programming, having *no predetermined set of instructions*.

Computation theory. The programming paradigms of the 90-ies emerged, to a great extent, from the mathematical way of considerations, adopted in the computation theory. In particular, one of its initial premises was the concept of 'information flow' along a certain 'possible' channel, which gave rise to the appearance of a very fruitful concept of a program, controlled by a data flow. Another example has to do with the idea of using a certain part of combinatory logic by building special instructions-objects within it. Those objects form a system of commands of a *categorical abstract machine*, which can successfully serve as a foundation of quite practical (albeit, object-oriented) systems of programming. More than that, the rules of combinatory logic make it possible to *optimize the compiled program code* by reducing it to a certain normal form. For a specialist in combinatory logic, it was something taken for granted from the very beginning, since that was one of the aims for developing combinatory logic as a mathematical discipline.

The contemporary research in the field of computer science has shown that combinatory logic and its various categorial dialects become an indispensable mathematical language of the programmer, used for exchanging ideas between colleagues. The point is that one of the subject matters of its research is the study of objects and development of different calculuses of objects, satisfying a set of aspects of each applied task. In other words, the solution of any particular task requires the creation of a specific exact language. As is well known by programmers, that is the language of the software interface. In terms of a computer science specialist, it is a specialized dialect of combinatory logic.

Objects and a systemic approach

If a programmer chooses an object-oriented approach to the development, it would most probably be a mistake to adjust the task under solution to a certain already known mathematical model. Probably,

it would be much better to look for some nonstandard solution that adequately matches the specific features related to the very nature of the applied domain. In computer science, the *metatheory* is chosen for this purpose, within the framework of which the research is done and which is "adjusted" to the specific character of the applied domain. One of the means for such an adjustment is to embed the applied theory (a "smaller" theory) into pure metatheory (a "bigger" theory). Besides, from the mathematical viewpoint, within the framework of combinatory logic, it is convenient to build up sub-theories, i.e. special mathematical modules which in their finished form preset the computation mechanisms having a value of their own. Such reasoning can easily find response with a programmer who has to deal with a big software project, when the advantages of considerations in terms of objects and their properties become especially evident. Combinatory logic makes it possible, by using mathematically idealized objects, to preliminarily "play out" all the most complicated and delicate issues of interactions of mechanisms within a big software project.

The City of Moscow
September 1994 *V.E. Wolfengagen*

Preface to the second edition

Applicative Computing Systems. The systems of calculuses of objects, based on combinatory logic and lambda-calculus, are traditionally treated as applicative computing systems, or ACS. The only thing that is being essentially developed within such systems is the notion of an *object*. In combinatory logic the only meta-operator is *application*, or, within a different system of terms, the *using the action of one object to another* . Lambda-calculus has two meta-operators, i.e. *application* and functional *abstraction* that makes it possible to bind one variable within one object.

Objects arising in such systems behave as functional entities, displaying the following features:

- a number of argument places, or the arity of an object, is not fixed in advance but manifests itself gradually in interaction with other objects;

- in designing a composite object, one of the initial objects, i.e. a function, is applied to another object, i.e. an argument, while in other contexts they may exchange their roles, meaning that functions and arguments are equally treated as objects;

- self-applicability of functions is allowed, i.e. an object can be applied to itself.

Computing systems with this most general and least restrictive qualities turn out to be the center of attention of the contemporary computer science community. It is precisely these systems that currently ensure the required meta-theoretical means, making it possible to study the properties of target applied theories, provide the basis for building semantic means of programming languages and ensure means for building data/metadata models within the informational systems.

The second half of the 70-ies – the beginning of the 80-ies witnessed an explosion in the development of applicative computing systems, which has led to the progress in a whole range of trends of research and brought about an abundance of scientific publications. They were written in a special language, their reading and understanding, to say nothing about mastering the very essence of the matter, required a considerable theoretical background. There were a few reasons for that.

In the first place, back in 1980 the theory of applicative computations was still being actively developed, while one of the purposes of the authors of the publications was to arouse interest and involve mathematically talented students into the research in that field.

In the second place, in the course of time, the role and place, assigned to the applicative systems in the curriculum, have changed with time. If earlier, in order to master the most important ideas, it was necessary to get closely familiar with a whole set of mathematical disciplines, now the study of basic elements of applicative computations is included into a standard curriculum, being compulsory for the students in the first and second years. It means that, in writing a volume, preference should be given to the intensional method of presentation of the material, which involves only basic means in order to make the subject matter easily understood.

In the third place, in the meantime, the courses in computer science have changed from purely mathematical and conceptual into a fairly prescriptional ones, giving something similar to a "quick overview" of a mass of ready-made solutions, methods, technologies and recommendations for use.

In the fourth place, the attitude towards computer science has become, to a great extent, purely practical and even that of a consumer. In the majority of cases, it is expected that the mastering of this or that branch of computer science should immediately bring about a certain return. Students even display quite evident unwillingness to do exercises or solve tasks of creative or theoretical character, being

of a fundamental importance. In order to preserve the required level of education in applicative computations, the present volume contains examples and exercises of different character in the field of programming and their solutions are given by using basic means, without assuming any preliminary mathematical background.

The required level of background. In this volume the main set of problems is presented using basic means and, as is expected, will cause no difficulties for students familiar with logic and the principles of proof methods. It is assumed that the reader is also familiar with the basics of programming, having a certain knowledge of data structures. At the same time, the presentation of the material as a whole and partly inside separate sections is built following the ever growing degree of complexity, meaning to please those advanced readers who avoid books intended for "easy reading". The mastering of a number of sections will help the readers to achieve the level of the up-to-date research and stimulate their own scientific studies.

Examples and exercises. The volume contains series of examples and exercises, the major part of which is provided with directions and solutions. As a rule, the solution itself includes a sufficient store of theoretical knowledge that will help not only understand the solution, but also look for other options. It stands to reason that such a book cannot be completely free from mistakes. In case you detect any mistakes or in case you have any suggestions as to how they should be corrected, please, do not hesitate to inform the author about it. If you have new tasks and exercises, the author will be especially glad to learn about it. However, if you decide to send them over, please, provide them with solutions before submitting them to the following e-mail address: `vew@jmsuice.msk.ru`.

Source material and laboratory exercises. This volume is based on a course of lectures, delivered by the author at the Moscow En-

gineering and Physical Institute (MEPhI). The courses of lectures, delivered at the MEPhI on the basis of the present volume, are provided with Laboratory Exercises. The Laboratory Exercises product is compatible with the IBM PC and is distributed on machine-readable media.

Acknowledgements. The author highly appreciates the contribution made by Dr. R.V. Khrapko, and N.P. Masliy, A.S. Afanasyev, T.V. Syrova, who displayed a high level of interest in and profound professional understanding of the problems involved and their great erudition. In developing software, they strived to use applicative or purely object technologies and solutions, as well as the technique of dynamic formation of objects.

The conditions and possibilities for application of the object approach to software development were provided by V.V. Ivanchenko, who did his best to organize the scientific research process.

I.V. Papaskiry's assistance and well-meaning attention made it possible to establish a permanently acting workshop, as well as a unique atmosphere of scientific debates during which the main range of ideas presented in the volume were discussed.

Prof. L.T. Kuzin paid considerable attention to the methods and means for work with abstract objects, promoting research in that direction, its development, application and extension to the solution of various tasks in the field of cybernetic simulation. A whole number of results obtained was discussed at scientific workshops of the "Applied Problems of Cybernetics" Section, chaired by him, while their application was presented within the framework of a cycle of disciplines, headed by him, for the students of the Cybernetics faculty of MEPhI as part of training engineers-mathematicians, specializing in applied mathematics.

Sincere gratitude is due to K.A. Sagoyan for the preparation and check-up of solutions of all the examples, given in Section 6, that were also used in the Preprint publications. His thoroughness, hard

work and resourcefulness evoke genuine admiration. Various variants of their solutions were tested by him in the course of conduction of laboratory exercises with the students at the MEPhI Cybernetics Department.

Enormous patience and readiness to show a whole range of applications for object calculus were shown by I.V. Chepurnova, who took upon herself the responsibility for the publication of the Preprint version of this volume. It would be impossible to overestimate her work of many years in organizing and conducting workshops for the MEPhI full-time students and for the post-graduate students of the MEPhI enhancement courses. Her special courses were based on the sections contained in the present volume, which were updated and extended at her suggestions.

I.A. Goryunova took upon herself a difficult assignment in picking up tasks for Section 16. Enthusiasm, displayed by her, and profound understanding of the mechanism of dynamic formation of objects made it possible to select exercises, provided with solutions.

A.G. Panteleev offered a number of valuable comments on Section 15.

Wu-Hoang Ham indicated other possible variants of solutions for some examples.

Profs. B.V. Biryukov and A.S. Kuzichev, who chaired the scientific workshop in the history and methodology of natural sciences at the Moscow State University (MSU), provided valuable recommendations in selecting bibliography. Prof. V.A. Smirnov indicated additional possible areas of applications of intensional methods in combination with object calculuses.

Prof. S.Kh. Aytyan provided remarks for parts of the volume material in the course of its writing. Some aspects of object applications were discussed with Prof. G.G. Belonogov, Prof. G.Ya. Voloshin, Prof. S.N. Seletkov, and Prof. E.N. Syromolotov.

Prof. P.-L. Curien of Université Paris VII, LITP, one of the authors of the categorical abstract machine concept, was very kind to

provide me with a number of his publications, which made it possible to speed up the work on the first publication of the volume.

Dr. V.Ya. Yatsuk, shared his procedures and experience he gained while presenting object calculi in the courses, delivered by him.
I.V. Mazhirin indicated some possibilities in developing versions of abstract machines. Dr. G.S. Lebedev, initiated the teaching of deductive extensions of object calculi.

E.M. Galstyan rendered technical support in the preparation of the preliminary version of the volume text.

The preparation of the present volume for publication was stretched in time and became feasible thanks to collaboration with the following numerous interested specialists, who took part in the work of scientific workshops held on the problems of computer science and information technologies at the Moscow Engineering and Physical Institute, as well as organized practical lessons for students of different years of education, and supervised their course and diploma projects: Yu.G. Gorbanev, V.I. Vasiliyev, O.V. Voskresenskaya, M.Yu. Chuprikov, O.V. Barinov, I.A. Alexandrova, G.K. Sokolov, S.V. Kosikov, I.A. Volkov, S.K. Saribekyan, K.A. Sagoyan,A.I. Mihkaylov, T.V. Volshanik, Z.I. Shargatova, I.A. Goryunova, A.V. Gavrilov, L.V. Goltseva, E.V. Burlyaeva, V.A. Donchenko, I.V. Chepurnova, A.V. Myasnikov, K.E. Aksyenov, S.I. Dneprovsky, S.V. Bryzgalov, S.V. Zykov, Ye.S. Pivovarova, A.Yu. Rukodanov, S.A. Kastanov, L.A. Dmitrieva, Yu.Yu. Parfenov, R.K. Barabash, A.I. Odrova, A.L. Brin, R.V. Snitsar, A.L. Zabrodin, A.M. Grigoryev, G.G. Pogodaev, B.B. Gorelov, M.L. Faybisovich, K.V. Pankin, N.V. Pishchimova, A.I. Vaizer.

Finally, it has become possible to issue the present volume "Combinatory Logic in Programming" only thanks to the enormous time and energy, rendered by the employees of the Institute for Contemporary Education "JurInfoR-MSU".

The City of Moscow
January 2003 *V.E. Wolfengagen*

Introduction

Objects and *modes of combining the objects* represent all that is essential in combinatory logic. Combination of objects with some other objects is implemented by initial identification of objects-constants, called *combinators*. There are just a few such initial combinators, however, by using them one can build such well-known formal systems as the logic of statements, the logic of predicates, arithmetic systems[1] and a whole number of others. In the past ten years, combinatory logic has become one of the basic metamathematical techniques of computer science, having shown its capabilities in the sphere of programming. There has appeared a whole family of functional programming languages. Miranda, ML, KRC, being sufficiently well known to programmers, give us an idea of their possible application. However, the object-oriented approach to programming and designing of applied systems as a whole, which has already displayed its overall potential, raises fundamental questions, referring to the method of operation itself in terms of objects. For this purpose, a preselected *universe of discourse* is required, being a sort of theoretical shell that guarantees mathematical fruitfulness of the performed investigation. This is felt especially acutely in the process of implementation of large software projects when the choice of a systematic method of representation and operation of objects acquires decisive importance.

Discussion of the volume structure

The structure of this volume and the layout of separate sections are implemented in such a way as to enable the reader to fully concentrate the attention on the matter of computations with objects being of fundamental importance.

[1] *More rigorously: the systems of numerals*

• Synthesis of a new object

One of the most important tasks, solved within the framework of combinatory logic, is formulated as a task of synthesis of an object with preset properties out of the available objects by using the already known modes of combining. At the initial stage, the availability of only three objects-combinators is assumed: I, K, S, as well as that of their properties, preset by characteristic equations. To preserve the intuitive clarity, one can assume that there exists a system of programming with these three instructions and, using exclusively these instructions, one will have to build up a system of programming, being rich enough in its expressive capabilities. The resultant system will contain solely objects-combinators.

• Characteristics of a fixed point combinator

The transparency of combinatory logic makes it rather easy to study. After the first steps, it may seem that within it we always have to deal with simple and finite in their nature objects. However, that impression is deceptive, and, by using combinators, it is possible to represent *processes*, including loop computations, which are problems with a recursion stack, being well known in programming.

• The use of the extensionality principle

The characteristic feature of combinatory logic is that it is possible to build and apply functions within it, having *a number of arguments that is not fixed a priori*. It means that one should be cautious enough while answering the question about how many argument positions the applied function-object has in reality. Indeed, careful use of sufficiently simple principles of extensionality (expansibility) makes it possible to overcome that indefiniteness.

• Numerals and their properties

Note that, *from the very beginning*, there are no . . . numerals among primary objects in combinatory logic. The point is that the concept of a numeral can be developed independently, by using known combinators. In such a case, numerals present themselves in a somewhat unusual aspect, being objects, which display their arity, depending on the used system of postulates. In the same way, arithmetical operations are successfully developed in the form of combinators. In other words, arithmetical entities are built into combinatory logic. This situation is well known in object-oriented programming, i.e. an application (arithmetical objects with their rules) is built into a software environment (combinatory logic).

• Study of properties of combinators with types

The concept of a class is one of the most important in object-oriented speculations. In this case, a class is understood as a sample for creating specimens of concrete objects. Moreover, classes themselves can be treated as objects. Similarly, combinators are classified or typified. The high order of functional spaces turns out to be important for combinators. Nevertheless, the intuitive clarity of the work with combinators as objects does not get lost.

• Expansion of terms in the I, K, S basis

Let us concentrate on the simplest system of programming with only three instructions: I, K, S. It is possible to synthesize a new object by a purely mechanical use of the algorithm of expansion in the basis, being quite similar to the process of compilation.

• Expansion of terms in the I, B, C, S basis

As it turns out, the I, K, S basis is not the only one, and the set of combiners I, B, C, S also displays the property of a basis. Compila-

tion (expansion) of an object in that basis also resolves the problem of synthesis of an object with preset properties. Evidently, it is possible to use the freedom of selecting a basis, depending on some criteria.

• Expression of function definition using the fixed point operator Y

The case of recursive definitions of objects is considered. By using the fixed point theorem, being fundamental for functional programming, recursive definitions can be successfully reduced to the conventional equational form.

• Study of properties of the $list1$ function

Capabilities for building a function-object in a parameterized form are shown. By assigning particular values to arguments — and functions can also be these particular values, — it is possible to get a whole family of definitions for particular functions.

• Establishment of isomorphism for the Cartesian closed category and applicative computation system

Now, we begin to move into the depths of mathematical abstractions and start coordinating the operational way of thinking with combinatory approach. Note that within combinatory logic, use is made of only one operator, i.e. the *application operator*, or the operator of application of one object to another. The system of computations, arising as a result of it, is called an *applicative computation system*. It is linked up with the conventional computation system, being represented by a special object, i.e. the *Cartesian closed category*.

• Building of representation, currying the n-ary function

Within combinatory logic, n-ary functions-operators, used in operational programming, have images in the form of objects, which inherit all their substantial properties.

• Deriving of basic properties of the Karoubi's Shell

A special category, called the Karoubi's shell, makes it possible to laconically express a whole body of knowledge, related to operators, in terms of combinatory logic. In this case, the types are also encoded by objects. As a result, we implement the immersion or embedding of the typed application into the type free software environment.

• Cartesian product and projections: embedding into ACS

The final completion of the started process of embedding is achieved by the introduction of ordered populations of objects into consideration. It has turned out that applicative computations also allow their representation.

• Lisp Representation by means of lambda-calculus or combinatory logic

A non-trivial application is built into the applicative computation system: a considerable and, actually, complete fragment of the well-known Lisp system of programming.

• Implementation of computation of expressions values with the help of supercombinators

It is discussed how the object-oriented systems, built into combinatory logic, work. Thus, it is possible to directly satisfy the need in denotational computation of instructions of programming languages,

when objects are used to express the functional meaning of a program. It is significant that computation begins with a certain a priori known set of instructions. In the process of the program value computation, new instructions dynamically arise, being earlier unknown but indispensable in the process, which are additionally fixed in the system of programming.

• Fully lazy implementation of supercombinators

When the dynamic formation of objects is performed "on-fly", the efficiency of the resulting code may be lost because of the need to repeatedly calculate the value of one and the same object. The application of the mechanism of lazy evaluations makes it possible to avoid it: once the value of an object has been calculated, this already calculated value will always be used further on.

• Optimization of computation process by permutation of parameters

The use of combinators makes it possible to build up an optimized program code, while, in the course of synthesis of the resulting object, analyzing the order of the possible substitution of formal parameters with actual ones.

• Implementation of direct computations of expressions of programming languages

The technique of evaluating the expressions is reconsidered in the view of systematic build-up of both syntactic and semantic equations, implementing the selected paradigm of object-oriented computations.

• Evaluation of the de Bruijn's code

A technique for re-designation of bound variables (formal parameters) is introduced into consideration, which makes it possible to avoid binding collisions in replacing formal parameters by actual ones. This technique of re-designation is called *de Bruijn's encoding* and allows, in fact, the lambda-calculus apparatus to be used on the same legal basis as for the combinatory logic apparatus.

• Implementation of machine instructions of the categorical abstract machine (CAM)

A special version of the theory of computation, named *categorical abstract machine*, is being built. For this purpose, a special fragment of combinatory logic, i.e. categorical combinatory logic, is introduced into consideration. It is represented by a set of combinators, each of which being of a self-sustained significance as an instruction of the system of programming. Thus, one more useful application, i.e. the system of programming, based on the Cartesian closed category, is being built into combinatory logic. It allows the connection between the operative and applicative styles of programming to be reconsidered once again but at a new level.

• Capabilities of optimization in computation on the CAM

The use of Cartesian closed category opens up additional capabilities for the optimization of the resulting program code. In addition to the properties of combinatory logic per se, used as a shell, it is allowed to use special categorical equations, borrowed from the Cartesian closed category, as from an application.

• Variable objects

The concluding part of considering the calculi of objects deals with generic questions of mathematical representation of objects. It is

shown that the use of the concept of the functor-as-an-object allows the basic laws of the object-oriented computations to be reviewed in a compact and laconic form. In particular, the emphasis is laid on the systems of changing (variable) notions-concepts, which are conventional objects of combinatory logic but display properties, being useful for programming. For example, with the help of variable concepts, the theory of computations is built without any complications, as well as the semantics of the programming systems and the models of data objects. The data-as-objects display new degrees of freedom in computer considerations.

Short recommendations about the order of studying the volume

It is possible to outline the order for reading of the given volume. Stand-alone reading of Sections 2–4, 6 requires only minor efforts on the part of a reader. This material gives you a feeling about flexibility and impressibilities of the language of combinatory logic.

To this preliminary part, Sections 5, 9, 7–8 can be added, which deal with numerals, recursion, and expansions in basis.

Then, one may read Sections 10, 15–18, introducing a body of concepts for programming by means of combinators with the dynamic system of instructions.

Other sections can be added to one's taste. In particular, a more detailed acquaintance with the categorial abstract machine in Sections 19–22 will require one to refer to the literature cited in the bibliography. Sections 11–14 are aimed at those readers, who want to independently begin reading original research papers in the field of computer science. Having mastered the basic ideas of the theory of computation, one can start reading papers, devoted to this topic. On the other hand, Section 23 may of interest to those readers, who would like to go deeper into the very essence of creating "built-in applications", requiring modification of the software environment. In

such a case, the researcher encounters variable notions or, in other terms, variable concepts.

Chapter 1

Preliminaries

Nowadays the theoretical studies in an area of computer science are grounded at the stable main mathematical means. As it happened, this is just the several interrelated branches, giving rise both to logic and computer science. Among them are λ-*calculus*, *combinators*, *type systems*, *category theory* and *programming languages*.

λ-calculus. In this branch those modes of constructing the pure calculus of functional abstraction and application of functions are studied, that obtain the application areas in metamathematics, logic and computer science.

Combinatory logic. In developing this branch it has been shown, that bound variables can be excluded without lost of expressive power of the formal system. The most important applications of combinatory logic have been found in constructing the foundations of mathematics, and in constructing methods and and tools for implementing the programming languages.

Type systems. Currently, λ-calculus gives the main means for studying the type systems, and its results gave an important insights

both for the foundations of mathematics and for the practice of constructing and applying of programming languages.

Category theory. This branch, from a mathematical point of view, deals just with two entities – *objects* and *mappings*, – but the last, in turn, can be considered as objects. Applying the methods of category theory seems especially important in case of research, development and application of the object programming systems.

Programming languages. It is rather difficult to consider these three main branches: λ-calculus, combinatory logic and type systems, – singularly, separated each from others. They are interrelated, and current point of view is in their mutual studies. Moreover, namely these three branches give a sound ground to construe the programming languages.

Currently, there are more and more reasons to add to this list, on equal rights, one more branch – *category theory*, which involves the only notion, namely: representation of an abstract *object* that covers the mappings as well.

At a first look, a spectrum of ideas, covered in these branches, is not homogeneous, resulting in growth of seemingly disparate formalisms and mathematical means. Actually, it happens even more: every current research, as a rule, includes the constructing of its own mathematical means. Rather often, the main mathematical ideas, used by the author, are hidden by the complicated evaluations.

Studying of the branches, mentioned above, helps to discover the intrinsic unity of the disparate researches, and their regular application supplies a researcher or developer by both the conceptually transparent and flexible and powerful theoretical means.

1.1 The spectrum of ideas

Currently, after more careful consideration, it appears that the fulfilled in an area of computer science researches use some or other ideas either of *category theory*, or of *combinatory logic*, or of *calculi of λ-conversions*, or of all these three disciplines altogether.

Importance of getting acquainted the knowledge of these branches of mathematics is so evident, that, opening practically occasional proceedings of any conference in computer science, you can find not only pure nominal usage of λ-notations, but actual constructing of own mathematical language, which is based on using the applications and abstractions. Nevertheless, the attempts of self-studying the foundations of λ-calculus or combinatory logic meet the difficulties from the very early steps: a known effort is needed "to fit a way of thinking" *from* operational point of view of mathematical notations *to* applicative one, when there is no generic separation of mathematical entities into 'functions' and 'arguments', but there are the only 'objects'. It is interesting to observe, that the same object, depending on a context, can be used in different *roles*, sometimes playing the role of an argument and sometimes — the role of a function.

From the syntax point of view, the objects are indicated either by the notations with parentheses, or by using an agreement of omitting the non-significant parentheses, when they, by need, can be restored without any ambiguity. Another significant feature is to establish a property of being *basic* for some pre-specified objects. Any time it is evident, that newly introduced object can be rather suitably represented by *combining* of generic objects, which, under such a circumstance, can be called the *combinators*. It is difficult to overestimate this ability to disassemble the arbitrary introduced object into basis: in place of studying of properties of complicated applied theory with a great amount of distinct objects, it could be possible to study the properties just of several objects, without any loss of generality of the

obtained result.

For a specialist in computer science this is rather desirable property. It is evident, that the basic combinators can be assumed as a *system of commands* of some abstract computing system, and the rest of the objects can be expressed, i.e. "programmed", by using namely this set of commands. In spite of seeming evidence, this ability of means of the combinatory logic is not yet properly applied in practice of computer studies.

1.2 Layout of a chapter

A discussion of philosophical, pure mathematical or technical aspects of applicative computations can lead far distant to the side of foundations of mathematics. Nevertheless, there is rather acceptable way. For a beginning, it can be reasonable to restrict yourself by solving some — however, for a first look, abstract, — problems, and after that make a conclusion, if is it needed to move further, in depth of ideas of applicative computations.

This chapter is to be assumed as some kind of menu, in which the main questions are indicated, and interaction of which, at first, could be seen immediately, but after that, in detailed study, getting clear its deeper essence.

The current chapter contains the sets of variants of the tasks, which are recommended to use in self studying.

In case of organizing the studying in the classes of λ-calculus and combinatory logic, it can be recommended to select out the special units of tasks by variants, allowing to make not "so large", but rather acceptable steps in learning of new mathematical, or, better, computational ideas. These variants of the tasks are arranged in the Table 1.1. The tasks for self studying are composed by such a way, that covers most of the chapters in this volume.

Table 1.1: The variants of tasks

Variant N°	Recommended unit of tasks					
1	1.1	2.6	3.3	4.7	5.1	6-1°
2	1.2	2.5	3.1	4.6	5.2	6-2°
3	1.3	2.4	3.2	4.5	5.3	6-3°
4	1.4	2.3	3.3	4.4	5.4	6-4°
5	1.5	2.2	3.1	4.3	5.1	6-5°
6	1.6	2.1	3.2	4.2	5.2	6-6°
7	1.7	2.5	3.3	4.1	5.3	6-7°
8	1.8	2.4	3.1	4.7	5.4	6-8°
9	1.9	2.3	3.2	4.6	5.1	6-9°
10	1.10	2.2	3.3	4.5	5.2	6-1°
11	1.11	2.1	3.1	4.4	5.3	6-2°
12	1.12	2.5	3.2	4.3	5.4	6-3°
13	1.1	2.4	3.3	4.2	5.1	6-4°
14	1.2	2.3	3.1	4.1	5.2	6-5°
15	1.3	2.2	3.2	4.7	5.3	6-6°
16	1.4	2.1	3.3	4.6	5.4	6-7°
17	1.5	2.5	3.1	4.5	5.1	6-8°
18	1.6	2.4	3.2	4.4	5.2	6-9°
19	1.7	2.3	3.3	4.3	5.3	6-1°
20	1.8	2.2	3.1	4.2	5.4	6-2°
21	1.9	2.1	3.2	4.1	5.1	6-3°
22	1.10	2.5	3.3	4.7	5.2	6-4°
23	1.11	2.4	3.1	4.6	5.3	6-5°
24	1.12	2.3	3.2	4.5	5.4	6-6°
25	1.1	2.2	3.3	4.4	5.1	6-7°
26	1.2	2.1	3.1	4.3	5.2	6-8°
27	1.3	2.5	3.2	4.2	5.3	6-9°
28	1.4	2.4	3.3	4.1	5.4	6-1°
29	1.5	2.3	3.1	4.7	5.1	6-2°
30	1.6	2.2	3.2	4.6	5.2	6-3°
31	1.7	2.1	3.3	4.5	5.3	6-4°

1.3 State-of-the-art in an area

For advanced and deeper learning of the chapters of this volume, for those, who will not be satisfied with rather elementary level, some papers and books can be recommended. A part of them is quite accessible as a material for the first reading, while others have a character of originally performed research. In the last case, a possibility to be in touch with that, what is done like a mathematical theory in computer science, is left. For this purpose best of all is to use the originally published works in its author's edition, to read and understand which it can be quite possible to get ready by solving the recommended tasks.

Pioneer for the computer science research has been conducted by J. McCarthy (J. McCarthy, [101]). He constructed a language for list processing Lisp, which is quite object-oriented and, in addition, a functional programming language. During the years, Lisp was one of the most popular programming systems in an area of artificial intelligence, getting acquainted the name of 'knowledge assembler'. Note, that Lisp is a computer implementation of λ-calculus, giving to a programmer practically all the means of this powerful mathematical theory. An effective implementation of the Lisp-interpreter, done by A.G. Panteleev (A.G. Panteleev, [32]; M.A. Bulkin, Yu.R. Gabovich, A.G. Panteleev, [4]), have revealed a lot of details, concerning the implementation of object and functional languages, and the ways and methods of their efficient usage. The methods of implementing the mechanisms of binding the variables, the data structures, the procedures, the functions and recursion, established in works on this project, have been stimulated the entire direction of research activity in a programming.

In the books (L.T. Kuzin, [25], [26]) it can be found a short and acceptable for an engineer introduction to the notations in λ-calculus and combinatory logic. A complete covering of all the spectrum of mathematical ideas is in the book (H. Barendregt, [2]), however, to

learn it, a preliminary mathematical background is needed. Classic, very detailed, transparent and circumstantial discussion of logical means of applicative computational systems, given in the book (H. Curry, [6]), will help to form a vision, possibly, of all the object-oriented approach, which was developed in the mathematical branches of computer science. It ought to have in mind, that due to the efforts of H. Curry, combinatory logic has been formulated not only as a general mathematical discipline, but as the necessary foundation for computer studies.

Different assistance in methods of solving the problems, giving insight for applications of applicative computational systems (ACS), can be found, e.g., in (A.A. Stogniy, V.E. Wolfengagen, V.A. Kushnirov, V.I. Sarkisyan, V.V. Araksyan, and A.V. Shitikov, [39]), (V.E. Wolfengagen,[41]), (V.E. Wolfengagen and V.Ya. Yatsuk, [42]), (V.E. Wolfengagen and K.A. Sagoyan, [43]), (V.E. Wolfengagen, K.E. Aksenov, L.Yu. Ismailova, and T.V. Volshanik, [44]), (A.A. Ilyukhin, L.Yu. Ismailova, and Z.I. Shargatova, [12]), (K.E. Aksenov, O.T. Balovnev, V.E. Wolfengagen, O.V. Voskresenskaya, A.V. Gannochka, and M.Yu. Chuprikov, [1]).

Research work (V.E. Wolfengagen and V.Ya. Yatsuk, [45]) reflects the early state-of-the-art and contains the detailed constructions for several variants of applied applicative systems, valuable for solving the applied problems aimed to development of information systems.

A distinctive place is occupied by the work (D. Scott, [115]). Its soundness and a range of influence for several generations of theoreticians in computer science hardly ever might be overestimated. Evidently, far not all the ideas from this paper have found its immediate application. In particular, an idea to construe the systems of combinators for special classes of computations stimulates the deep and vital studies.

The papers (G. Cousineau, P.-L. Curien, and M. Mauny [75]), (P.-L. Curien, [76], [77]) contain the constructing of a special kind of combinatory logic called *categorical combinatory logic*. It gave

a foundation for constructing the abstract machine, which is both an enhanced notion of computation and self standing programming system, aimed to pursue the research in an area of programming.

Several supplementary works, given in bibliography, will assist to start with your own research in various applied areas, using the possibilities of applicative computational systems. In (R.B. Banerji, [62]), (V. Wolfengagen, [128]) the applications in artificial intelligence could be found.

A particular attention could be paid to the research papers by D. Scott (D. Scott, [110], [111], [112], [113], [114], [115], [116], [117], [118]). These (and many other) of his works gave not only the mathematical foundations, but all the contemporary system of thinking in computer science. Computation theory, semantic of programming languages, object based computations – this is far not complete list of research directions, inspired by this mathematician.

An introduction to mathematical problems of combinatory logic and λ-calculus can be found in a series of papers (A.S. Kuzichev, [16],[17], [18], [19],[20], [21], [22], [23],[24]). In theses papers a research experience of expressive power of the systems with operators of application and (functional) abstraction is covered. The elementary basics of combinatory logic are covered in (J. Hindley, H. Lercher, and J. Seldin, [93]).

The research works by N. Belnap could be useful in constructing a theory of computattional information systems (N. Belnap, [3], [64], [65]). Similar topics are covered in (M. Coppo, M. Dezani, G. Longo, [74]).

The programming systems in terms of objects, based on applicative computations, with various details are covered in [5], [54], [59], [60], [67], [84], [85], [91], [95], [102], [105], [124]. As an introduction into the spectrum of ideas, how to develop a semantic of programming languages, the book (V.E. Wolfengagen, [51]) can be used.

As a source work – for *supercombinator* programming, – a particular attention could be paid to the research (R.J.M. Hughes, [95])

and (S.L. Peyton Jones, [105]).

The advance in formal methods is in (R. Amadio and P.-L. Curien, [56]), (J. Lambek and P. Scott, [99]), and, in application to event-driven computations, is in (V. Wolfengagen, [130]).

The rest of the citations, given in the bibliography, could assist to get intuitive vision in the adjacent to mentioned here questions and start with your own research in applicative computing.

1.4 Typical task

General advice to solving a typical task.

Task formulation. Derive via K and S the object with combinatory characteristic:

$$Ia = a, \qquad (I)$$

using postulates α, β, μ, ν, σ, τ, ξ of λ-conversion.

Solution.

$I-1$. List the postulates, which determine a relation of conversion '=' :

$$(\alpha)\ \ \lambda x.a = \lambda z.[z/x]a;\ \ (\beta)\ \ (\lambda x.a)b = [b/x]a;$$

$$(\nu)\ \ \frac{a=b}{ac=bc};\qquad (\mu)\ \ \frac{a=b}{ca=cb};$$

$$(\xi)\ \ \frac{a=b}{\lambda x.a=\lambda x.b};\quad (\tau)\ \ \frac{a=b;\ b=c}{a=c};\quad (\sigma)\ \ \frac{a=b}{b=a}.$$

$I-2$. Define the combinatory characteristics of the objects K and S:

$$\begin{aligned} v\,(Kxy) &= v\,x, && (K)\\ v\,(Sxyz) &= v\,(xz(yz)), && (S)\end{aligned}$$

which are expressible in λ-calculus by the equalities $K = \lambda xy.x$ and $S = \lambda xyz.xz(yz)$.

$I-3$. Using schemes (K) and (S), get confidence in that:

$$\begin{array}{rcll} a & = & Ka(Ka) & (K) \\ & = & SKKa. & (S) \end{array}$$

Checking. Make sure, that actually $I = SKK$. Let $v = empty$ (empty object).

$I-1$. $SKKa = Ka(Ka)$, because in scheme (S) can be set $x = K, y = K, z = a$. Then it is evident, that by postulate (α):

$$Sxyz = SKKa,\ xz(yz) = Ka(Ka),\ SKKa = Ka(Ka).$$

$I-2$. Using in a similar way the scheme (K), conclude, that $Ka(Ka) = a$.

$I-3$. By rule of transitivity (τ), if the equalities $SKKa = Ka(Ka)$ and $Ka(Ka) = a$ are valid, then $SKKa = a$.

Answer. Object I with given combinatory characteristic $Ia = a$ is SKK, i.e. $I = SKK$.

1.5 Variants of task

‡ **Work 1.** Derive via K and S the objects with given combinatory characteristics:

1) $Babc = a(bc)$,
2) $Cabc = acb$,
3) $Wab = abb$,
4) $\Psi abcd = a(bc)(bd)$,
5) $C^{[2]}abcd = acdb$,
6) $C_{[2]}abcd = adbc$,
7) $B^2abcd = a(bcd)$,
8) $Ya = a(Ya)$ (prove, that $Y = WS(BWB)$),
9) $C^{[3]}abcde = acdeb$,
10) $C_{[3]}abcde = aebcd$,
11) $B^3abcde = a(bcde)$,
12) $\Phi abcd = a(bd)(cd)$.

‡ **Work 2.** Determine a combinatory characteristic of the following objects:

1) $(\lambda x.(P(xx)a))(\lambda x.(P(xx)a)) = Y$,
2) $Y = S(BWB)(BWB)$,
   where $B = \lambda xyz.x(yz)$, $S = \lambda xyz.xz(yz)$, $W = \lambda xy.xyy$,
3) $Y = WS(BWB)$,
   where $Wab = abb$, $Sabc = ac(bc)$, $Babc = a(bc)$,
4) $Y_0 = \lambda f.X(X)$, where $X = \lambda x.f(x(x))$,
5) $Y_1 = Y_0(\lambda y.\lambda f.f(y(f)))$,
   where $Y_0 = \lambda f.X(X)$, $X = \lambda x.f(x(x))$.
   (Hint: prove, that $Y_i a = a(Y_i a)$.)

‡ **Work 3.** Prove, that:

$$\begin{array}{lrcl}
1) & X & = & \lambda x.Xx, x \notin X, \\
2) & Y_0 & = & \lambda f.f(Y_0(f)), \text{ where } Y_0 = \lambda f.X(X), \\
 & & & X = \lambda x.f(x(x)), \\
3) & Y_1 & = & \lambda f.f(Y_1(f)), \text{ where } Y_1 = Y_0(\lambda y.\lambda f.f(y(f))), \\
 & & & Y_0 = \lambda f.X(X),\ X = \lambda x.f(x(x)).
\end{array}$$

‡ **Work 4.** Determine a combinatory characteristic of the following objects[1] (prove it!):

$$\begin{array}{lrcl}
1) & \Xi & = & C(BCF)I, \\
2) & F & = & B(CB^2B)\Xi, \\
3) & P & = & \Psi\Xi K, \\
4) & \& & = & B^2(C\Xi I)(C(BB^2P)P), \\
5) & \vee & = & B^2(C\Xi I)(C(B^2B(B(\Phi\&))P)P), \\
6) & \neg & = & CP(\Pi I), \\
7) & \exists^* & = & B(W(B^2(\Phi P)C\Xi))K, \text{ where } \exists[a] = \exists^*[a],\ \exists = \exists[I].
\end{array}$$

Hints.

1) $Cabc = acb,\ Ia = a,\ Babc = a(bc).$
2) $Babc = a(bc),\ B^2abcd = a(bcd),\ \Xi ab = FabI.$
3) $\Psi abcd = a(bc)(bd),\ Kab = a.$
4) $Babc = a(bc),\ B^2abcd = a(bcd),\ Ia = a,\ Cabc = acb.$
5) $Babc = a(bc),\ B^2abcd = a(bcd),\ Ia = a,$
   $\Phi abcd = a(bd)(cd),\ \&ab = \Xi(B^2(Pa)Pb)I.$
6) $Cabc = acb,\ Ia = a,\ \Pi = \Xi W\Xi,\ Wab = abb.$
7) Prove, that $\exists[b]a = P(\Xi a(Kb))b.$

[1] Notations: P – implication, Ξ – formal implication, F – operator of functionality, & – conjunction, $\vee$ – disjunction, Π – universal quantifier, $\neg$ – negation, $\exists$ – existential quantifier. The objects Ξ, F, P, &, $\vee$ are two placed, and the objects $\neg$, Π, $\exists$ are one placed.

Use the following equalities:

$$\begin{array}{lllllll} Babc & = & a(bc), & B^2abcd & = & a(bcd), & Wab & = & abb, \\ Cabc & = & acb, & \Phi abcd & = & a(bd)(cd). & & & \end{array}$$

‡ **Work 5.** Verify the validity of the the following combinatory characteristics:

1) $S(KS)Kabc = a(bc)$,
2) $S(BBS)(KK)abc = acb$,
3) $B(BW(BC))(BB(BB))abcd = a(bc)(bd)$,
4) $B(BS)Babcd = a(bd)(cd)$.

Hint. $Kab = a$, $Sabc = ac(bc)$, $Babc = a(bc)$, $Cabc = acb$, $Wab = abb$.

‡ **Work 6.** Perform the following target studies:

6-1° study a disassembling of terms into the basis I, K, S;

6-2° study a disassembling of terms into the basis I, B, C, S;

6-3° express definitions of the functions, using a fixed point combinator Y;

6-4° study the properties of the function:

$$\begin{array}{l} list1\ a\ g\ f\ x = if\ null\ x \\ \qquad\qquad\qquad\quad then\ a \\ \qquad\qquad\qquad\quad else\ g(f(car\ x))\ (list1\ a\ g\ f(cdr\ x)); \end{array}$$

6-5° establish the isomorphism between a cartesian closed category (c.c.c.) and applicative computational system (ACS);

6-6° derive the mapping, which corresponds to the currying mapping of a function (for n-ary function);

6-7° derive the main properties of Karoubi's shell;

6-8° determine the encoding of a product of n objects ($n \geq 5$) by terms of type free λ-calculus and derive the corresponding expressions for projections;

6-9° represent the main functions of the applicative programming language Lisp by means of λ-calculus and combinatory logic.

Formulation of the tasks for corresponding target studies in the **Work 6**, p. 39.

6-1° Let the definition of term $\lambda x.P$ be given by induction on constructing P:

$$\begin{array}{lll} 1.1) & \lambda x.x & = I, \\ 1.2) & \lambda x.P & = K\,P, \text{ if } x \notin FV(P), \\ 1.3) & \lambda x.P'P'' & = S(\lambda x.P')(\lambda x.P''). \end{array}$$

Exclude all the variables from the following λ-expressions:

$$\lambda xy.xy, \quad \lambda fx.fxx, \quad f = \lambda x.B(f(Ax)).$$

6-2° Let the definition of term M such, that $x \in FV(M)$, be given by induction on constructing M:

2.1) $\lambda x.x = I$,

2.2) $\lambda x.PQ = \begin{cases} (a)\ BP(\lambda x.Q), & \text{if } x \notin FV(P) \text{ and } x \in FV(Q), \\ (b)\ C(\lambda x.P)Q, & \text{if } x \in FV(P) \text{ and } x \notin FV(Q), \\ (c)\ S(\lambda x.P)(\lambda x.Q), & \text{if } x \in FV(P) \text{ and } x \in FV(Q). \end{cases}$

Exclude all the variables from the following λ-expressions:

$$\lambda xy.xy, \quad \lambda fx.fxx, \quad f = \lambda x.B(f(Ax)).$$

6-3° Using the fixed point function Y, represent the following definitions of functions, given by examples:

$$\begin{array}{lcl} \mathit{length}(a_5, a_2, a_6) & = & 3, \\ \mathit{sum}(1,2,3,4) & = & 10, \\ \mathit{product}(1,2,3,4) & = & 24, \\ \mathit{append}(1,2)(3,4,5) & = & (1,2,3,1,1), \\ \mathit{concat}((1,2),(3,4),()) & = & (1,2,3,4), \\ \mathit{map\ square}(1,2,3,4) & = & (1,4,9,16). \end{array}$$

For the examples, given above, perform a detailed verification of the steps of computations.

6-4° Using definition of the function $\mathit{list}1$ and the following definitions: $Ix = x$, $Kxy = x$, $\mathit{postfix}\ x\ y = \mathit{append}\ y(ux)$, where (ux) is the notation of a singular list, containing a single element x, express the functions, given below:

(a) length, $\mathit{sumsquares}$, $\mathit{reverse}$, $\mathit{identity}$;

(b) sum, $\mathit{product}$, append, concat, map.

6-5° Derive the following equalities:

$$\begin{array}{rcl} h & = & \varepsilon\circ < (\Lambda h) \circ p, q >, \\ k & = & \Lambda(\varepsilon\circ < k \circ p, q >), \end{array}$$

where:

$$\begin{array}{rcl} [x,y] & = & \lambda r.rxy, \\ < f, g > & = & \lambda t.[f(t), g(t)] = \lambda t.\lambda z.z(ft)(gt), \\ h & : & A \times B \to C, \\ k & : & A \to (B \to C), \\ \varepsilon_{BC} & : & (B \to C) \times B \to C,\ x : A,\ y : B, \\ \Lambda_{ABC} & : & (A \times B \to C) \to (A \to (B \to C)), \\ p & : & A \times B \to A, \\ q & : & A \times B \to B, \\ \varepsilon\circ < k \circ p, q > & : & A \times B \to C. \end{array}$$

6-6° Concerning a family of functions h:

$$\begin{array}{lcl} h_2 & : & A \times B \to C, \\ h_3 & : & A \times B \times C \to D, \\ h_4 & : & A \times B \times C \times D \to E, \\ \dots & : & \dots, \end{array}$$

find the family of mappings

$$\Lambda_{ABC},\ \Lambda_{(A\times B)CD},\ \Lambda_{(A\times B\times C)DE}, \dots,$$

which make currying of the functions above, i.e. transform the functions from "operator" form to applicative form.

6-7° By *Karoubi's shell* we mean the category, which contains for $a \circ b = \lambda x.a(bx)$:

sets of

$$\begin{array}{rl} \text{objects:} & \{a \mid a \circ a = a\}, \\ \text{morphisms:} & Hom(a,b) = \{f \mid b \circ f \circ a = f\}, \end{array}$$

and *morphisms of*

$$\begin{array}{rl} \text{identity:} & id\ a = a, \\ \text{composition:} & f \circ g. \end{array}$$

Assume, that

$$\begin{array}{r} [x,y] \equiv \lambda r.rxy, \\ < f,g > \equiv \lambda t.[f(t),g(t)] \equiv \lambda t.\lambda z.z(ft)(gt). \end{array}$$

Verify, that:

$$h = \varepsilon\circ < (\Lambda h) \circ p, q >,\ k = \Lambda(\varepsilon\circ < k \circ p, q >),$$

where

$$\begin{aligned} h &: A \times B \to C, \\ k &: A \to (B \to C), \\ \varepsilon_{BC} &: (B \to C) \times B \to C,\ x : A,\ y : B, \\ \Lambda_{ABC} &: (A \times B \to C) \to (A \to (B \to C)), \\ p &: A \times B \to A, \\ q &: A \times B \to B, \\ \varepsilon \circ < k \circ p, q > &: A \times B \to C. \end{aligned}$$

(**Hint**. Use the encoding of functions as $f : A \to B$ by the terms $B \circ f \circ A = \lambda x.B(f(Ax))$. Next, use the equality: $A \circ A = A(= \lambda x.A(A(x)))$. Take into account, that $\Lambda h = \lambda xy.h[x, y]$.)

6-8° Derive a term of lambda-calculus, which corresponds to the product of n objects. In addition, derive such n terms, that behave as the projections.
(**Hint**. In case of n=2:

$$\begin{aligned} A_0 \times A_1 &= \lambda u.[A_0(uK), A_1(u(KI))], \\ \pi_0^2 &= \lambda u.(A_0 \times A_1)(u)K, \\ \pi_1^2 &= \lambda u.(A_0 \times A_1)(u)(KI).) \end{aligned}$$

6-9° Represent via combinators the following set of function of the language *Lisp*:

$$\{Append,\ Nil,\ Null,\ List,\ Car,\ Cdr\}.$$

(**Hint.** For $Append \equiv \frown$ and for $Nil \equiv <>$:

(1) $A \frown (B \frown C) = (A \frown B) \frown C$

(2) $A \frown <> = <> \frown A = A$

(3) $$Null\ A = \begin{cases} 1, & \text{if } A = Nil, \\ 0, & \text{if } A \neq Nil, \end{cases}$$

(4) $List\ x = < x >$,

(5) $Car < x_1, x_2, \ldots, x_n > = x_1$,

(6) $Cdr < x_1, x_2, \ldots, x_n > = < x_2, \ldots, x_n > .$)

1.6 A recommended order of solving the tasks

1) By a variant number, select out the corresponding formulation of a task.
2) Learn of the solution of a typical task, included in the text. Similar to the solution of a typical task, perform the steps of the needed derivation.
3) Check the correctness of the obtained result (answer), performing the computations in a reversed order.

Chapter 2

Derivation of Object

A question what are the 'data' or what is the 'program' is so nice that we would not even make an attempt to answer it. It's possible to assume that "a data structure complexity is exchanging for an algorithmic complexity", but this needs to determine what are the representations for algorithm and data structure. Any case, usually the programs are observed vs data, but not always.

Possibly, of course, that there is a need to deal with the *objects*. Then a chance to consider programs by the same way as data still exists.

One of the most important issues in combinatory logic is to determine how to derive from the available objects an object with the given properties by applying the known combining modes. At a starting stage only three objects-combinators are available: I, K, S, along with their properties given by their characteristic equalities. For reasons of intuitive clarity, assume that there is a programming system with these three instructions, using only which a rather rich in expressive power programming system will be constructed. A resulting system will contain only the objects-combinators.

2.1 Principle of combinatory completeness

2.1.1 Combinatory characteristic

As known, the system of combinators I, K, S allows to construe such a programming system, or, in other words, it has a property of *combinatory completeness*.

Definition 2.1 (combinatory completeness). The set of combinators $X_1, \ldots, X_m$, which is determined by the proper conversions, is assumed *combinatory*, or *functionally complete*, if for any object X, combined from distinct variables $x_1, \ldots, x_n$, there is such a combinator U, expressible in terms of $X_1, \ldots, X_m$, and such, that

$$Ux_1 \ldots x_n = X. \qquad (U)$$

The property (U) can be considered as a *characteristic rule* for conversion of combinator U, or *combinatory characteristic* of U.

Thus, the system I, K, S gives a guarantee of possibility to construe such an object U, i.e. it is possible to construe, or

> *derive* a "program", or "procedure" U, which, when it is called on the *actual* parameters $x_1, \ldots, x_n$, *results* in X – and this is a combination of variables $x_1, \ldots, x_n$.

While dealing with an applicative computational system, note the conventions concerning the notations in use. Assume by agreement that

$$U\,x_1x_2 \ldots x_n \equiv \underbrace{((\ldots((}_{n \text{ ones}} U\,x_1)x_2)\ldots)x_n),$$

i.e. the omitted parentheses are to be restored by *association to the left*.

Example 2.1.

$$Ux_1x_2x_3 \equiv (((Ux_1)x_2)x_3).$$

2.1.2 Systems of concepts

Untyped combinatory logic is a main mathematical mean to construe the calculi for objects, which are abstract in their nature. In fact, combinatory logic is a pure calculus of *concepts*, making it possible by need to generate or modify "on the fly" its own system of concepts. Studying the systems of "variable" concepts has a priority in all the valid applied areas including the development of object-oriented programming systems. In spite of visible simplicity, – in fact, combinatory logic could be used just after learning only the combinators K and S, – a deep understanding of all the possibilities within combinatory calculus needs a special computational experience. First of all this is caused by namely the applicative style of notations in use. However, this is not unexpected: a forty years practical experience with programming system Lisp had already prepared a ground for rethinking of "minimal theoretical knowledge of programmer" which contains combinatory logic as well as λ-calculus and some other topics from logic, traditionally mentioned as non-classic logics.

2.1.3 Combinatory completeness

Combinatory logic is a branch of mathematical logic which studies combinators and their properties. More details can be found in the book (H. Curry, [6]). Combinators and the similar operators can be defined in terms of λ-conversion, and that is why the diversity of calculi of λ-conversion is assumed as a part of combinatory logic.

Principle of combinatory completeness

The calculi of λ-conversion and systems of combinatory logic are combinatory complete theories.

Definition 2.2 (combinatory completeness). Combinatory completeness in the calculi of λ-conversion is determined by the axiom

scheme (β):

$$(\lambda x.a)b = [b/x]a, \qquad (\beta)$$

where λ is an abstraction operator, expression '$[b/x]a$' denotes a result of substituting object b for each free occurrence of variable x in object a, and '=' is a symbol of relation of conversion.

Given an arbitrary object a in the systems of combinatory logic, a new object can be constructed

$$U \equiv [x_1, \ldots, x_n]a,$$

via K, S, I, where $[x_1, \ldots, x_n]$ for $n > 0$ has a role of abstraction operator by the variables x_1, ..., x_m. The principle of combinatory completeness:

$$\begin{aligned} Ub_1 \ldots b_n &\equiv ([x_1, \ldots, x_n]a)b_1 \ldots b_n \\ &= [b_1, \ldots, b_n/x_1, \ldots, x_n]a, \end{aligned}$$

can be proved for abstraction operator, where the expression '$[b_1, \ldots, b_n/x_1, \ldots, x_n]a$' denotes a result of simultaneous substituting the objects $b_1, \ldots, b_n$ in the object a instead of the corresponding occurrences of graphically distinct variables $x_1, \ldots, x_n$ for $n > 0$.

An intuitive interpretation is as follows:

"procedure" U with a *formal*, or *substitutional* parameters

$$x_1, \ldots, x_n$$

is called with the *actual* parameters $b_1, \ldots, b_n$, *resulting in*

$$[b_1, \ldots, b_n/x_1, \ldots, x_n]a.$$

2.1.4 Elementary combinatory logic

Systems of combinators serve the same functions that the systems of λ-conversion, but without using the bound variables. Thus, the technical difficulties caused by a substitution and congruence, disappear.

Introducing a concept by the combinator

Combinators can be used to determine a *concept* which corresponds to some law. In other words, This law is comprehended to a concept – this is one of the most important virtues of combinatory logic as a metatheory.

Consider a commutativity law in arithmetics:

$$\forall x, y.x + y = y + x.$$

This law can be rewritten without any usage of bound variables x and y by defining

$$\forall x, y.A(x, y) = x + y$$

and introducing metaoperator $\mathbf{C}$:

$$\forall f, x, y.(\mathbf{C}(f))(x, y) = f(y, x).$$

This law is written as follows:

$$\mathbf{C}A = A.$$

> Metaoperator $\mathbf{C}$ can be called as a 'combinator' which represents a commutativity law for the operation A.

In below, instead of indicating these metaoperators in bold face, the italics is used.

Exercise 2.1. Write, without using the variables, a commutativity law for multiplication.

Simplest combinators

To begin with, list the combinators which are most usable in practice.

Identity. A simplest combinator is the *identity* combinator I:

$$If = f.$$

Compositor. Elementary *compositor* :

$$Bfgx = f(gx)$$

determines a composition of the functions f and g.

Duplicator. Elementary *duplicator* W:

$$Wfx = fxx$$

duplicates the second argument.

Permutator. The combinator , mentioned earlier, is called elementary *permutator* and is re-written as C:

$$Cfxy = fyx.$$

Connector. Elementary *connector* S is defined by the rule:

$$Sfgx = fx(gx).$$

Cancellator. Elementary *cancellator*

$$Kcx = c$$

determines the constant (constant function) as a function of x.

Example 2.2. Let $f \equiv sin$ be the function 'sine', $g \equiv exp^5$ be the function of fifth power. Then Bfg is the sine of x to the fifth power:

$$Bfgx = B\ sin\ exp^5\ x = sin(exp^5\ x) = sin\ x^5,$$
Bgf is the fifth power of sine,
$$Bgfx = B\ exp^5\ sin\ x = exp^5(sin\ x) = sin^5 x.$$

Example 2.3. If Q is the second power operation, then BQQ or WBQ is the fourth power operation:

$$WBQ\ x \overset{W}{=} BQQ\ x \overset{B}{=} Q(Q\ x) = x^4.$$

Example 2.4. From the equation (conversion):

$$B(Bf)\ g\ x\ y\ =\ Bf\ (gx)\ y\ =\ f(gxy),$$

it follows, that if f is a differentiation operator D, then $B(Bf)$ is that for the function of two arguments:

$$B(BD)\ g\ x\ y\ =\ BD\ (gx)\ y\ =\ D\ (gxy).$$

2.2 Deriving main combinators: tasks

Let now to acquire the technical experience of establishing (and case studies) of a newly generated concept. Select the practically used combinators as those concepts. The general problem is to establish the concept/combinator by the specified *combinatory characteristics* (see p. 46).

Task 2.1. Specify the combinator B .

Task formulation. Specify the object with the given combinatory characteristics by K and S:

$$Babc = a(bc), \qquad (B)$$

using the postulates α, β, μ, ν, σ, τ, ξ from the calculus of λ-conversions.

Solution.

$B-1$. Give the formulation of the postulates for relation '=':

$$(\alpha)\quad \lambda x.a = \lambda z.[z/x]a; \qquad (\beta)\quad (\lambda x.a)b = [b/x]a;$$

$$(\nu)\quad \frac{a=b}{ac=bc}; \qquad (\mu)\quad \frac{a=b}{ca=cb};$$

$$(\xi)\quad \frac{a=b}{\lambda x.a = \lambda x.b}; \quad (\tau)\quad \frac{a=b;\ b=c}{a=c}; \quad (\sigma)\quad \frac{a=b}{b=a}.$$

$B-2$. Establish the combinatory characteristics of the objects K and S:

$$\begin{array}{lcll} x(Kyz) & = & xy, & (K)\\ x(Syzw) & = & x(yw(zw)), & (S)\end{array}$$

which within the λ-calculus are expressed by: $K = \lambda xy.x$ and $S = \lambda xyz.xz(yz)$.

Indeed, by postulate (β):

$$\begin{array}{lcll} x(Kyz) & \equiv & x(\underbrace{(\lambda xy.x)}_{\equiv K}yz) & \overset{(\beta)}{=} xy,\\ x(Syzw) & \equiv & x(\underbrace{(\lambda xyz.xz(yz))}_{\equiv S}yzw) & \overset{(\beta)}{=} x(yw(zw)),\end{array}$$

$B-3$. The schemes (K) and (S) result in:

$$\begin{array}{lcll} a(bc) & = & Kac(bc) & (K)\\ & = & S(Ka)bc & (S)\\ & = & KSa(Ka)bc & (K)\\ & = & S(KS)Kabc. & (S)\end{array}$$

Checking. A verification that $B = S(KS)K$ is given below.

B–1. $S(KS)Kabc = KSa(Ka)bc$, because in the scheme (S) it can be assumed that $y \equiv (KS)$, $z \equiv K$, $w \equiv a$. Then:

$$Syzw = S(KS)Ka, \qquad yw(zw) = (KS)a(Ka),$$

i.e. $S(KS)Ka = (KS)a(Ka)$. Omitting the non-significant parentheses results in $S(KS)Ka = KSa(Ka)$. Twice applying of postulate (ν) to this expression, results in: $S(KS)Kabc = KSa(Ka)bc$.

B–2. The same reasons by scheme (K) give $KSa = S$. Taking into account postulate (ν), obtain $KSa(Ka)bc = S(Ka)bc$.

B–3. The same way successive applying of schemes (S) and (K), and postulate (ν) and omitting the nonsignificant parentheses leads to the equations:

$$S(Ka)bc = Kac(bc); \ (Kac)bc = a(bc).$$

B–4. Repeatedly using the transitivity (τ), obtain $S(KS)Kabc = a(bc)$. (This equation is valid for if $S(KS)Kabc = KSa(Ka)bc$ and $KSa(Ka)bc = S(Ka)bc$, then $S(KS)Kabc = S(Ka)bc$ etc.)

Answer. The object B with a combinatory characteristic $Babc = a(bc)$ is $S(KS)K$, i.e. $B = S(KS)K$.

Task 2.2. Specify an expression for combinator C.

Task formulation. Using K, S and other predefined objects derive the object with combinatory characteristic:

$$Cabc = acb, \qquad (C)$$

by postulates α, β, μ, ν, σ, τ, ξ of λ-calculus.

Solution.

$C-1$. List the postulates which determine a conversion relation '$=$' (see task given above).

$C-2$. Recall the combinatory characteristics of the objects in use:

$$\begin{array}{llllllll} (K) & Kxy & = & x, & (S) & Sxyz & = & xz(yz), \\ (I) & Ix & = & x, & (B) & Bxyz & = & x(yz). \end{array}$$

Note that the scheme (B), derived in task 2.1 on p. 51 is added to already known schemes (K), (S) (I). As was shown, this newly derived scheme can be expressed in terms of schemes (K) and (S).

$C-3$. Applying these schemes to (acb), obtain:

$$\begin{array}{lcll} acb & = & ac(Kbc) & \text{(by scheme } (K)) \\ & = & Sa(Kb)c & \text{(by scheme } (S)) \\ & = & B(Sa)Kbc & \text{(by scheme } (B)) \\ & = & BBSaKbc & \text{(by scheme } (B)) \\ & = & BBSa(KKa)bc & \text{(by scheme } (K)) \\ & = & S(BBS)(KK)abc. & \text{(by scheme } (S)) \end{array}$$

Using transitivity postulate (τ), obtain:

$$S(BBS)(KK)abc = acb,$$

i.e. $C = S(BBS)(KK)$.

Answer. The object with a combinatory characteristic $Cabc = acb$ is $C = S(BBS)(KK)$.

Task 2.3. Derive combinator W.

Task formulation. Derive combinator W with the following characteristic:

$$Wab = abb. \qquad (W)$$

Solution.

$W-1$. Write down the characteristics of objects in use as follows:

$$(S)\ \ Sxyz = xz(yz),\ \ (I)\ \ Ix = x,\ \ (C)\ \ Cxyz = xzy.$$

$W-2$. Apply these schemes to abb:

$$\begin{array}{rcll} abb & = & ab(Ib) & (\text{by } (I)) \\ & = & SaIb & (\text{by } (S)) \\ & = & CSIab. & (\text{by } (C)) \end{array}$$

Using the postulates, obtain: $CSIab = abb$. Thus, $W = CSI$.

$W-3$. The additional two variants of derivations for W are as follows:

$$\begin{array}{rcl} abb & = & ab(Kba) \\ & = & ab(CKab) \\ & & Sa(CKa)b \\ & = & SS(CK)ab \end{array} \qquad \begin{array}{rcl} abb & = & ab(Kb(Kb)) \\ & = & ab(SKKb) \\ & = & Sa(SKK)b \\ & = & Sa(K(SKK)a)b \\ & = & SS(K(SKK))ab. \end{array}$$

Answer. The object W with a characteristic $Wab = abb$ is as follows: $W = CSI$ ($= SS(CK) = SS(K(SKK))$).

Task 2.4. Derive the expression for combinator Ψ.

Task formulation. Derive combinator Ψ with the following characteristic:

$$\Psi abcd = a(bc)(bd). \qquad (\Psi)$$

Solution.

$\Psi-1$. List the postulates for conversion relation.

Ψ–2. Recall the combinatory characteristics of the objects in use:

$$(C)\ Cxyz = xzy,\ \ (W)\ Wxy = xyy,\ \ (B)\ Bxyz = x(yz).$$

Ψ–3. Applying these schemes to $a(bc)(bd)$, obtain:

$$\begin{array}{rcll} a(bc)(bd) & = & B(a(bc))bd & (B)\\ & = & BBa(bc)bd & (B)\\ & = & B(BBa)bcbd & (B)\\ & = & BB(BB)abcbd & (B)\\ & = & C(BB(BB)ab)bcd & (C)\\ & = & BC(BB(BB)a)bbcd & (B)\\ & = & W(BC(BB(BB)a))bcd & (W)\\ & = & BW(BC)(BB(BB)a)bcd & (B)\\ & = & B(BW(BC))(BB(BB))abcd. & (B) \end{array}$$

Using the needed postulates, conclude that:
$B(BW(BC))(BB(BB))abcd = a(bc)(bd)$, i.e.
$\Psi = B(BW(BC))(BB(BB))$.

Answer. The object Ψ with the combinatory characteristic $\Psi abcd = a(bc)(bd)$ is $\Psi = B(BW(BC))(BB(BB))$.

Task 2.5. Derive the expression for combinator B^2.

Task formulation. Derive via K and S and other predefined objects the object with the following combinatory characteristic:

$$B^2abcd = a(bcd). \qquad (B^2)$$

Solution.

B^2–1. List the postulates for relation of conversion.

B^2–2. Recall a combinatory characteristic of the object in use:
$(B)\ Bxyz = x(yz)$.

B^2–3. Applying this scheme to $a(bcd)$, obtain:

$$\begin{array}{rcll} a(bcd) & = & Ba(bc)d & \text{(by scheme } (B)) \\ & = & B(Ba)bcd & \text{(by scheme } (B)) \\ & = & BBBabcd. & \text{(by scheme } (B)) \end{array}$$

Using the postulates, obtain: $BBBabcd = a(bcd)$, i.e. $B^2 = BBB$.

Answer. The object B^2 with a combinatory characteristic $B^2abcd = a(bcd)$ is $B^2 = BBB$.

Task 2.6. Specify the expression for combinator B^3.

Task formulation. Derive via K and S and other predefined objects an object with combinatory characteristic:

$$B^3abcde = a(bcde). \qquad (B^3)$$

Solution.

B^3–1. Use the postulates which determine a relation of conversion.

B^3–2. Combinatory characteristics of the objects in use are the following: $(B)\ \ Bxyz = x(yz)$, $(B^2)\ \ B^2xyzw = x(yzw)$.

B^3–3. Using these schemes to $a(bcde)$, obtain:

$$\begin{array}{rcll} a(bcde) & = & B^2a(bc)de & \text{(by scheme } (B^2)) \\ & = & B(B^2a)bcde & \text{(by scheme } (B)) \\ & = & BBB^2abcde. & \text{(by scheme } (B)) \end{array}$$

Using the postulates, obtain: $BBB^2abcde = a(bcde)$, i.e. $B^3 = BBB^2$.

Answer. The object B^3 with a combinatory characteristic $B^3abcde = a(bcde)$ is $B^3 = BBB^2$.

Task 2.7. Specify the expression for combinator $C^{[2]}$.

Task formulation. Derive via K and S and other predefined objects the object with a combinatory characteristic:

$$C^{[2]}abcd = acdb. \qquad (C^{[2]})$$

Solution.

$C^{[2]}$–1. Use the postulates, which determine a relation of conversion.

$C^{[2]}$–2. Recall the combinatory characteristics of the objects in use:

$$(B)\ \ Bxyz = x(yz), \qquad (C)\ \ Cxyz = xzy.$$

$C^{[2]}$–3. Using these schemes with $acdb$, obtain:

$$\begin{array}{rcll} acdb & = & C(ac)bd & \text{(by scheme } (C)) \\ & = & BCacbd & \text{(by scheme } (B)) \\ & = & C(BCa)bcd & \text{(by scheme } (C)) \\ & = & BC(BC)acbd. & \text{(by scheme } (B)) \end{array}$$

From the postulates, conclude: $BC(BC)acbd = acbd$, i.e $C^{[2]} = BC(BC)$.

Answer. The object $C^{[2]}$ with given combinatory characteristic $C^{[2]}abcd = acbd$ is $C^{[2]} = BC(BC)$.

Task 2.8. Specify an expression for combinator $C_{[2]}$.

Task formulation. Derive, using K and S and other predefined objects an object with combinatory characteristic:

$$C_{[2]}abcd = adbc. \qquad (C_{[2]})$$

Solution.

$C_{[2]}$–1. Use the needed postulates.

$C_{[2]}$–2. Write down the combinatory characteristics of the objects in use:

$$(B^2)\ \ B^2xyzw = x(yzw), \qquad (C)\ \ Cxyz = xzy.$$

$C_{[2]}$–3. Using these schemes with $adbc$, obtain:

$$\begin{array}{rcll} adbc & = & Cabdc & \text{(by scheme } (C)) \\ & = & C(Cab)cd & \text{(by scheme } (C)) \\ & = & B^2CCabcd. & \text{(by scheme } (B^2)) \end{array}$$

From the postulates, obtain: $B^2CCabcd = adbc$, i.e. $C_{[2]} = B^2CC$.

Answer. The object $C_{[2]}$ with given combinatory characteristic $C_{[2]}abcd = adbc$ is $C_{[2]} = B^2CC$.

Task 2.9. Derive an expression for combinator $C^{[3]}$.

Task formulation. Derive using K and S and other predefined objects an object with combinatory characteristic:

$$C^{[3]}abcde = acdeb. \qquad (C^{[3]})$$

Solution.

$C^{[3]}$–1. Select out the needed postulates.

$C^{[3]}$–2. Write down the combinatory characteristics of objects in use:

$$(C)\quad Cxyz = xzy, \qquad (B)\quad Bxyz = x(yz),$$
$$(C^{[2]})\quad Cxyzw = xzwy.$$

$C^{[3]}$–3. Using these schemes to $acdeb$, obtain:

$$\begin{array}{rcll} acdeb & = & C^{[2]}(ac)bde & \text{(by scheme } (C^{[2]})) \\ & = & BC^{[2]}acbde & \text{(by scheme } (B)) \\ & = & C(BC^{[2]}a)bcde & \text{(by scheme } (C)) \\ & = & BC(BC^{[2]})abcde. & \text{(by scheme } (B)) \end{array}$$

From the postulates, obtain: $BC(BC^{[2]})abcde = acdeb$, i.e. $C^{[3]} = BC(BC^{[2]})$.

Answer. The object $C^{[3]}$ with a combinatory characteristic $C^{[3]}abcde = acdeb$ is $C^{[3]} = BC(BC^{[2]})$.

Task 2.10. Specify an expression for combinator $C_{[3]}$.

Task formulation. Derive via K and S and other predefined objects an object with combinatory characteristic:

$$C_{[3]}abcde = aebcd. \qquad (C_{[3]})$$

Solution.

$C_{[3]}$–1. Select out the postulates.

$C_{[3]}$–2. Write the combinatory characteristics of objects in use:

$$(B^2)\quad B^2xyzw = x(yzw), \qquad (C)\quad Cxyz = xzy,$$
$$(C_{[2]})\quad C_{[2]}xyzw = xwyz.$$

Using these schemes with $aebcd$, obtain:

$$\begin{array}{rcll} aebcd & = & Cabecd & \text{(by scheme } (C)) \\ & = & C_{[2]}(Cab)cde & \text{(by scheme } (C_{[2]})) \\ & = & B^2C_{[2]}Cabcde. & \text{(by scheme } (B^2)) \end{array}$$

From the postulates, obtain: $B^2C_{[2]}Cabcde = aebcd$, i.e. $C_{[3]} = B^2C_{[2]}C$.

Answer. The object $C_{[3]}$ with a combinatory characteristic $C_{[3]}abcde = aebcd$ is $C_{[3]} = B^2C_{[2]}C$.

Task 2.11. Specify an expression for combinator Φ.

Task formulation. Derive using K and S and other predefined objects an object with combinatory characteristic:

$$\Phi abcd = a(bd)(cd). \qquad (\Phi)$$

Solution.

Φ–1. Use the postulates for a relation of conversion.

Φ–2. The combinatory characteristics of the objects in use are as follows:

$$(B^2)\quad B^2xyzw = x(yzw), \qquad (B)\quad Bxyz = x(yz),$$
$$(S)\quad Sxyz = xz(yz).$$

Φ–3. Using these schemes with $a(bd)(cd)$, obtain:

$$\begin{aligned} a(bd)(cd) &= Babd(cd) && \text{(by scheme } (B)) \\ &= S(Bab)cd && \text{(by scheme } (S)) \\ &= B^2SBabcd. && \text{(by scheme } (B^2)) \end{aligned}$$

From the postulates, obtain: $B^2SBabcd = a(bd)(cd)$, i.e. $\Phi = B^2SB$.

Answer. The object Φ with a combinatory characteristic $\Phi abcd = a(bd)(cd)$ is $\Phi = B^2SB$.

Task 2.12. Specify an expression for combinator Y.

Task formulation. Derive using K and S and other predefined objects an object with combinatory characteristic:

$$Ya = a(Ya). \qquad (Y)$$

Solution.

$Y-1$. Use the postulates for a relation of conversion.

$Y-2$. Write down the combinatory characteristics of the objects in use:

$$(S)\ Sxyz = xz(yz),\ (W)\ Wxy = xyy,\ (B)\ Bxyz = x(yz).$$

$Y-3$. Prove, that $Y = WS(BWB)$.

$$\begin{array}{rcll} Ya & = & \underline{WS(BWB)\,a} & \text{(by assumption)} \\ & = & \underline{S(BWB)(BWB)\,a} & \text{(by scheme } (W)) \\ & = & \underline{BWBa(BWBa)} & \text{(by scheme } (S)) \\ & = & W(Ba)(BWBa) & \text{(by scheme } (B)) \\ & = & Ba(BWBa)(BWBa) & \text{(by scheme } (W)) \\ & = & a(BWBa(BWBa)) & \text{(by scheme } (B)) \\ & = & a(\underline{S(BWB)(BWB)\,a}) & \text{(by scheme } (S)) \\ & = & a(\underline{WS(BWB)\,a}) & \text{(by scheme } (W)) \\ & = & a(Ya). & \text{(by assumption)} \end{array}$$

Hence, one of the representations of object Y is as follows: $Y = WS(BWB)$.

Answer. The object Y with a combinatory characteristic $Ya = a(Ya)$ is $Y = WS(BWB)$.

2.3 Historical remark

Purely chronologically, combinatory logic as a mathematical mean was introduced by Moses Shönfinkel in 1920. His paper was published in 1924 with a title "On building blocks of mathematical logic". In those years an idea to reduce logic to simplest possible basis of primitives was attractive, however, in our days this seems not so important. It has been shown in this paper that the logic can be avoided

of the usage of bound variables. The higher order functions made it possible to reduce logic to a language with a single *constructor* – applying function to argument, – and three primitive *constants* – U, C (which in our days is denoted as K) and S. Function is called 'higher order function', if its argument can in turn be a function, or results in a function. All these three constants are the higher order functions. The formal definitions are as follows.

Constant C, defined by

$$Cxy = (C(x))(y) = x,$$

is a *constant function*, which for any x returns x.

Constant S, defined by

$$Sfgx = ((S(f))(g))(x) = (f(x))(g(x)),$$

is a *combination* of two functions f and g. For x it results in a value, which is obtained as an application of function f of argument x to function g of argument x.

Constant U, defined by

$$UPQ = (U(P))(Q) = \forall x.\neg(P(x) \wedge Q(x)),$$

is a generalization of *Sheffer's operation*. This operation is applied to two predicates, and results in a universal generalization of negated conjunction of these two predicates.

Note. These combinators are sufficient to express the arbitrary first order predicates without bound variables, which are used in first order logic with quantification. The same result can be achieved, when the algorithm of translation the λ-expressions to combinators is used. In this case the restrictions for the first order of the predicates in use is eliminated. However, an unlimited using of this translation for the typed objects can violate a consistency. Combinatory logic tends to get rid of the typed system and the associated restrictions to overcome the strict requirements to preserve consistency.

The paper by M. Shönfinkel is an introduction to combinatory logic, which gives a clear vision, what are the initial motivations for its advances.

Chapter 3

Fixed Point

Conceptual transparency of combinatory logic makes it rather easy to learn of. After the first advances it may seems, that objects in use are always simple and finite in there nature. But this is just imaginable, and combinators can be used to represent *processes*, and, among them, the computations with cycles, which represent the known in programming manipulation with stack of recursion.

These cyclic computations are the mappings, which contain a *fixed point*.

3.1 Theoretical background.

Computations with a fixed point are the *representations* of the cycles in programs. One of the main results of λ-calculus is the Theorem 3.1, concerning a fixed point.

Theorem 3.1. For any object F there is the object X such, that $X = F\,X$:

$$\forall F \exists X \quad (X = FX).$$

Proof. See [2], p. 140. Let $P \equiv \lambda x.F(xx)$ and $X \equiv PP$. Then

$$X \equiv (\lambda x.F(xx))P = F(P\,P) \equiv FX,$$

and this turns X into a fixed point of F. □

The feature of this proof is in starting from term X and converting it to $F\,X$, but not vice versa. Combinatory logic gives a special concept-combinator Y, called as *fixed point combinator*, a mathematical meaning of which is a cycle in computations – given a map F, it returns its fixed point $(Y\,F)$.

3.1.1 Abstraction

For any term M and variable x the term $[x]M$, called the *abstraction M of x*, is defined by[1] induction on constructing the term M:

(i) $[x]x = I$;
(ii) $[x]M = KM$, if x does not belong to M;
(iii) $[x]Ux = U$, if x does not belong to U;
(iv) $[x](UV) = S([x]U)([x]V)$, if neither (ii), nor (iii) are valid.

Example 3.1. $[x]xy \overset{(iv)}{=} S([x]x)([x]y) \overset{(i)}{=} SI([x]y) \overset{(ii)}{=} SI(Ky)$.

Theorem 3.2 (comprehension principle). For any objects M, N and variable x an application of abstraction $([x]M)$ to object N is comprehended by a principle: 'substitute N for each free occurrence of x in M':

$$\forall M, N, x: \quad ([x]M)N = [N/x]M.$$

Proof. See in [93]. □

3.1.2 Multiabstraction

For any variables $x_1, \ldots, x_m$ (not obviously distinct) define:

$$[x_1, \ldots, x_m]M \overset{\text{def}}{=} [x_1]([x_2](\ldots([x_m]M)\ldots)).$$

Example 3.2. $[x, y]x = [x]([y]x) \overset{(ii)}{=} [x](Kx) \overset{(iii)}{=} K$.

[1] The term '$[x]M$', or '$[x].M$' for our purposes can, for a while, be considered similar to the term '$\lambda x.M$'.

3.1.3 Local recursion

An important application of a fixed point combinator is given by the programs with recursive definitions. Consider a local recursion with the elementary means. A local recursion of

$$E1\ where\ x\ =\ \ldots\ x\ \ldots$$

transforms into

$$([x]E1)(Y([x](\ldots x\ldots))),$$

where Y is a fixed point combinator, which is defined by the equation:

$$Y\ f = f(Y\ f).$$

For function f the expression $Y\ f$ is a fixed point of f.

When a mutual recursion is used with $where$-clauses in the text of program

```
E1 where f x = ... g ...
         g y = ... f ...
```

then it is, first of all, transformed into the expression:

```
E1 where f = [x] (... g ...)
         g = [y] (... f ...),
```

where all *free* occurrences of variables `x` and `y` are omitted. After that these two recursive definitions are transformed into general recursive definition:

```
E1 where (f,g) = ( [x](... g ...), [y](... f ...) ),
```

which can be compiled into expression:

```
([f,g]E1) (Y ([f,g]([x](... g ...), [y](... f ...)) ))
```

using already known rule.

3.2 Main tasks

Task 3.1. Learn of properties of the object

$$Y \equiv (\lambda x.(P(x\,x)a))(\lambda x.(P(x\,x)a)).$$

Task formulation. Find out a combinatory characteristic of the object, using postulates α, β, μ, ν, σ, τ, ξ of calculus of λ-conversions and schemes (K), (S):

$$Y = (\lambda x.(P(x\,x)a))(\lambda x.(P(x\,x)a)). \qquad (Y)$$

Solution.

Y–1. The expression for object Y is already known:

$$Y \equiv (\lambda x.(P(xx)a))(\lambda x.(P(xx)a)).$$

Y–2. Using rule of substitution (β) for its representation gives the following:

$$\begin{array}{rcll} Y & = & (\lambda x.(P(xx)a))(\lambda x.(P(xx)a)) & \\ & = & (P((\lambda x.(P(xx)a))(\lambda x.(P(xx)a)))a & (\beta) \\ & = & P(Y)a. & \end{array}$$

Thus, $Y = PYa = P(PYa)a = \ldots$.

Answer. Combinatory characteristic of the initial object $Y = (\lambda x.(P(xx)a))(\lambda x.(P(xx)a))$ is: $Y = PYa$.

Task 3.2. Learn of properties of the object:

$$Y \equiv S(BWB)(BWB).$$

Task formulation. Find out a combinatory characteristic of the object, using postulates α, β, μ, ν, σ, τ, ξ of calculus of λ-conversions and schemes (K), (S):

$$Y = S(BWB)(BWB). \qquad (Y)$$

Solution.

Y–1. The object Y is determined by: $Y = S(BWB)(BWB)$.

Y–2. Write down combinatory characteristics of the following objects: $Babc = abc$, $Sabc = ac(bc)$, $Wab = abb$.

Y–3. Make an application of object Y to a:

$$\begin{aligned}
Ya &\equiv \underbrace{S(BWB)(BWB)}_{\equiv Y}\,a && \text{(by Df.)}\\
&= BWBa(BWBa) && \text{(by scheme } S)\\
&= W(Ba)(BWBa) && \text{(by scheme } B)\\
&= Ba(BWBa)(BWBa) && \text{(by scheme } W)\\
&= a(BWBa(BWBa)) && \text{(by scheme } B)\\
&= a(\underbrace{S(BWB)(BWB)}_{\equiv Y}\,a) && \text{(by scheme } S)\\
&\equiv a(Ya). && \text{(by Df.)}
\end{aligned}$$

Thus, Ya is a fixed point for a.

Answer. Combinatory characteristic of the initial object $Y = S(BWB)(BWB)$ is: $Ya = a(Ya)$.

Task 3.3. Learn of properties of the object:

$$Y \equiv WS(BWB).$$

Task formulation. Find out a combinatory characteristic of the object, using postulates α, β, μ, ν, σ, τ, ξ of calculus of λ-conversions and schemes (K), (S):

$$Y = WS(BWB). \qquad (Y)$$

Solution.

Y–1. The object Y is determined by: $Y = WS(BWB)$.

Y–2. Write down combinatory characteristics of the following objects:

$$Babc = abc,\ Sabc = ac(bc),\ Wab = abb.$$

Y–3. By scheme (W), we obtain:

$$Y \equiv WS(BWB) = S(BWB)(BWB).$$

Thus, object Y has the same combinatory characteristic as the object Y from the previous task.

Y–4. Make an application of object Y to a:

$$\begin{array}{rcll} Ya & \equiv & S(BWB)(BWB)a & \text{(by Df.)} \\ & = & BWBa(BWBa) & \text{(by scheme } S\text{)} \\ & = & W(Ba)(BWBa) & \text{(by scheme } B\text{)} \\ & = & Ba(BWBa)(BWBa) & \text{(by scheme } W\text{)} \\ & = & a(BWBa(BWBa)) & \text{(by scheme } B\text{)} \\ & = & a(S(BWB)(BWB)a) & \text{(by scheme } S\text{)} \\ & \equiv & a(Ya) & \text{(by Df.).} \end{array}$$

Y–5. Finally, we obtain $Ya = a(Ya)$.

Answer. Combinatory characteristic of the object $Y \equiv S(BWB)$ is: $Ya = a(Ya)$.

Task 3.4. Learn of properties of the object:

$$Y_0 \equiv \lambda f.XX,\ \text{where } X \equiv \lambda x.f(xx). \qquad (Y_0)$$

Task formulation. Find out a combinatory characteristic of the object, using postulates α, β, μ, ν, σ, τ, ξ of calculus of λ-conversions and schemes (K), (S):

$$Y_0 \equiv \lambda f.XX,\ \text{where } X \equiv \lambda x.f(xx).$$

Solution.

Y_0–1. The object Y_0 is as follows: $Y_0 = \lambda f.XX$, where $X = \lambda x.f(xx)$.

Y_0–2. At first, consider the object (XX).

$$\begin{array}{rcll} XX & = & (\lambda x.f(xx))(\lambda x.f(xx)) & \text{(by Df.)} \\ & = & f((\lambda x.f(xx))(\lambda x.f(xx))) & \text{(by } \beta\text{)} \\ & = & f(XX). & \text{(by Df.)} \end{array}$$

Hence,

$$XX = f(XX). \qquad (*)$$

Y_0–3. Now the object Y_0 is applied to any object a:

$$\begin{array}{rcll} Y_0a & \equiv & (\lambda f.XX)a & \text{(by Df.)} \\ & = & (\lambda f.f(XX))a & \text{(by (*))} \\ & = & (\lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))))a & \text{(by Df. } X\text{)} \\ & = & a((\lambda x.a(xx))(\lambda x.a(xx))) & \text{(by } \beta\text{)} \\ & = & a((\lambda f.((\lambda f.f(xx))(\lambda x.f(xx))))a) & \text{(by } \beta, \xi\text{)} \\ & = & a((\lambda f.(XX))a) & \text{(by Df. } X\text{)} \\ & \equiv & a(Y_0a). & \text{(by Df. } Y_0\text{)} \end{array}$$

Using transitivity τ, we obtain: $Y_0a = a(Y_0a)$.

Answer. Combinatory characteristic of object Y_0 is as follows: $Y_0a = a(Y_0a)$.

Task 3.5. Learn of properties of the object:

$$Y_1 \equiv Y_0(\lambda y.\lambda f.f(yf)), \text{ where } Y_0 \equiv \lambda f.XX,\ X = \lambda x.f(xx).$$

Task formulation. Find out a combinatory characteristic of the object, using postulates α, β, μ, ν, σ, τ, ξ of calculus of λ-conversions and schemes (K), (S):

$$Y_1 \equiv Y_0(\lambda y.\lambda f.f(yf)), \text{ where } Y_0 \equiv \lambda f.XX,\ X = \lambda x.f(xx). \quad (Y_1)$$

Solution.

Y_1−1. The object Y_1 is as follows: $Y_1 = Y_0(\lambda y.\lambda f.f(yf))$, where the equalities $Y_0 = \lambda f.XX$, and $X = \lambda x.f(xx)$ are valid.

Y_1−2. Hence, $Y_0 a = a(Y_0 a)$,

$$\begin{array}{rcll} Y_1 a & = & Y_0(\lambda y.\lambda f.f(yf))a & \text{(by Df.)} \\ & = & (\lambda y.\lambda f.f(yf))(Y_0(\lambda y.\lambda f.f(yf)))a & \text{(by } (Y_0)) \\ & = & (\lambda yf.f(yf))Y_1 a & \text{(by } (Y_1)) \\ & = & a(Y_1 a). & \text{(by } \beta) \end{array}$$

Thus, the equality $Y_1 a = a(Y_1 a)$ is derived.

Answer. Combinatory characteristic of object Y_1 is as follows: $Y_1 a = a(Y_1 a)$.

Task 3.6. Use function Y for representing a circular list L.

Task formulation. Circular list L, which is defined by

$$L = (1 : 2 : 3 : L),$$

results in an infinite periodic list

$$L = 1, 2, 3, 1, 2, 3, 1, 2, 3, \ldots.$$

Find out a finite representation for this data structure, which does not use the self referenced definitions.

Solution. This circular construction gives L as a list, where the first element is 1, second element is 2, third element is 3, fourth element is 1 and so on.

L−1. Consider a chain of transformations:

$$\begin{array}{rcll} L & = & (1 : 2 : 3 : L) & \\ & = & (\lambda L.(1 : 2 : 3 : L))L. & \text{by } (\beta) \end{array}$$

$L{-}2$. Whenever $L \notin (\lambda L.(1:2:3:L))$, then, by fixed point theorem,

$$L = Y(\lambda L.(1:2:3:L)).$$

Answer. $L = Y(\lambda L.(1:2:3:L))$.

Exercises

Exercise 3.1. Consider a circular definition

```
s(n,k) =
       if k = 1
       then 1
       else if k = n
            then 1
            else s(n - 1, k - 1) + k * s(n - 1, k)
```

of Stirling numbers of the second kind. Find out a finite representation for this function, which does not contain self referenced definitions.

Hint. As, for instance, we have

```
s(4,2) = s(3,1) + 2 * s(3,2)
       = 1 + 2 * (s(2,1) + 2 * s(2,2))
       = 1 + 2 * (1 + 2 * 1) = 7,
```

then we deal with recursive computations. Try to determine the λ-abstraction of n and k, and, after that, to use a fixed point theorem.

Exercise 3.2. Avoid the cycles in the following definitions:

```
length x =
        if null x
        then 0
        else 1 + length (tail x)

factorial n =
        if zero n
        then 1
        else n × factorial (n - 1)
```

Hint. A circular definition can be transformed into a standard form, where a left part is the identifier, and a right part is the expression. This can be done by self referenced function, which involves the fixed point function Y. Characteristic property of this function is as follows: $Yf = f(Yf)$.

Self referencing definition of the function is determined by the equality $f = Ef$, where expression E does not contain free occurrences of f. One of the solutions for this equation is $f = YE$.

Chapter 4

Extensionality

Combinatory logic has a specific feature that allows to construe and use the functions with *a priori not fixed number of arguments*. This means that the answering for a question, how many argument places has the function-object in actual use, needs some care. Nevertheless, a careful using of rather simple principles of extensionality enables overcoming of this uncertainty.

4.1 Theoretical background

The model M, as a rule, needs in validation of some property. In particular, let object F be built from, at most, free variables $x_0, \ldots, x_{n-1}$ using the modes of combining. Besides namely these free variables F does not contain any other free variables. Then *combinatory completeness* of model M is understood as existing in it of such an object (concept) f, that for any variables $x_0, \ldots, x_{n-1}$ the following equality is valid:

$$fx_0 \ldots x_{n-1} = F.$$

In other words, a self standing concept f does explicitly exist – as an object in the model, – having the same meaning as F-combination

of other objects (with a restriction on the usage of free variables).

The models, in which the equality of functions f and g, evaluated with an arbitrary argument d, implies an equality of these functions as the objects, are practically often used:

$$\frac{\forall d \in D.(fd = gd)}{f = g}, \qquad (ext)$$

where, by agreement, assume, that $fd = (fd)$ and $gd = (gd)$. This kind of models is called the extensional models (ext).

In case of applicative structures, a stronger version of combinatory completeness is valid: *there is such a concept f that for any free in F variables $x_0, \ldots, x_{n-1}$:*

$$fx_0 \ldots x_{n-1} = F(x_0, \ldots, x_{n-1}),$$

or, formally:

$$\mathbf{I}f.\forall x_0 \ldots \forall x_{n-1}(fx_0 \ldots x_{n-1} = F(x_0, \ldots, x_{n-1})).$$

In this statement a symbol of *the description* '**I**' is used as an abbreviation for 'such ... , that ...'. This, in fact, is a *principle of comprehension*:

> combination of objects F, among which as the free variables the only $x_0, \ldots, x_{n-1}$ are used, is comprehended to a single object (concept) f with those and only properties, that are attributive to the combination $F(x_0, \ldots, x_{n-1})$.

This is true only for extensional applicative structures and allows to transform the complicated expression into a single object, decreasing the complexity of reasoning.

The postulate

$$\lambda x.Xx = X,\ x \notin X, \qquad (\eta)$$

plays an important role and, as will be shown, is similar to extensionality in applicative structure.

4.2 Tasks

Task 4.1. Prove the equality:

$$\lambda xy.xy = \lambda x.x.$$

Task formulation. Prove, that equality:

$$\lambda xy.xy = \lambda x.x \qquad (\overline{1})$$

is derivable in $\eta\xi$-calculus of λ-conversion (the symbol '=' represents a relation of conversion).

Proof.

η–1. The following postulates are used:

$$(\eta)\quad \lambda x.Xx = X,\ x \notin X, \quad (\xi)\quad \frac{a = b}{\lambda x.a = \lambda x.b},$$

$$(\tau)\quad \frac{a = b,\ \ b = c}{a = c}.$$

Here: '$x \notin X$' is the same as 'x has no free occurrences in X'.

η–2. Using the postulates, we obtain:

(1)	$\lambda xy.xy$	$\equiv$	$\lambda x.(\lambda y.xy)$,	(by definition)
(2)	$\lambda y.xy$	$=$	x,	(by scheme (η))
(3)	$\lambda x.(\lambda y.xy)$	$=$	$\lambda x.x$,	$((2), (\xi))$
(4)	$\lambda xy.xy$	$=$	$\lambda x.x$.	$((1), (3); (\tau))$

Thus, in $\eta\xi$-system the equality $\lambda xy.xy = \lambda x.x$ is derivable. □

An abbreviated form of this proof could be given as a tree-like derivation, which is growing in the direction 'from-top-to-bottom'. Its

premises are written above the horizontal line, and *conclusions* – below. The line is understood as a replacement of the word 'therefore':

$$\dfrac{\dfrac{\lambda y.xy \qquad x}{\lambda y.xy = x}\ (\eta)}{\lambda xy.xy = \lambda x.x}\ (\xi)$$

Task 4.2. Using λ-notations ([ ]-notations), define a construction:

```
f(x) + f(f(7)) where f(x) = sqr(x) + 3
```

Task formulation. This task assumes that the constructions

`M where f(x) = N` or `let f(x) = N M`,

will be used, whose particular cases

`M where x = N` or `let x = N M`,

could be represented by the λ-term:

$$(\lambda x.M)N.$$

Solution. Construction `where` corresponds to a programming 'from-top-to-bottom', when, at first, it is supposed that the objects exist, and after that they are defined. Construction `let` corresponds to a programming 'from-bottom-to-top', when the objects are defined before they are used.

η–1. Each of these expressions consists of two parts:

1) expression `M`, called *main expression*, or *body*;
2) definition of one of the forms:

`where f(x) = N` or `let (x) = N`.

η–2. In a general case, the expression

```
M where f(x) = N
```

can be transformed into expression with operator/operand, using two stages:

1) at first, we establish the expression

```
M where f = λx.N;
```

2) at second, we obtain the expression

```
(λf.M) (λx.N).
```

η–3. Thus, for objects

```
M ≡ f(x) + f(f(7)),   N ≡ sqr(x) + 3
```

we obtain, that

```
(λf.f(x) + f(f(7))) (λx.sqr(x) + 3).
```

Answer. The initial construction of programming language can be represented by:

```
(λf.f(x) + f(f(7))) (λx.sqr(x) + 3),
```

or, in other notations:

```
([f].f(x) + f(f(7))) ([x].sqr(x) + 3).
```

Exercises

Exercise 4.1. Learn of a solution for task 4.2 and find, where and in what context the postulate (η) is used.

Exercise 4.2. Re-formulate a self referenced definition

```
length x =
         if null x
         then 0
         else 1 + length (tail x)
```

to standard form, where the definable identifier in written in the left part, and the defining expression — in the right.

Hint. A chain of transformations:

```
 length  =  [x].if null x
            then 0
            else 1 + length (tail x)
         =  ([length].[x].if null x
                         then 0
                         else 1 + length (tail x))length
         =  Y([length].[x].if null x
                          then 0
                          else 1 + length (tail x))
```

could be determined. Postulate (η) is used. Now the definition has a standard form, and its right part does not contain the definable identifier `length`, because the variable `length` is bound.

Chapter 5

Numerals

We pay your attention that from the *very beginning* combinatory logic does not contain ... numbers – among the generic objects. This is so, because the notion of a number can be constructed using the known combinators. Then the numbers look like some unusual entities – they are the objects with their arity depending on the involved postulates. Similarly, the arithmetic operations could be derived using combinators. In other words, arithmetic entities are embedded into combinatory logic. This is the known ability of object-oriented programming – an application (the arithmetic objects with their associated rules) is embedded into programming environment (combinatory logic).

5.1 Numbers and numerals

As known, one of the generic concepts in mathematics is a notion of *number*. Using the numbers, an investigator can establish other objects, that are more meaningful to represent the concepts in a problem domain. In the theoretical studies this is a "good manner" to reduce an applied theory to some predefined arithmetic system.

A question is that is it so necessary to use as a generic concept

the notion of a number. This is one of the most intriguing question in modern mathematics, the attempts to get an answer for which lead to the far growing research conclusions.

Nevertheless, combinatory logic or λ-calculus among the generic objects has no numbers. Is an expressive power of these systems rich enough? As known, combinatory logic or λ-calculus allows the introducing of those combinators or, respectively, λ-terms, whose behavior is similar to that of numbers. These representations of the numbers are called *numerals*. Numerals, as combinators, conform to all the laws of combinatory logic. Moreover, the combinators, which represent the arithmetic operations, for instance, addition, can be defined. The research in this area is yet far from final stage.

5.2 Combinatory arithmetic

The constructing of arithmetic usually starts with an introducing the natural numbers (see A.S. Kuzichev, [18]). For this purpose in a combinatory logic two objects are introduced and use the abstraction operator (see point 3.1.1 on page 66 and so forth):

$$Z_0 \equiv [xy]y \quad \widehat{\sigma} \equiv [xyz](y(xyz)).$$

First of them is a representation of the number 0, and second — is a representation of operation of adding one '+1' and is called the *successor combinator*.

Combinatory numbers, or *numerals* are generated step by step, by induction on constructing, or by structural induction:

i) Z_0 is a combinatory number;

ii) combinatory number Z_{k+1}, as a representation of the natural number $k+1$, assumed to be equal to $\widehat{\sigma} Z_k$ for $k \geq 0$.

Statement 5.1. It can be shown that combinator Z_n has a property:

$$Z_n XY = \underbrace{X(X(\ldots X(X}_{n} Y)\ldots)),$$

where X and Y are the objects, and $n \geq 0$.

Proof. By induction on n.

i) $Z_0 = [xy]y$, and $Z_0XY = \underbrace{\quad}_{0} Y$;

ii) $Z_1 = \widehat{\sigma} Z_0$, and

$$\begin{aligned} Z_1XY &= \widehat{\sigma} Z_0XY \\ &= ([xyz](y(xyz)))([xy]y)XY \\ &= X(([xy]y)XY) \\ &= \underbrace{X}_{1} Y; \end{aligned}$$

and so on. □

For introducing other arithmetic objects the *pairing combinator* is needed

$$\mathcal{D} \equiv [xyz](z(Ky)x)$$

with the following properties:

1) $\mathcal{D}XYZ_0 = X$;

2) $\mathcal{D}XY(\widehat{\sigma}U) = Y$ for objects $X, Y\ U$.

To construe more representative arithmetic, some other arithmetic objects are to be added.

Predecessor

Operation of obtaining a predecessor '-1' is denoted by π and is defined by

$$\pi \equiv [x](x\overline{\sigma}(KZ_0)Z_1),$$

where $\overline{\sigma} \equiv [x](\mathcal{D}(\widehat{\sigma}(xZ_0))(xZ_0))$. 'Predecessor' combinator π can be viewed as the reversed operation of obtaining 'successor' $\widehat{\sigma}$.

Some of the properties of combinator π are listed below. These properties can be proved by induction on n, where $n \geq 0$:

1) $\pi Z_0 = Z_0$;

2) $Z_n \overline{\sigma}(K Z_0) Z_0 = Z_n$;

3) $\pi Z_{n+1} = Z_n$.

Modified subtraction

Operation of *modified*, or *cut subtraction* in usual arithmetic is denoted by '$\dot{-}$' and defined by:

$$a \dot{-} b = \begin{cases} a - b, & a > b; \\ 0, & a \leq b. \end{cases}$$

For representing cut subtraction in combinatory arithmetic the combinator $\mathbb{L}$ is used, and is defined by

$$\mathbb{L} \equiv [xy](y \pi x),$$

and has the following properties:

1) $\mathbb{L} Z_n Z_0 = Z_n$;

2) $\mathbb{L} Z_n Z_{m+1} = \pi(\mathbb{L} Z_n Z_m)$.

Operation of minimum

Using the combinator $\mathbb{L}$ of cut subtraction, the combinator $\overline{\min}$, which represents in a combinatory arithmetic the function of minimum $\min(a, b)$, can be established. Combinator $\overline{\min}$ is defined by

$$\overline{\min} \equiv [xy](\mathbb{L} x(\mathbb{L} xy)),$$

where

$$\overline{\min} Z_n Z_m = \begin{cases} Z_n, & n \leq m; \\ Z_m, & n > m. \end{cases}$$

Combinator of minimum, in turn, can be used for constructing other arithmetic objects.

Example 5.1. Assume, that

$$\alpha \equiv \overline{\min} Z_1.$$

It is obvious, that:

1) $\alpha Z_0 = Z_0$;

2) $\alpha Z_m = Z_1$ for $m > 0$.

Addition

Combinator $\mathbb{A}$, which represents an operation of addition, is defined by

$$\mathbb{A} \equiv [x](\mathcal{E}\mathcal{E}x),$$

where

$$\begin{array}{rclcrcl} \mathcal{E} & \equiv & [xy](Uy[z](xxz)), & & U & \equiv & [x](\mathcal{C}(\mathcal{M}xZ_1)x), \\ \mathcal{C} & \equiv & [x](xCZ_0T\mathcal{V}), & & \mathcal{V} & \equiv & [xy](xyI), \end{array}$$
$$T \equiv [xyz](\widehat{\sigma}(y(\pi x)z)).$$

Exercise 5.1. Verify, that combinator $\mathbb{A}$ has all the properties of operation of addition '+', i.e., for $n, m \geq 0$:

$$\mathbb{A}Z_0Z_m = Z_m; \qquad \mathbb{A}Z_{n+1}Z_m = \widehat{\sigma}(\mathbb{A}Z_nZ_m).$$

Symmetric difference

Symmetric, or *positive difference* in usual arithmetic is the operation '$(x \dot{-} y) + (y \dot{-} x)$'. Combinator $\mathbb{R}$, representing a symmetric difference, is defined by

$$\mathbb{R} \equiv [xy](\mathbb{A}(Lxy)(Lyx)).$$

Multiplication

Combinator of *multiplication* $\mathbb{M}$ is defined similar to the combinator of addition $\mathbb{A}$ with the exception, that combinator $\mathcal{V}$ is replaced by combinator $\mathcal{V}'$:

$$\mathcal{V}' \equiv [xy](xy(KZ_0)),$$

and combinator T is replaced by combinator T':

$$T' \equiv [xyz](\mathbb{A}(y(\pi x)z)z).$$

Exercise 5.2. Prove, that combinator $\mathbb{M}$ has the properties of multiplication operator for arithmetic objects:

$$\mathbb{M}Z_0Z_m = Z_0; \qquad \mathbb{M}Z_{n+1}Z_m = A(\mathbb{M}Z_nZ_m)Z_m.$$

Combinatory definable functions

Addition and multiplication are the instances of the class of *combinatory definable* functions. Necessary constructions see, e.g., in (A.S. Kuzichev, [18]; H. Barendregt, [2]). The notions that are most significant for combinatory definable functions are listed below.

Definition 5.1 (combinatory definability). The function ϕ of n arguments is called *combinatory definable*, if there is an object X such, that for any set of n natural numbers $r_1, \ldots, r_n$, for which

$$\phi(r_1, \ldots, r_n) = r,$$

where r is a natural number, the equality

$$X\overline{r_1}\ldots\overline{r_n} = \overline{r}$$

is valid for $n \geq 0$. Here: for $n = 0$ we have $X = \overline{r}$; arithmetic objects (numerals), representing the natural numbers r_i are overlined $-\ \overline{r_i}$.

Thus, an overline is observed as mapping:

$$\overline{\ }: o \mapsto \overline{o}, \quad \overline{\ }(o) = \overline{o},$$

which, for object o, builds corresponding arithmetic object $\overline{o}$.

Primitive recursion

The way of constructing the combinators, which represent addition and multiplication, can be generalized to arbitrary functions, definable by the scheme of *primitive recursion*:

$$\begin{array}{rcl} \phi(0, r_1, \ldots, r_n) & = & \psi(r_1, \ldots, r_n); \\ \phi(r+1, r_1, \ldots, r_n) & = & \psi(r, \phi(r, r_1, \ldots, r_n), r_1, \ldots, r_n), \end{array}$$

where ϕ and ψ are predefined functions of corresponding number of arguments.

Indeed, it can be assumed that ϕ and ψ are combinatory definable by the objects g and H respectively. As an object, which combinatory defines the function ϕ, that is defined by the recursion scheme, we can select

$$\mathcal{E} \equiv [x](\mathcal{E}^*\mathcal{E}^* x),$$

where:

$$\begin{array}{rclcrcl} \mathcal{E}^* & \equiv & [xy](U\underline{y}[z](xxz)), & & U & \equiv & [x](C^*(\overline{\min}\; x\; \overline{1})x), \\ C^* & \equiv & [x](xC\overline{0}TV), & & V & \equiv & [xy](xyg), \\ & & T \equiv [xyz_1 \ldots z_n]H(\pi x)(y(\pi x)z_1 \ldots z_n)z_1 \ldots z_n. & & & & \end{array}$$

For this choice, it can be shown that:

$$\begin{array}{rcl} g\,\overline{0}\,\overline{r_1}\ldots\overline{r_n} & = & \overline{\psi(r_1, \ldots, r_n)}, \\ g\,\overline{r+1}\,\overline{r_1}\ldots\overline{r_n} & = & \overline{\psi(r, \phi(r, r_1, \ldots, r_n), r_1, \ldots, r_n)}. \end{array}$$

5.3 Tasks

Task 5.1. Define the objects with the properties of natural numbers (numerals) and learn of their properties.

Task formulation. Numerals are the following objects:

$$\overline{n} \stackrel{\text{def}}{=} \lambda xy.(x^n)y,$$

where n is a natural number from the set $\{1, 2, 3, \dots\}$. Show, that numerals are the objects with characteristics:

$$\overline{n} = (SB)^n(KI). \qquad (\overline{n})$$

Solution.

$\overline{n}$–1. The notion of $x^n y$ is defined by induction:

$$\begin{array}{lrcl} (i) & x^0 y & = & y, \\ (ii) & x^{n+1} y & = & x(x^n y),\ n \geq 0. \end{array}$$

Thus, $x^4 y = x(x(x(xy)))$.

$\overline{n}$–2. A behavior of the objects $\overline{n} = (SB)^n(KI)$ for $n = 0, 1$ is verified by:

$$\begin{array}{rcl} \overline{0} & = & (SB)^0(KI) = KI, \\ \overline{0}ab & = & KIab = Ib = b = (\lambda xy.y)ab, \\ \overline{1} & = & SB(KI), \\ \overline{1}ab & = & SB(KI)ab = Ba(KIa)b = BaIb \\ & = & a(Ib) = ab = (\lambda xy.xy)ab. \end{array}$$

$\overline{n}$–3. A behavior of $\overline{n} = (SB)^n(KI)$ in a general case is verified by:

$$\begin{array}{rcll} \overline{n}ab & = & (SB)^n(KI)ab & \\ & = & SB((SB)^{n-1}(KI))ab & (\text{ by Df.}) \\ & = & Ba((SB)^{n-1}(KI)a)b & (\text{ by } (S)) \\ & = & a((SB)^{n-1}(KI)ab) & (\text{ by } (B)) \\ & = & a(\overline{n-1}ab) & (\text{ by Df.}) \\ & = & a(a^{n-1}b) & (\text{ by Df.}) \\ & = & a^n b = (\lambda xy.x^n y)ab. & (\text{ by Df.}) \end{array}$$

Answer. Numerals $\overline{n} = (\lambda xy.x^n y)$ are the objects as $(SB)^n(KI)$.

Task 5.2. Determine an object representing the operation of '+1' on a set of numerals and learn of its properties.

Task formulation. Show, that $\sigma = \lambda xyz.xy(yz)$ determines the 'successor' function ('adding of one') on a set of numerals:

$$\sigma\overline{n} = \overline{n+1}. \qquad (\sigma)$$

Solution.

$\sigma-1$. Combinatory characteristic of the numerals is as follows:

$$\overline{n}ab = a^n b.$$

$\sigma-2$. The function σ can be applied to the numeral $\overline{n}$ in a general case:

$$\begin{array}{rcll} \sigma\overline{n}ab & = & (\lambda xyz.xy(yz))\overline{n}ab & (\text{ by Df.}) \\ & = & \overline{n}a(ab) & (\text{ by } (\beta)) \\ & = & a^n(ab) & (\text{ by Df. } (\overline{n})) \\ & = & a^{n+1}b & (\text{ by Df.}) \\ & = & \overline{n+1}ab. & (\text{ by Df.}) \end{array}$$

Thus, $\sigma\overline{n}ab = \overline{n+1}ab$, i.e. $\sigma\overline{n} = \overline{n+1}$.

Answer. Function $\sigma = \lambda xyz.xy(yz)$ is the successor function for numerals $\overline{n} = \lambda xy.x^n y$.

Task 5.3. Determine an object, which returns the length of a finite sequence (list).

Task formulation. Show, that the function

$$Length = \lambda xy.Null\ x\ \overline{0}(\sigma(Length(Cdr\ x))y)$$

is a function, which returns the length of a list x:

$$Length < a_1, a_2, \ldots, a_n >= n \qquad (Length)$$

Solution.

Length−1. The auxiliary functions $Null$ and Cdr can be introduced:

$$Null\ x = \begin{cases} \overline{1}, & x = NIL = \langle\,\rangle \\ & (x \text{ is an empty list}), \\ \overline{0}, & \text{otherwise}; \end{cases} \qquad (Null)$$

$$\begin{array}{lcll} \overline{1} & = & \lambda xy.xy = \lambda x.x = I, & (\overline{1}) \\ \overline{0} & = & \lambda xy.y = KI, & (\overline{0}) \\ \sigma\,\overline{n} & = & \overline{n+1}, & \end{array}$$

$$Cdr\ x = \begin{cases} NIL = \langle\,\rangle, & \text{for } x = \langle a_1\rangle, \\ \langle a_2, \ldots, a_n\rangle, & \text{for} \\ & x = \langle a_1, a_2, \ldots, a_n\rangle. \end{cases} \qquad (Cdr)$$

Length−2. The function $Length$ is applied to the empty list Nil:

$$\begin{array}{lll} Length\ Nil\ = & & \\ = \lambda y.Null\ Nil\ \overline{0}(\sigma(Length(Cdr\ Nil))y) & & (\text{ by } (\beta)) \\ = \lambda y.\overline{1}\ \overline{0}(\sigma(Length(Cdr\ Nil))y) & & (\text{ by } (Null)) \\ = \lambda y.I(KI)(\sigma(Length(Cdr\ Nil))y) & & (\text{ by } (\overline{0}), (\overline{1})) \\ = \lambda y.KI(\sigma(Length(Cdr\ Nil))y) & & (\text{ by } (I)) \\ = \lambda y.I & & (\text{ by } (K)) \\ = \lambda y.(\lambda z.z) & & (\text{ by } (I)) \\ = \lambda yz.z = \overline{0}. & & (\text{ by } (\overline{0})) \end{array}$$

Thus, the function $Length$ is sound relatively application to the empty list, i.e. $Length\ Nil = \overline{0}$.

Length−3. The function *Length* can be applied to the list x, consisting of a single element: $x = \langle a \rangle$:

$$\begin{aligned}
&Length\ x = \\
&= \lambda y.Null\ x\overline{0}(\sigma(Length(Cdr\ x))y) && (\beta)\\
&= \lambda y.\overline{0}\ \overline{0}(\sigma(Length\ Nil)y) && (Null),\ (Cdr)\\
&= \lambda y.KI(KI)(\sigma\overline{0}y) && (\overline{0})\\
&= \lambda y.I(\sigma\overline{0}y) && (K)\\
&= \lambda y.\sigma\overline{0}y && (I)\\
&= \lambda y.\overline{1}y && (\sigma)\\
&= \lambda y.Iy = \lambda y.y = I = \overline{1}.
\end{aligned}$$

Thus, the function *Length* is sound relatively application to the list, consisting of a single element, i.e. it equals to $\overline{1}$.

Length−4. At last, the function *Length* will be verified in a general case, for non empty list x of length n: $x \neq Nil$, where symbol '$\neq$' means 'is not convertible to':

$$\begin{aligned}
Length\ x &= \lambda y.Null\ x\ \overline{0}(\sigma(Length(Cdr\ x))y)\\
&= \lambda y.\overline{0}\ \overline{0}(\sigma(Length(Cdr\ x))y)\\
&= \lambda y.\sigma(\underline{Length(Cdr\ x)})y\\
&= \lambda y.\sigma\overline{n-1}\ y\\
&= \lambda y.\overline{n}y\\
&= \lambda y.(\lambda xz.x^n z)y = \lambda y.\lambda z.y^n z = \lambda yz.y^n z = \overline{n}.
\end{aligned}$$

Answer. The function *Length* actually returns the length of a list.

Exercises

Confirm or neglect the following.

Exercise 5.3. Addition is defined by the λ-expression

$$\lambda mnfx.mf(nfx).$$

Exercise 5.4. Multiplication is defined by the λ-expression

$$\lambda m \lambda n \lambda f.m(nf).$$

Exercise 5.5. Exponentiation is defined by the λ-expression

$$\lambda m \lambda n.nm,$$

because, for instance, $Z_3 Z_2 f x = (Z_2)^3 f x = f^8 x$.

Chapter 6

Typed combinators

The notion of a class is one of the most basic in object-oriented reasonings. In this case the class is understood as a pattern for constructing the instances of the particular objects. Moreover, classes themselves could be considered as the objects. The same way, combinators could be classified, or typed. The higher orders of functional spaces are significant for combinators. Nevertheless, an intuitive clarity of manipulations with the combinators as with objects is not lost.

6.1 Notion of a type

The non-formal discussion, what kind of entity is a type, exemplifies rather transparent idea. Each function has a domain and range. Hence, not all the arguments are of interest, but those of them that belong to the indicated domain. This means, that the arguments as objects are type assigned.

Pure type systems

It is assumed, that pure type systems are the families of typed λ-calculi, every member of which is characterized by a triple $\langle S, A, R\rangle$, where:

S is a subset of constants from the system, that are the *sorts*;

A is a subset of *axioms* like

$$c : s,$$

where c is a constant, and s is a sort;

R is a set of triples of sorts, each of them determines, what of the functional spaces could be constructed in the system and on what sorts each of functional spaces is constructed.

Each of the pure type systems is a formal system with sentences, that are constructed as the derivable statements like this:

$$\text{context} \vdash \text{type assignment}.$$

Using these statements, the type in this context is assigned to a term of λ-calculus.

Constructing a class of the types

A class of the types is generated as follows. At a starting point, there is a finite or infinite set of the *basic types*:

$$\delta_1,\ \delta_2,\ \delta_3,\ \ldots\ \delta_n,\ \ldots,$$

and every of them has an intuitive interpretation as the associated set. Next, the inductive class of types is generated:

i) a basic type is a *type*;

ii) if a and b are the types, then $(a \to b)$ is a *type*:

$$\frac{a \ - \ \text{type}, \qquad b \ - \ \text{type}}{(a \to b) \ - \ \text{type}}.$$

An intuitive interpretation for '$(a \to b)$' is a 'set of all the mappings from a to b'. By agreement, the notations '$(a \to b)$', '$(a\ b)$', '$(a,\ b)$' are assumed as having the same meaning. The parentheses, omitted from the notation of a type, can be restored *by association to the right*. This agreement is exemplified as follows:

Example 6.1.

$$\begin{aligned}(a\ (b\ c)\ d) &\equiv (a\ ((b\ c)\ d)) \equiv (a \to ((b \to c) \to d)),\\ (a\ b\ c\ d) &\equiv (a\ b\ (c\ d)) \quad \equiv (a(b(c\ d))),\end{aligned}$$

i.e. '$(a\ (b\ c)\ d)$' means '$(a \to ((b \to c) \to d))$', and '$(a\ b\ c\ d)$' means '$(a \to (b \to (c \to d)))$', and they are distinct types.

A central notions in applicative computational systems are *not* the domains of the functions, but the functions themselves as the general *correspondences*. In fact, namely these correspondences are the entities in the calculus, they are *concepts* of the functions, i.e., the objects in their own meaning. In this case, more augmented reasonings are needed to capture a sense of types. Indeed, combinators represent the functions, functions of functions, functions of functions of functions, ..., i.e. they are higher order functions, or functionals. The task of establishing a type of the object becomes non-trivial, and the types are generated in accordance with the definite rules. The resulting domains become strongly interrelated. The contradictions could be observed in corresponding logical constructions.

All of this needs more rigorous background. At first, a ground notation of a type

$$'a \to b'$$

has a diversity of equal notational variants:

$$a\,b, \quad b^a, \quad F\,a\,b$$

and some others.

6.1.1 Combinatory terms

For a combinatory term X, i.e. for such a term, that is derived by the postulates of combinatory logic, we say that

'a type a is assigned to combinator X',

or,

$$\vdash \#(X) = a$$

if and only if this statement is derived from the following axioms and the rule:

Axiom schemes:

$$\begin{array}{lcll} \vdash \#(I) & = & (a,a), & (FI) \\ \vdash \#(K) & = & (a,(b,a)) = (a,b,a), & (FK) \\ \vdash \#(S) & = & ((a,(b,c)),((a,b),(a,c))). & (FS) \end{array}$$

Rule:

$$\frac{\vdash \#(X) = (a,b), \qquad \vdash \#(U) = a}{\vdash \#(XU) = b}. \qquad (F)$$

Note, that according to the rule (F), type scheme is assigned to the *application* (XU) of object X to object U, when the type schemes of these component objects are known. An object X is viewed as a function from a to b, and U – as an argument of this function, which is taken from the domain (of) a. Then the values of a result $(X\,U)$ of applying X to U are taken from the range (of) b.

6.1.2 λ-terms

Note, that in case of λ-terms a type assigning for objects can be suitably done, using the following to rules:

Rules:

$$\frac{\vdash \#(x) = a, \qquad \vdash \#(X) = b}{\vdash \#\lambda x.X = (a, b)}, \tag{λ}$$

$$\frac{\vdash \#(X) = (a, b), \qquad \vdash \#(U) = a}{\vdash \#(XU) = b}\ , \tag{F}$$

where X, U are λ-terms, and x is a variable.

Note, that according the rule (F), an assignment of type scheme of *application* $(X\,U)$ of object X to object U can be obtained, when the type schemes of these last objects are known. An object X is viewed as a function from a to b, and U – as an argument of this function, which is taken from the domain (of) a. Then the values of a result $(X\,U)$ of applying X to U are taken from the range (of) b.

In accordance with the rule (λ), for known type scheme a, assigned to variable x, and known type scheme b, assigned to term X, the type scheme (a, b) is introduced, and this type scheme is assigned to the term $\lambda x.X$, and this term is interpreted as a function from a to b.

6.2 Tasks

Using the axioms and rule (F), find the type assignments for the main combinators, that are listed in Table 6.1. During the process of solving these tasks, get more knowing in that, what are the mathematical functions, how to obtain their composition, and how to construct the simplest programs, using the method of composing. Each combinator gives an idealization of a program as a "black box". This means, that the knowing of intrinsic structure of a program is not so important to be determined, but it is more important to establish

Table 6.1: The list of main combinators.

(1)	$\#(B)$,	where $B = S(KS)K$,
(2)	$\#(SB)$,	
(3)	$\#(Z_0)$,	where $Z_0 = KI$,
(4)	$\#(Z_1)$,	where $Z_1 = SB(KI)$,
(5)	$\#(Z_n)$,	where $Z_n = (SB)^n(KI)$,
(6)	$\#(W)$,	where $W = CSI$,
(7)	$\#(B^2)$,	where $B^2 = BBB$,
(8)	$\#(B^3)$,	where $B^3 = BBB^2$,
(9)	$\#(C^{[2]})$,	where $C^{[2]} = BC(BC)$,
(10)	$\#(C^{[3]})$,	where $C^{[3]} = BC(BC^{[2]})$,
(11)	$\#(C_{[2]})$,	where $C_{[2]} = B^2CC$,
(12)	$\#(C_{[3]})$,	where $C_{[3]} = B^2C_{[2]}C$,
(13)	$\#(\Phi)$,	where $\Phi = B^2SB$,
(14)	$\#(Y)$,	where $Y = WS(BWB)$,
(15)	$\#(D)$,	where $D = C_{[2]}I$,
(16)	$\#(C)$,	where $C = S(BBS)(KK)$.

the behavior of a program, knowing, what are its input and output values. Combinations (compositions), constructed from the combinators, give an ability to observe the arbitrary programs as applicative forms. Applicative forms have a simple structure: its counterparts drop down to the left part and the right part, so that a binary tree is the representation of applicative form. Note, that particular branches of this tree can be evaluated independently of others, giving rise to ability of parallel computations.

Task 6.1. Assign a type to the object: $\#(B)$.

Hint. The construction of $S(KS)K$ can be represented by a tree, which is as follows:

$$
\begin{aligned}
&Exp\,1 = (a_1, (b_1, a_1))\\
&Exp\,2 = a_1\\
&Exp\,3 = ((b_1, a_1), ((a_2, b_2), (a_2, c_2)))\\
&Exp\,4 = (b_1, a_1)\\
&Exp\,5 = ((a_2, b_2), (a_2, c_2))\\
&Exp\,6 = (a_2, b_2)\\
&Exp\,7 = (a_2, c_2)
\end{aligned}
$$

$$
\dfrac{\dfrac{\dfrac{\vdash \#(K) = Exp\,1 \qquad \vdash \#(S) = Exp\,2}{\vdash \#(S) = Exp\,3}\,(F) \qquad \vdash \#(KS) = Exp\,4}{\vdash \#(S(KS)) = Exp\,5}\,(F) \qquad \vdash \#(K) = Exp\,6}{\vdash \#(S(KS)K) = Exp\,7}\,(F)
$$

Solution.

$\#(B)-1$. Let a, b, c be the known types. As, by scheme (FK), the type assignment is $\vdash\ \#\,(K) = (a_1, (b_1, a_1))$, and, by scheme $(FS) : \vdash\ \#\,(S) = a_1$, then by rule $(F) : \vdash \#\,(KS) = (b_1, a_1)$, where $a_1 = ((a, (b, c)), ((a, b), (a, c)))$.

$\#(B)-2$. Next, the type is assigned by scheme (FS):

$$\vdash\ \#\,(S) = ((b_1, a_1), ((a_2, b_2), (a_2, c_2))),$$

where $b_1 = a_2$, $a_1 = (b_2, c_2)$, i.e. $b_2 = (a, (b, c))$, $c_2 = ((a, b), (a, c))$.

$\#(B)$–3. From $\vdash \#(KS) = (b_1, a_1)$ and rule (F) it follows, that:

$$\vdash \#(S(KS)) = ((a_2, b_2), (a_2, c_2)).$$

By scheme (FK) the type is: $\vdash \#(K) = (a_3, (b_3, a_3))$, where $a_3 = a_2$, $(b_3, a_3) = b_2$, i.e. $b_3 = a$, $a_3 = (b, c)$.

$\#(B)$–4. Thus, $a_2 = a_3 = (b, c)$. From $\vdash \#(S(KS)) = ((a_2, b_2), (a_2, c_2))$ and rule (F) we obtain:

$$\vdash \#(S(KS)K) = (a_2, c_2) = ((b, c), ((a, b), (a, c))).$$

Answer. B has a type: $\#(B) = ((b, c), ((a, b), (a, c)))$.

The other types that should be assigned to the rest of the combinators will be defined similarly.

Task 6.2. Assign a type to the object: #(SB).

Solution.

$\#(SB)$–1. Building of the tree:

$$\begin{aligned} &Exp\,1 = ((a_1, (b_1, c_1)), ((a_1, b_1), (a_1, c_1))) \\ &Exp\,2 = (a_1, (b_1, c_1)) \\ &Exp\,3 = ((a_1, b_1), (a_1, c_1)) \end{aligned}$$

$$\frac{\vdash \#(S) = Exp\,1 \qquad \vdash \#(B) = Exp\,2}{\vdash \#(SB) = Exp\,3}\ (F)$$

$\#(SB)$–2. A type scheme for B is as follows: $\vdash \#(B) = ((b, c), ((a, b), (a, c)))$, but $\vdash \#(B) = (a_1, (b_1, c_1))$, i.e. $a_1 = (b, c)$, $b_1 = (a, b)$, $c_1 = (a, c)$.

Thus, $\vdash \#(SB) = (((b, c), (a, b)), ((b, c), (a, c)))$.

Answer. (SB) has a type $(((b,c),(a,b)),((b,c),(a,c)))$.

Task 6.3. Assign a type to the object: $\#(Z_0)$.

Solution.

$\#(Z_0)$−1. $Z_0 = KI$.

$\#(Z_0)$−2.

$$\frac{\vdash \#(K) = (a_1,(b_1,a_1)) \qquad \vdash \#(I) = a_1}{\vdash \#(KI) = (b_1,a_1)}\ (F)$$

$\#(Z_0)$−3. By scheme $(FI) :\vdash \#(I) = a_1$, where $a_1 = (a,a)$; the type of b_1 is distinct from a_1, i.e. $b_1 = b$ (here: a, b are the known types).

Thus, $\vdash \#(KI) = (b,(a,a))$.

Answer. $Z_0 = KI$ has the type $(b,(a,a))$.

Task 6.4. Assign a type to the object: $\#(Z_1)$.

Solution.

$\#(Z_1)$−1. $Z_1 = SB(KI)$.

$\#(Z_1)$−2. $(F(KI)) :\vdash \#(KI) = (b,(a,a))$;
$(F(SB)) :\vdash \#(SB) = (((b,c),(a,b)),((b,c),(a,c)))$.

$\#(Z_1)$−3. $Exp\,1 = (((b_1,c_1),(a_1,b_1)),((b_1,c_1),(a_1,c_1)))$
$Exp\,2 = ((b_1,c_1),(a_1,b_1))$
$Exp\,3 = ((b_1,c_1),(a_1,c_1))$

$$\frac{\vdash \#(SB) = Exp\,1 \qquad \vdash \#(KI) = Exp\,2}{\vdash \#(SB(KI)) = Exp\,3}\ (F)$$

$\#(Z_1)$−4. By $(F(KI)) :\vdash \#(KI) = ((b_1,c_1),(a_1,b_1))$, where $(b_1,c_1)) = b, a_1 = a, b_1 = a$. Type c_1 differs from a and b, $c_1 = c$.

Thus, the type is as follows: $\vdash \#(Z_1) = ((a,c),(a,c))$. Note, that the statement $\vdash\ \#\,(Z_1) = ((a,b),(a,b))$ is valid as well (the difference is just notational).

Answer. $Z_1 = SB(KI)$ has the type $((a,b),(a,b))$.

Task 6.5. Assign a type to the object: $\#(Z_n)$.

Solution. At first, a type $\#(Z_2)$ is to be established.

$\#(Z_2)$–1. $Z_2 = SB(SB(KI))$.

$\#(Z_2)$–2. $(FZ) :\vdash \#(Z_1) = ((a,b),(a,b))$.

$\#(Z_2)$–3.
$$\begin{aligned} Exp\,1 &= (((b_1,c_1),(a_1,b_1)),((b_1,c_1),(a_1,c_1))) \\ Exp\,2 &= ((b_1,c_1),(a_1,b_1)) \\ Exp\,3 &= ((b_1,c_1),(a_1,c_1)) \end{aligned}$$

$$\frac{\vdash \#(SB) = Exp\,1 \qquad \vdash \#(Z_1) = Exp\,2}{\vdash \#(Z_2) = Exp\,3}\ (F)$$

$\#(Z_2)$–4. By scheme $(FZ) :\vdash \#(Z_1) = ((b_1,c_1),(a_1,b_1))$, where a pair of equalities should be valid simultaneously: $(b_1,c_1) = (a,b)$, $(a_1,b_1) = (a,b)$, i.e.:

$$\left.\begin{array}{ll} b_1 = a, & c_1 = b, \\ a_1 = a, & b_1 = b. \end{array}\right\} \qquad (*)$$

These equalities (*) are valid if and only if (iff) $a_1 = b_1 = c_1 = a = b$. Thus, $\vdash\ \#\,(Z_2) = ((a,a),(a,a))$.

Now type $\#(Z_n)$ should be determined.

$\#(Z_n)$–1. $Z_n = (SB)^n(KI) = SB((SB)^{n-1}(KI))$, where $n > 2$.

$\#(Z_n)$–2. $(FZ_2) :\vdash \#(Z_2) = ((a,a),(a,a))$.

$\#(Z_n)$–3. $Exp\,1 = (((b_1, c_1), (a_1, b_1)), ((b_1, c_1), (a_1, c_1)))$
$Exp\,2 = ((b_1, c_1), (a_1, b_1))$
$Exp\,3 = ((b_1, c_1), (a_1, c_1))$

$$\frac{\vdash \#(SB) = Exp\,1 \qquad \vdash \#(Z_2) = Exp\,2}{\vdash \#(Z_3) = Exp\,3}\ (F)$$

$\#(Z_n)$–4. By scheme (F) $: \vdash \#(Z_2) = ((b_1, c_1), (a_1, b_1))$, $b_1 = a$, $c_1 = a$, $a_1 = a$, i.e. $\vdash \#(Z_3) = ((a, a), (a, a))$. The following equality is obtained: $\#(Z_2) = \#(Z_1)$.
By induction, adding the step for n, it can be derived that $\vdash \#(Z_n) = ((a, a), (a, a))$, where $n > 1$.

Answer. The objects $Z_n = (SB)^n(KI)$, where $n > 1$, have been assigned the same type: $((a, a), (a, a))$.

Task 6.6. Assign a type to the object: $\#(W)$.

Solution.

$\#(W)$–1. $W = CSI$.

$\#(W)$–2. $(FC) : \vdash \#(C) = ((b, (a, c)), (a, (b, c)))$. This statement will be proved below (see Task 6.16).

$\#(W)$–3. $Exp\,1 = ((b_1, (a_1, c_1)), (a_1, (b_1, c_1)))$
$Exp\,2 = (b_1, (a_1, c_1))$
$Exp\,3 = (a_1, (b_1, c_1))$
$Exp\,4 = (a_1)$
$Exp\,5 = (b_1, c_1)$

$$\frac{\dfrac{\vdash \#(C) = Exp1 \qquad \vdash \#(S) = Exp\,2}{\vdash \#(CS) = Exp\,3}\ (F) \qquad \vdash \#(I) = Exp\,4}{\vdash \#(SCI) = Exp\,5}\ (F)$$

$\#(W)$–4. By scheme $(FS) :\vdash \#(S) = (b_1, (a_1, c_1))$, however $\vdash\ \#\,(S) = ((a,(b,c)),((a,b),(a,c)))$.
Thus, $b_1 = (a,(b,c))$, $a_1 = (a,b)$, $c_1 = (a,c)$. Similarly, the following is valid: $(FI) :\vdash\ \#\,(I) = a_1$. But $\vdash \#(I) = (a,a)$, i.e. $a_1 = (a,a)$ $a = b$. The following equalities are valid: $a = b$, $a_1 = (a,a)$, $b_1 = (a,(a,c))$, $c_1 = (a,c)$.

Thus, $\vdash\ \#\,(W) = ((a,(a,c)),(a,c))$, and this is equal to:
$\vdash\ \#\,(W) = ((a,(a,b)),(a,b))$.

Answer. $W = CSI$ has the type $((a,(a,b)),(a,b))$.

Task 6.7. Assign a type to the object: $\#(B^2)$.

Solution.

$\#(B^2)$–1. $B^2 = BBB$.

$\#(B^2)$–2. $(B) :\ ((bc)((ab)(ac)))$.
Here and thereafter it is assumed that a notion of:

$$\text{`}\vdash \#(X) = (a,(b,c))\text{'}$$

is similar to:

$$\text{`}(X) :\ (a(b\,c))\text{'}.$$

In the following commas "," will be omitted, i.e.

$$\text{`}(X) :\ (a,(b,c))\text{'}$$

is the same as

$$\text{`}(X) :\ (a(b\,c))\text{'}.$$

$\#(B^2)$–3. Find, at first $\#(BB)$:

$$\frac{(B):\ ((b_1\ c_1)((a_1\ b_1)(a_1\ c_1)))\ \ (B):\ (b_1\ c_1)}{(BB):\ ((a_1\ b_1)(a_1\ c_1))}\ \ (F)$$

where $b_1 = (b_2\ c_2)$, $c_1 = ((a_2\ b_2)(a_2\ c_2))$,
$(BB):\ ((a_1(b_2\ c_2))(a_1((a_2\ b_2)(a_2\ c_2))))$.
Assume: $a_1 = a$, $b_2 = b$, $c_2 = c$, $a_2 = d$. Then (BB) : $((a(b\ c))(a((d\ b)(d\ c))))$.

Find now $\#(BBB)$.

$$\frac{(BB):\ ((a_1(b_1\ c_1))(a_1((d_1\ b_1)(d_1\ c_1))))\ \ (B):(a_1(b_1\ c_1))}{(BBB):\ (a_1((d_1\ b_1)(d_1\ c_1)))},\ \ (F)$$

where $(a_1(b_1\ c_1)) = ((b\ c)((a\ b)(a\ c)))$, i.e.: $a_1 = (b\ c)$, $b_1 = (a\ b)$, $c_1 = (a\ c)$. Let $d_1 = d$.
Thus, $(BBB):\ ((b\ c)((d(a\ b))(d(a\ c))))$.

Answer. $B^2 = BBB$ has the type $((b\ c)((d(a\ b))(d(a\ c))))$.

Task 6.8. Assign a type to the object: $\#(B^3)$.

Solution.

$\#(B^3)$–*1.* $B^3 = BBB^2$.

As

$$\frac{(BB):\ ((a_1(b_1\ c_1))(a_1((d_1\ b_1)(d_1\ c_1))))\quad (B^2):(a_1(b_1\ c_1))}{(BBB^2):\ (a_1((d_1\ b_1)(d_1\ c_1)))},\ \ (F)$$

where $(a_1(b_1\ c_1)) = ((b\ c)((d(a\ b))(d(a\ c))))$, then $a_1 = (b\ c)$, $b_1 = (d(a\ b))$, $c_1 = (d(a\ c))$.
Let $d_1 = e$. Then $(BBB^2) : ((b\ c)((e(d(a\ b)))(e(d(a\ c)))))$.

Answer. $B^3 = BBB^2$ has the type $((b\ c)((e(d(a\ b)))(e(d(a\ c)))))$.

Task 6.9. Assign a type to the object: $\#(C^{[2]})$.

Solution.

$\#(C^{[2]})$–1. $C^{[2]} = BC(BC)$.

$\#(C^{[2]})$–2. Find the type $\#(BC)$.

$$\frac{(B):\ ((b_1\ c_1)((a_1\ b_1)(a_1\ c_1)))\ \ (C):(b_1\ c_1)}{(BC):\ ((a_1\ b_1)(a_1\ c_1))},\ \ (F)$$

where $(b_1\ c_1) = ((a(b\ c))(b(a\ c)))$, i.e. $b_1 = (b(c\ d))$, $c_1 = (c(b\ d))$. $a_1 = a$. Thus, $(BC):\ ((a(b(c\ d)))(a(c(b\ d))))$.

$\#(C^{[2]})$–3. $Exp\,1 = ((a_1(b_1(c_1\ d_1)))(a_1(c_1(b_1\ d_1))))$
$Exp\,2 = (a_1(b_1(c_1\ d_1)))$
$Exp3 = (a_1(c_1(b_1\ d_1)))$

$$\frac{(BC):\ Exp\,1 \qquad\qquad (BC):\ Exp\,2}{(BC(BC)):\ Exp\,3},\ \ (F)$$

where $(a_1(b_1(c_1\ d_1))) = ((a(b(c\ d)))(a(c(b\ d))))$, i.e. $a_1 = (a(b(c\ d)))$, $b_1 = a$, $c_1 = c$, $d_1 = (b\ d)$.

Thus, $(BC(BC)):\ ((a(b(c\ d)))(c(a(b\ d))))$.

Answer. $C^{[2]} = BC(BC)$ has the type $((a(b(c\ d)))(c(a(b\ d))))$.

Task 6.10. Assign a type to the object: $\#(C^{[3]})$.

Solution.

$\#(C^{[3]})$–1. $C^{[3]} = BC(BC^{[2]})$.

$\#(C^{[3]})$–2. $Exp\,1 = ((b_1\ c_1)((a_1\ b_1)(a_1\ c_1)))$
$Exp\,2 = (b_1\ c_1)$
$Exp\,3 = ((a_1\ b_1)(a_1\ c_1))$
$Exp\,4 = ((b_2\ c_2)((a_2\ b_2)(a_2\ c_2)))$
$Exp\,5 = (b_2\ c_2)$
$Exp\,6 = ((a_2\ b_2)(a_2\ c_2))$

As

$$\dfrac{\dfrac{(B):\ Exp\,1 \quad (C):\ Exp\,2}{(BC):\ Exp\,3}\,(F) \quad \dfrac{(B):\ Exp\,4 \quad (C^{[2]}):\ Exp\,5}{(BC^{[2]}):\ Exp\,6}\,(F)}{(BC(BC^{[2]})):\ (a_1\ c_1)}\,(F),$$

where $(b_1\ c_1) = ((a_3(b_3\ c_3))(b_3(a_3\ c_3)))$,
$(b_2\ c_2) = ((a(b(c\ d)))(c(a(b\ d))))$, $(a_1\ b_1) = ((a_2\ b_2)(a_2\ c_2))$, then $a_3 = a_2 = e$, $c_2 = (b_3\ c_3)$, $b_3 = c$, $c_3 = (a(b\,c))$.

Thus, $a_1 = (a_2\ b_2) = (e(a(b(c\ d))))$, $c_1 = (b_3(a_3\ c_3)) = (c(e(a(b\,d))))$, i.e. $(BC(BC^{[2]})):\ (a_1\ c_1)$.

Answer. $\#(C^{[3]}) = \#(BC(BC^{[2]})) = ((e(a(b(c\,d))))(c(e(a(b\,d)))))$.

Task 6.11. Assign a type to the object: $\#(C_{[2]})$.

Solution.

$\#(C_{[2]})$–1. $C_{[2]} = B^2CC$.

$\#(C_{[2]})$–2.
$$\begin{aligned} Exp\,1 &= ((b_1\ c_1)((d_1(a_1\ b_1))(d_1(a_1\ c_1)))) \\ Exp\,2 &= (b_1\ c_1) \\ Exp\,3 &= ((d_1(a_1\ b_1))(d_1(a_1\ c_1))) \\ Exp\,4 &= (d_1(a_1\ b_1)) \\ Exp\,5 &= (d_1(a_1\ c_1)) \end{aligned}$$

As

$$\dfrac{(B^2):\ Exp\,1 \qquad (C):\ Exp\,2}{\dfrac{(B^2C):\ Exp\,3 \qquad (C):\ Exp\,4}{(B^2CC):\ Exp\,5}\,(F)}\,,\ (F)$$

where

$$\begin{aligned} (d_1(a_1\ b_1)) &= ((a_2(b_2\ c_2))(b_2(a_2\ c_2))), \\ (b_1\ c_1) &= ((a(b\,c))(b(a\,c))), \end{aligned}$$

then

$$\begin{aligned} b_1 &= (a(b\,c)),\\ c_1 &= (b(a\,c)),\\ a_1 &= b_2,\\ d_1 &= (a_2(b_2\,c_2)),\\ b_1 &= (a_2\,c_2). \end{aligned}$$

The following is valid: $d_1 = (a(b_2(b\,c)))$, $a_1 = b_2$, $c_1 = (b(a\,c))$. Let $b_2 = d$. Then $(B^2CC) : (d_1(a_1\,c_1)) = ((a(d(b\,c)))(d(b(a\,c))))$.

Answer. $C_{[2]} = B^2CC$ has the type $((a(d(b\,c)))(d(b(a\,c))))$.

Task 6.12. Assign a type to the object: $\#(C_{[3]})$.

Solution.

$\#(C_{[3]})$–1. $C_{[3]} = B^2C_{[2]}C$.

$\#(C_{[3]})$–2.
$$\begin{aligned} Exp\,1 &= ((b_1\,c_1)((d_1(a_1\,b_1))(d_1\,(a_1\,c_1))))\\ Exp\,2 &= (b_1\,c_1)\\ Exp\,3 &= ((d_1(a_1\,b_1))(d_1\,(a_1\,c_1)))\\ Exp\,4 &= (d_1(a_1\,b_1))\\ Exp\,5 &= (d_1(a_1\,c_1)) \end{aligned}$$

$$\dfrac{\dfrac{(B^2) : Exp\,1 \qquad (C_{[2]}) : Exp\,2}{(B^2C_{[2]}) : Exp\,3}(F) \qquad (C) : Exp\,4}{(B^2C_{[2]}C) : Exp\,5}(F), \quad (F)$$

where $(b_1\,c_1) = ((a(b(c\,d)))(b(c(a\,d))))$, $(d_1\,(a_1\,b_1)) = ((a_2\,(b_2\,c_2))(b_2\,(a_2\,c_2)))$.

The following is valid: $d_1 = (a_2(b_2\,c_2)) = (a(b_2(b(c\,d))))$, $a_1 = b_2$, $c_1 = (b(c(a\,d)))$. Let $b_2 = e$. Substitute e in place of b_2, and in place of d_1, a_1, c_1 the corresponding expressions are to be substituted, resulting in type: $(B^2C_{[2]}C) : (d_1(a_1\,c_1))$.

Answer. $C_{[3]} = B^2C_{[2]}C$ has the type $((a(e(b(c\,d))))(e(b(c(a\,d)))))$.

Task 6.13. Assign a type to the object: $\#(\Phi)$.

Solution.

$\#\Phi{-}1$. $\Phi = B^2SB$.

$\#\Phi{-}2$.
$$\begin{aligned} Exp\,1 &= ((b_1\ c_1)((d_1(a_1\ b_1))(d_1(a_1\ c_1)))) \\ Exp\,2 &= (b_1\ c_1) \\ Exp\,3 &= ((d_1(a_1\ b_1))(d_1(a_1\ c_1))) \\ Exp\,4 &= (d_1(a_1\ b_1)) \\ Exp\,5 &= (d_1(a_1\ c_1)) \end{aligned}$$

As

$$\dfrac{(B^2):\ Exp\,1 \qquad\qquad (S):\ Exp\,2}{\dfrac{(B^2S):\ Exp\,3 \qquad (B):\ Exp\,4}{(B^2SB):\ Exp\,5}\ (F)},\ (F)$$

where

$$(b_1\ c_1) = ((a(b\ c))((a\ b)(a\ c))),$$
$$(d_1(a_1\ b_1)) = ((b_2\ c_2)((a_2\ b_2)(a_2\ c_2))),$$

then $d_1 = (b_2\ c_2) = (b_2(b\ c))$, $a_1 = (a_2\ b_2) = (a\ b_2)$, $c_1 = ((a\ b)(a\ c))$. Let $b_2 = d$; then

$$(B^2SB):\ (d_1(a_1\ c_1)) = ((d(b\ c))((a\ d)((a\ b)(a\ c)))).$$

Answer. $\Phi = B^2SB$ has the type $((d(b\ c))((a\ d)((a\ b)(a\ c))))$.

Task 6.14. Assign a type to the object: $\#(Y)$.

Solution.

$\#Y{-}1$. Use the equality $Y = WS(BWB)$.

$\#Y{-}2$. The type schemes are restricted by the following correspondences:

$$\dfrac{(W):\ ((a_1(a_1\ b_1))(a_1\ b_1)) \qquad (S):\ (a_1(a_1\ b_1))}{(WS):(a_1\ b_1)},\ (F)$$

As $\#(S) = (a(b\,c))((a\,b)(a\,c))$, then $a_1 = (a(b\,c))$, $a_1 = (a\,b)$, $b_1 = (a\,c)$. It follows, that $b = (b\,c)$, but it is impossible in a finite form. Hence, the assumption of existing a type $\#(WS)$ is invalid. In addition,

$$\frac{(B):\ ((b_2\,c_2)((a_2\,b_2)(a_2\,c_2)))\qquad (W):(b_2\,c_2)}{(BW):\ ((a_2\,b_2)(a_2\,c_2))}.\ (F)$$

As $\#(W) = (a_1(a_1\ b_1))(a_1\ b_1)$, then $b_2 = (a_1(a_1\ b_1))$, $c_2 = (a_1\ b_1)$. Next,

$$\frac{(BW):\ ((a_2\,b_2)(a_2\,c_2))\qquad (B):\ (a_2\,b_2)}{(BWB):\ (a_2\,c_2)},\ (F)$$

But $\#(B) = (a_3\,b_3)((c_3\,a_3)(c_3\,b_3))$, Hence, $a_2 = (a_3\,b_3)$, $b_2 = \underline{(c_3\,a_3)}(\underline{\underline{c_3}}\,b_3)$, $b_2 = \underline{a_1}(\underline{\underline{a_1}}\,b_1)$, $c_2 = (a_1\ b_1)$. It follows from these equalities, that, in particular, $a_1 = (c_3\,a_3)$ and at the same time $a_1 = c_3$, i.e. $c_3 = (c_3\,a_3)$, but this is impossible in a finite form. Therefore, the assumption of existing a type $\#(BWB)$ is invalid as well.

It follows, that it is impossible to assign type to the expression $WS(BWB)$. Hence, as $Y = WS(BWB)$, then it is impossible to assign type for a combinatory representation of Y.

$\#Y{-}3$. Nevertheless, consider the following reasonings. It is known, that $Yx = x(Yx)$, i.e. $\#(Yx) = \#(x(Yx))$.
Let $\#(x) = a$, $\#(Yx) = b$, then, according to the rule (F) : $\#(Y) = (a, b)$, because

$$\frac{(Y):\ (a,\ b)\qquad (x):\ a}{(Yx):\ b}\ (F)$$

Next, taking into account, that $\#(x(Yx)) = \#(Yx) = b$, we obtain $\#(x)$:

$$\frac{(x):\ (b,\ b)\qquad (Yx):\ b}{(x(Yx)):\ b}\ (F)$$

Therefore, $a = (b,\ b),\ (Y) :\ (a,\ b) = ((b,\ b), b)$.

Answer. Y has the type $((b,\ b), b)$, but $WS(BWB)$ has no type.

Task 6.15. Assign a type to the object: $\#(D)$.

Solution.

$\#D{-}1.\quad D = C_{[2]}I.$

$\#D{-}2.$

$$\frac{(C_{[2]}) :\ ((a(d(b\,c)))(d(b(a\,c))))\quad (I) :\ (a(d(b\,c)))}{(C_{[2]}I) :\ (d(b(a\,c)))},\quad (F)$$

where $(I) :\ (a_1,\ a_1)$, i.e. $a = (d(b\,c))$.

Thus, $\#(C_{[2]}I) = (d(b((d(b\,c))c)))$.

Answer. $D = C_{[2]}I$ has the type: $(a, (b, ((a, (b,\ c)), c)))$.

Task 6.16. Assign a type to the object: $\#(C)$.

Solution.

$\#C{-}1.\quad C = S(BBS)(KK).$

$\#C{-}2.\quad (S) :\ (a\,b\,c)((a\,b)(a\,c)),\ (B) :\ (b\,c)((a\,b)(a\,c)),\ (K) :\ (a(b\,a)).$

$\#C{-}3.$

$$\frac{(B) :\ (b_2\ c_2)((a_2\ b_2)(a_2\ c_2))\quad (B) :\ (b_2\ c_2)}{(\mathbf{BB}) :\ (a_2\ b_2)(a_2\ c_2)}$$

$$\frac{(\mathbf{BB}) :\ (a_2\ b_2)(a_2\ c_2)\quad (S) :\ (a_2\ b_2)}{(\mathsf{BBS}) :\ (a_2\ c_2)}$$

$$\frac{(S) :\ (a_3(b_3\ c_3))((a_3\ b_3)(a_3\ c_3))\quad (\mathsf{BBS}) :\ (a_2\ c_2)}{\mathcal{S}(\mathcal{BBS}) :\ (a_3\ b_3)(a_3\ c_3)}$$

$$\frac{(K):\ (a_4(b_4\ a_4))\ \ (K):\ a_4}{(\mathcal{K}\mathcal{K}):\ (b_4\ a_4)}$$

$$\frac{\mathcal{S}(\mathcal{B}\mathcal{B}\mathcal{S}):\ (a_3\ b_3)(a_3\ c_3)\quad (\mathcal{K}\mathcal{K}):\ (b_4\ a_4)}{S(BBS)(KK):\ (\underline{a_3}\ \underline{c_3}\ )},$$

where:

$(a_3\ b_3) = (b_4\ a_4)$,
$(b_2\ c_2) = ((b_6\ c_6), ((a_6\ b_6), (a_6\ c_6)))$, scheme for B is taken

$a_4 = (a_5(b_5\ a_5))$, scheme for K is taken

$(a_2\ c_2) = (a_3(b_3\ c_3))$,
$(a_2\ b_2) = (a\ b\ c)(a\ b)(a\ c)$. scheme for S is taken

The following is derived:

$$\begin{cases} \underline{a_3} = a_2 = \underline{(a\ b\ c)} = b_4, \\ c_2 = (b_3\ c_3) = (a_6\ b_6)(a_6\ c_6), \\ b_2 = (a\ b)(a\ c) = (b_6\ c_6), \\ b_3 = a_4 = (a_5\ b_5\ a_5). \end{cases}$$

Next,

$c_6 = (a\ c)$,
$b_6 = (a\ b)$,
$b_3 = (a_6\ b_6) = (a_6\ a\ b) = (b\ a\ b)$, because $b_3 = (a_5\ b_5\ a_5)$.

Thus, $\underline{c_3} = (a_6\ c_6) = (b\ c_6) = \underline{(b\ a\ c)}$. The only thing is left to write the type $(a_3,\ c_3)$.

Answer. $C = S(BBS)(KK)$ has the type: $((a, (b,\ c)), (b, (a,\ c)))$.

Chapter 7

Basis I, K, S

In constructing a combinatory logic, one of the generic tasks was to declare a minimum amount of primitive objects, which are atomic in their origin, and to declare the modes of their combining, which give the sufficiently rich mathematical means. These primitive objects should have the same behavior as the *constant* functions, and, from a point of view of computer science give the primitive core system for programming, which should grow by "self-evolving". This means, that more comprehensive objects are constructed using the more primitive ones, and then they are joined to the system and, in turn, can be used as the elements in other constructions.

Let fix an attention at the most primitive programming system, which consists of just three instructions: I, K, S. A new object can be generated by purely routine applying of algorithm of disassembling into a basis, that is quite similar to compiling.

7.1 Theoretical background

It will be shown, that an object, which is denoted by λ-term (source representation), can be represented by the combinatory term (target representation). A procedure of transforming the object from source

to target representation can be significantly simplified in case a predefined set of combinators is in use. To achieve this goal the set I, K, S, will be fixed, which, as known, conforms the property of being a basis.

7.2 Tasks

Task 7.1. Express the term$\lambda x.P$ via combinators $I,\ K,\ S$.

Task formulation. Let the definition of term $\lambda x.P$ be given by induction on constructing P:

$$\begin{array}{lllll} (1) & \lambda x.x & = & I, \\ (2) & \lambda x.P & = & KP, \text{ if } x \text{ does not belong to } FV(P), \\ (3) & \lambda x.PQ & = & S(\lambda x.P)(\lambda x.Q) \text{ otherwise.} \end{array}$$

Exclude all the variables from the following λ-expressions:

$$1.\ \lambda xy.yx;\ 2.\ \lambda fx.xx;\ 3.\ f = \lambda x.B(f(Ax)).$$

Solution.

$$\begin{array}{llll} P{-}1. & \lambda xy.yx & \overset{def}{=} & \lambda x.(\lambda y.yx)) \\ & & \overset{(3)}{=} & \lambda x.(S(\lambda y.y)\lambda y.x)) \\ & & \overset{(1),\ (2)}{=} & \lambda x.SI(Kx) \\ & & \overset{(3)}{=} & S(\lambda x.SI)(\lambda x.Kx) \\ & & \overset{(2)}{=} & S(K(SI))(S(\lambda x.K)(\lambda x.x)) \\ & & \overset{(1),\ (2)}{=} & S(K(SI))(S(KK)I) \end{array}$$

$$
\begin{array}{lll}
P{-}2. \quad \lambda fx.fxx & \stackrel{def}{=} & \lambda f.(\lambda x.fxx) \\
& \stackrel{(3)}{=} & \lambda f.(S(\lambda x.fx)(\lambda x.x)) \\
& \stackrel{(1),\,(3)}{=} & \lambda f.S(S(\lambda x.f)(\lambda x.x))I \\
& \stackrel{(1),\,(2)}{=} & \lambda f.S(S(Kf)I)I \\
& \stackrel{(3)}{=} & S(\lambda f.S(S(Kf)I))(\lambda f.I) \\
& \stackrel{(2),\,(3)}{=} & S(S(\lambda f.S)(\lambda f.S(Kf)I))(KI) \\
& \stackrel{(2),\,(3)}{=} & S(S(KS)(S(\lambda f.S(Kf))(\lambda f.I)))(KI) \\
& \stackrel{(2),\,(3)}{=} & S(S(KS)(S(S(\lambda f.S)(\lambda f.Kf)))(KI)))(KI) \\
& \stackrel{(2),\,(3)}{=} & S(S(KS)(S(S(KS)(S(\lambda f.K)(\lambda f.f))(KI)))(KI)) \\
& \stackrel{(1),\,(2)}{=} & S(S(KS)(S(S(KS)(S(KK)I)(KI)))(KI))
\end{array}
$$

$$
\begin{array}{lll}
P{-}3. \quad f & \stackrel{def}{=} & \lambda x.b(f(ax)) \\
& \stackrel{(3)}{=} & S(\lambda x.b)(\lambda x.f(ax)) \\
& \stackrel{(2),\,(3)}{=} & S(Kb)(S(\lambda x.f)(\lambda x.ax)) \\
& \stackrel{(2),\,(3)}{=} & S(Kb)(S(Kf)(S(\lambda x.a)(\lambda x.x))) \\
& \stackrel{(2),\,(1)}{=} & S(Kb)(S(Kf)(S(Ka)I))
\end{array}
$$

Exercises

Exercise 7.1. Express the combinator $W\,mult$ of second power of function in terms of combinators I, K, S.

Exercise 7.2. For the combinator Φ with a characteristic

$$\Phi fabx = f(ax)(bx)$$

do the following:

1° write the predicate

$$1 < x < 5$$

without a variable.

Hint. To get this, introduce a construction, which conforms $\Phi\ and\ (greater\ 1)\ (less\ 5)$;

2° express the combinator

$$\Phi\ and\ (greater\ 1)\ (less\ 5)$$

in terms of combinators I, K, S.

Exercise 7.3. Express a combinator, representing the function of sum of sines of two numbers, in terms of combinators I, K, S.

Hint. It is possible, for instance, to use the combinator Φ and write

$$\Phi\ plus\ sin\ sin.$$

Try to transform this expression, using the combinators for elimination of duplicate occurrence of 'sin'. E.g.,

$$\Phi\ plus\ sin\ sin = W(\Phi\ plus)\ sin = \ldots$$

etc.

Chapter 8

Basis I, B, C, S

As was established, the basis I, K, S is not a unique, and a set of combinators I, B, C, S has a property of being a basis as well. Compiling (disassembling) of an object using this basis gives a solution of a task to generate the object with given properties. Obviously, a selection of basis is of free choice depending of some criteria.

8.1 Theoretical background

As could be assured, representing terms by compiling them into basis I, K, S, there is a regular or even mechanical way to transit from one object representation to another. Increasing the number of distinct combinators leads to reducing a number of steps in applying the compiling algorithm. Of course, as was awaited, not all the sets of combinators consist of a pairwise independent combinators. In particular, combinator I can be expressed in terms of K and S, hence, strictly speaking, this combinator is excessive and not necessary. Nevertheless, its usage gives some technical advantages.

There are other sets of combinators, using which one can get a representation of any well formed term. Such sets of combinators are considered as basic, or, as accepted to speak, conform a *basis*. As

previously, combinator is an object which if constructed from basic combinators using application. Hence, the combinatory basis is not to be obviously unique, and the multiple representations of the same object are awaited. Depending on the aims, one or another representation can be selected[1].

8.2 A property of being basic

A huge variety of disparate combinators can be generated, but, as it appears, these combinators are not independent. Part of them can be determined via the set of combinators called *basic*. Combinator can be thought as an object being constructed from the basic combinators by its applications.

Consider two basic systems of combinators C, W, B, K and S, K:

$$\begin{array}{l|l} Cxyz = xyz, & Sxyz = xz(yz), \\ Wxy = xyy, & Kxy = x. \\ Bxyz = x(yz), & \\ Kxy = x; & \end{array}$$

For any syntactic object V, constructed from distinct variables $x_1,\ldots,$ x_n by its applications, one can determine a combinator X, composed from basic combinators, such that:

$$X\ x_1, \ldots, x_n = V.$$

For instance, if $V = xz(yz)$, then there is the combinator X constructed of the instances from basic system S, K, such that $X\ xyz = xz(yz)$. It is not difficult to conclude, that this is the combinator S, i.e. $X = S$.

[1] For example, besides the bases I, K, S and I, B, C, S, used here, the other bases can be introduced. In particular, the set C, W, B, K has a property of being basic as well.

In general, a generation of combinator X is given by the following actions:

1) using B to eliminate the parentheses in V;

2) using C to re-order the variables;

3) using W to eliminate multiple occurrences of the variables;

4) using K to bring in the variables which are not present in V.

Example 8.1. Let to construe, in the first of the basic systems, a combinator X with the property

$$ac(bc) = X\,abc:$$

$$\begin{aligned}
\underbrace{(ac)}_{x}(\underbrace{b}_{y}\ \underbrace{c}_{z}) &= \underbrace{B}_{x}(\underbrace{a}_{y}\ \underbrace{c}_{z})bc \\
&\overset{B}{=} \underbrace{(BBa)}_{x}\ \underbrace{c}_{z}\ \underbrace{b}_{y}\ c \\
&\overset{C}{=} \underbrace{(C(BBa)b)}_{x}\ \underbrace{c}_{y}\ \underbrace{c}_{y} \\
&\overset{W}{=} \underbrace{W}_{x}(\underbrace{C(BBa)}_{y}\ \underbrace{b}\ )_z c \\
&\overset{B}{=} \underbrace{(BW)}_{x}(\underbrace{C}_{y}\ \underbrace{(BBa)}_{z})bc \\
&\overset{B}{=} \underbrace{(B(BW)C)}_{x}(\underbrace{(BB)}_{y}\ \underbrace{a}_{z}\ )bc \\
&\overset{B}{=} B(B(BW)C)(BB)abc.
\end{aligned}$$

In this example, under the objects, to which the known *scheme* can be applied, the variables have been written — in the same order as in the corresponding combinatory characteristic. Thus,

$$X \;=\; B(B(BW)C)(BB).$$

8.3 Elementary examples

Consider examples of disassembling the object into basis. Let the source terms be the same as in the occasion of disassembling into basis I, K, S. The resulting expressions can be compared to conclude, what of the bases could be preferred[2].

Task 8.1. Represent the term $M = \lambda x.PQ$, using the only combinators I, B, C, S.

Task formulation. Let the definition of such a term M that the variable x occurs in a set of free in (PQ) variables, i.e. $x \in FV(PQ)$, be given by induction on constructing M (here: '$\in$' means 'belongs', and '$\notin$' means 'does not belong'):

$$
\begin{array}{lll}
(1) & \lambda x.x & = I, \\
(2) & \lambda x.PQ = \left\{ \begin{array}{lll}
(a) & BP(\lambda x.Q), & \text{if } x \notin FV(P) \text{ and } x \in FV(Q), \\
(b) & C(\lambda x.P)Q, & \text{if } x \in FV(P) \text{ and } x \notin FV(Q), \\
(c) & S(\lambda x.P)(\lambda x.Q), & \text{if } x \in FV(P) \text{ and } x \in FV(Q).
\end{array} \right.
\end{array}
$$

Exclude all the variables from the following λ-expressions:
1. $\lambda xy.yx$; 2. $\lambda fx.fxx$.

Solution.

[2] The solving of the task of disassembling an object into the basis can be viewed differently. As any combinator is a *notion* and even a *concept* in a mathematical sense, then a source object is 'under investigation', or 'under knowing', basis is a 'system of known concepts', and a procedure of disassembling into basis is 'knowledge representation' of the source object in terms of already known concepts. Similar reasons in its essence are used in applications of object-oriented approach.

$$
\begin{array}{lrl}
M{-}1. & \lambda xy.yx & = \lambda x.(\lambda y.yx) \\
 & & \overset{(2)(b)}{=} \lambda x.(C(\lambda y.y)x) \\
 & & \overset{(1)}{=} \lambda x.CIx \\
 & & \overset{(2)(a)}{=} B(CI)(\lambda x.x) \\
 & & \overset{(1)}{=} B(CI)I.
\end{array}
$$

Checking. $B(CI)Ixy = CI(Ix)y = Iy(Ix) = Iyx = yx.$

$$
\begin{array}{lrl}
M{-}2. & \lambda fx.fxx & = \lambda f.(\lambda x.fxx) \\
 & & \overset{(2)(c)}{=} \lambda f.S(\lambda x.fx)(\lambda x.x) \\
 & & \overset{(1),\,(2)(a)}{=} \lambda f.S(Bf(\lambda x.x))I \\
 & & \overset{(1)}{=} \lambda f.S(BfI)) \\
 & & \overset{(2)(b)}{=} C(\lambda f.S(BfI))I \\
 & & \overset{(2)(a)}{=} C(BS(\lambda f.BfI))I \\
 & & \overset{(2)(b)}{=} C(BS(C(\lambda f.Bf)I))I \\
 & & \overset{(2)(a)}{=} C(BS(C(BB(\lambda f.f))I))I \\
 & & \overset{(1)}{=} C(BS(C(BBI)I))I.
\end{array}
$$

Exercises

Exercise 8.1. Prove, that a set of combinators C, W, B, K has a property of being *basic*, i.e. is the basis.

Hint. Use a definition of basis in a general form (see. [2], p. 172):

Definition 8.1 *(basis).* **(i)** Let $\mathcal{X}$ be a subset of all the λ-terms Λ, $\mathcal{X} \subset \Lambda$. Denote by $\mathcal{X}^+$ a set of terms, *generated by* $\mathcal{X}$. Set $\mathcal{X}^+$ is the least set $\mathcal{Y}$, such that:

1) $\mathcal{X} \subseteq \mathcal{Y}$,

2) if terms $M, N \in \mathcal{Y}$, then their application $(M\,N) \in Y$.

(ii) Consider a subset of λ-terms $\mathcal{A} \subseteq \Lambda$. A set of terms $\mathcal{X} \subseteq \Lambda$ is called *basis* of $\mathcal{A}$, if

$$(\forall M \in \mathcal{A})(\exists N \in \mathcal{X}^+).N = M.$$

(iii) Set $\mathcal{X}$ is called *basis*, if $\mathcal{X}$ is a basis of the set of closed terms.

Exercise 8.2. Represent the combinator $W\,mult$ of the second power of a function in terms of combinators I, B, C, S.

Exercise 8.3. For the combinator Φ with the a characteristic

$$\Phi fabx = f(ax)(bx),$$

do the following:

1° write the predicate $1 < x < 5$ without a variable.

Hint. To achieve this goal, bring in a suitable construction $\Phi\ and\ (greater\ 1)\ (less\ 5)$;

2° represent combinator $\Phi\ and\ (greater\ 1)\ (less\ 5)$ in terms of combinators I, B, C, S.

Exercise 8.4. Express a combinator, representing the function of sum of sines of two numbers, in terms of combinators $I, B, C\ S$.

Hint. It is possible, for instance, to use combinator Φ and write

$$\Phi\ plus\ sin\ sin.$$

Try to transform this expression, using the combinators for elimination of twofold occurrence of 'sin'. E.g.,

$$\Phi\ plus\ sin\ sin = W(\Phi\ plus)\ sin = \ldots$$

etc.

Chapter 9

Applications of fixed point combinator Y

The recursive definitions of the objects are considered in this chapter. Using a fundamental for functional programming fixed point theorem, the recursive definitions are succeeded in reducing to the usual equational form.

9.1 Fixed point theorem

The tasks, considered in this chapter, are composed as a small self-contained study. Aim is to establish the relationships of the programming language constructions with the notions of calculus of λ-conversion.

A characteristic equality for the function Y is as follows:

$$Y f = f(Y f).$$

Theorem 9.1 (fixed point). Self referencing definition of the function f:

$$f = E f$$

where f has no free occurrence in E, $f \notin E$, can be found as:

$$f = Y\, E.$$

Proof. Evidently, if in the definition:

$$\underbrace{f}_{=Y\,E} = E\ \underbrace{f}_{=Y\,E}$$

replace f by $(Y\ E)$, then obtain $Y\ E = E(Y\ E)$. □

9.2 Elements of recursive computations

A function is assumed as recursive, if its defining expression contains, at least, one reference to the definable function itself.

Consider a recursive definition of the factorial function:

$$FAC\ =\ (\lambda n.IF(=\ n\,0)\,1\,(\times n\,FAC(-\,n\,1)))$$

When this definition is applied directly, then the following transformations occur:

$$\begin{array}{lcl}
FAC & = & (\lambda n.IF(=\ n\,0)1 \\
 & & \quad (\times n\,FAC(-\,n\,1))) = \\
FAC & = & (\lambda n.IF(=\ n\,0)1 \\
 & & \quad (\times n\,(\lambda n.IF(=\ n\,0)1 \\
 & & \quad\quad (\times n\,FAC(-\,n\,1)))(-\,n\,1))) = \\
FAC & = & (\lambda n.IF(=\ n\,0)1 \\
 & & \quad (\times n\,(\lambda n.IF(=\ n\,0)1 \\
 & & \quad\quad (\times n(\lambda n.IF(=\ n\,0)1 \\
 & & \quad\quad\quad (\times n\,FAC(-\,n\,1)))(-\,n\,1)))(-\,n\,1))) = \\
\ldots & & \ldots.
\end{array}$$

It is not difficult to view, the a chain of these transformations never completes. In the notations above the name FAC is assigned to the

λ-term. This notion is natural and suitable, but does not conform to a syntax of calculus, because there is no function referencing in the λ-calculus. In a connection with this, an expression is to be translated in the language of λ-calculus, i.e. to express a recursion in its pure form (without self referencing). As it happened, this translation is possible, and without violation a framework of the combinatory logic.

9.3 Using the combinator Y

As a main sample, whose evaluation exemplifies the principal effects of computations with a fixed point, use the factorial function FAC. The notion of this function FAC in an abbreviated form is as follows:

$$FAC = \lambda n.(\ldots FAC \ldots). \tag{9.1}$$

Using λ-abstraction, obtain:

$$\lambda fac.FAC\ fac = (\lambda fac.(\lambda n.(\ldots fac \ldots)))FAC. \tag{9.2}$$

By rule (η), conclude that:

$$\lambda fac.FAC\ fac = FAC,$$

and the definition (9.2) can be written as follows:

$$FAC = (\lambda fac.(\lambda n.(\ldots fac \ldots)))FAC. \tag{9.3}$$

Assuming $H = \lambda fac.(\lambda n.(\ldots fac \ldots))$, obtain:

$$FAC = H\ FAC. \tag{9.4}$$

The definition of H is a usual λ-abstraction, which does not use a recursion. A recursion is expressed only in a form of equality (9.4). The definition (9.4) is similar to a mathematical equation. E.g., to solve the equation $(x^2 - 2) = x$ means to find out the values of x,

which satisfy this equation: $(x = -1, x = 2)$. Similarly, to solve (9.4), means to find out the λ-abstraction for FAC, which satisfies (9.4). The equation $FAC = H\ FAC$ expresses the fact, that when the function H is applied to the argument FAC, its results in FAC again. That is why FAC is called a fixed point of the function H.

Example 9.1. The numbers 0 and 1 are the fixed points of the function $f = \lambda x. \times x\ x$, i.e. $f\,0 = 0$ and $f\,1 = 1$. Actually,

$$\begin{array}{rcll} f\,0 & = & (\lambda x. \times x\ x)0 = & (\beta) \\ & = & \times\ 0\ 0 = 0, & \\ f\,1 & = & (\lambda x. \times x\ x)1 = & (\beta) \\ & = & \times\ 1\ 1 = 1. & \end{array}$$

Thus, it is needed to find out a fixed point of the function H. Evidently, this point depends only of H. Introduce the function Y, which conforms the following scheme:

> getting input function as its argument, the function Y generates as an output the fixed point of this function.

It means, that we obtain $YH = H(YH)$, where Y is a fixed point combinator. If we find out such Y, then we obtain a solution of the equation (9.4):

$$FAC = YH.$$

During a performance of this solution a rather general method is used. It is based on the principal in a theory of recursive computations *fixed point theorem* (see Theorem 9.1). In fact, it was given its particular "proof" for the function FAC. The obtained solution gives the *non-recursive* definition of FAC. An essence of the fixed point theorem is namely in a assuring the transition from recursive in its notations definitions to non-recursive in its notations definitions. In the last case an effect of cycling in the definition (and corresponding computation) is hidden by the combinator Y.

Example 9.2. To get assured, that thus defined function FAC properly works, let's apply it to some argument, e.g., to 1:

$$
\begin{aligned}
&FAC\ 1 = \underline{Y\ H}\ 1 = \overline{H(Y\ H)}1 && (FAC), (Y)\\
&= (\lambda fac.\lambda n.IF(=\ n\ 0)1(\times n(fac(-\ n\ 1))))(Y\ H)1 && (H)\\
&= (\lambda n.IF(=\ n\ 0)1(\times n((Y\ H)(-\ n\ 1))))1 && (\beta)\\
&= IF(=\ 1\ 0)1(\times 1((Y\ H)(-\ 1\ 1))) && (\beta)\\
&= \times 1(\overline{(Y\ H)}0)\ =\ \times 1(\overline{H(Y\ H)}0) && (Y)\\
&= \times 1((\overline{\lambda fac.\lambda n.IF(=\ n\ 0)1(\times n(fac(-\ n\ 1))))}(Y\ H)0) && (H)\\
&= \times 1((\lambda n.IF(=\ n\ 0)1(\times n((Y\ H)(-\ n\ 1))))0) && (\beta)\\
&= \times 1(IF(=\ 0\ 0)1(\times 0(Y\ H(-\ 0\ 1)))) && (\beta)\\
&= \times 1\ 1\\
&= 1.
\end{aligned}
$$

9.4 Evaluation of a function

Consider the examples of definitions for most often used in practice of programming functions. Note, that for simplicity, the functions, which use lists as their arguments, are selected out. As usually, the *list* is understood as a finite sequence.

Task 9.1. Using the fixed point function Y, express the definitions of

the following functions:

$$
\begin{array}{lll}
2) & sum & = \lambda x.if\ null\ x \\
 & & \quad then\ 0 \\
 & & \quad else\ (car\ x) + sum(cdr\ x), \\
3) & product & = \lambda x.if\ null\ x \\
 & & \quad then\ 1 \\
 & & \quad else\ (car\ x)\ \times\ product(cdr\ x), \\
4) & append & = \lambda x.\lambda y.if\ null\ x \\
 & & \quad then\ y \\
 & & \quad else\ (list((car\ x)(append(cdr\ x)y)), \\
5) & concat & = \lambda x.if\ null\ x \\
 & & \quad then\ () \\
 & & \quad else\ append(car\ x)(concat(cdr\ x)), \\
6) & map & = \lambda f\lambda x.if\ null\ x \\
 & & \quad then\ () \\
 & & \quad else\ list((f(car\ x))(map\ f(cdr\ x))).
\end{array}
$$

$$
\begin{array}{l}
length(a_1, a_2, a_3) = 3; \\
sum(1, 2, 3, 4) = 10; \\
product(1, 2, 3, 4) = 24; \\
append(1, 2)(3, 4, 5) = (1, 2, 3, 4, 5); \\
concat((1, 2), (3, 4), ()) = (1, 2, 3, 4); \\
map\ square\ (1, 2, 3, 4) = (1, 4, 9, 16).
\end{array}
$$

For examples of "calls" of each of the functions perform its checking.

Starting up to work with lists, it is possible to estimate the abilities of the programming system Lisp[1]. This system makes no distinction between "programs" and "data": both of them are the objects

[1] In its ground Lisp has, practically, all the abilities of untyped lambda-calculus. Depending on a dialect in use, the outer notation of the objects can be varied. Pay attention one more, that this language has *no operators*. The only kind of its objects in use are the *functions*. The implemented mechanism of recursion from a mathematical point of view illustrates an effect of the fixed point computations. Usually, a language is supplied with the additional non-recursive means of organizing the cycled, or iterative computations.

on equal rights. A great interest to the programming system Lisp is completely justified. Having clear and short mathematical foundations, this programming system overcomes a barrier between a practical program encoding of a task and its mathematical understanding.

Solution. As an example, give the corresponding computations for the function *length* (getting the length of a list) and *map* (the functional, "distributing" an action of function-argument along the list). During the evaluation of these functions, the main features of recursive computations over lists reveal.

length−1. For the function *length* its initial definition

$$\begin{aligned} length \;=\; & \lambda x.if\ null\ x \\ & then\ 0 \\ & else\ 1 + length(cdr\ x), \end{aligned}$$

is to be re-written as:

$$\begin{aligned} length \;=\; & (\lambda f.\lambda x.if\ null\ x \\ & then\ 0 \\ & else\ 1 + f(cdr\ x))length. \end{aligned}$$

length−2. It follows from this, that

$$\begin{aligned} length \;=\; & Y(\lambda f.\lambda x.if\ null\ x \\ & then\ 0 \\ & else\ 1 + f(cdr\ x)). \end{aligned}$$

Thus, an awaited combinatory characteristic is derived.

length−3. Make a checking of the definition for list of length

2, i.e. let $x = (a_1, a_2)$:

$$\begin{aligned}
&length(a_1, a_2) = Y(\lambda f \lambda x.if\ null\ x \\
&\qquad\qquad\qquad\qquad\qquad then\ 0 \\
&\qquad\qquad\qquad\qquad\qquad else\ 1 + f(cdr\ x))(a_1, a_2) \\
&= (\lambda f \lambda x.if\ null\ x \\
&\qquad\quad then\ 0 \\
&\qquad\quad else\ 1 + f(cdr\ x))(Y(\ldots))(a_1, a_2) \\
&= if\ null\ (a_1, a_2)\ then\ 0\ else\ 1 + (Y(\ldots))(cdr\ (a_1, a_2)) \\
&= 1 + (Y(\ldots))(cdr\ (a_1, a_2)) \\
&= 1 + (\lambda f \lambda x.if\ null\ x\ then\ 0 \\
&\qquad\qquad\qquad\qquad\qquad else\ 1 + f(cdr\ x))(Y(\ldots))(a_2) \\
&= 1 + if\ null\ (a_2)\ then\ 0\ else\ 1 + (Y(\ldots))\ nil \\
&= 1 + (1 + (Y(\ldots))\ nil) \\
&= 1 + (1 + (\lambda f \lambda x.if\ null\ x\ then\ 0 \\
&\qquad\qquad\qquad\qquad\qquad else\ 1 + f(cdr\ x))(Y(\ldots))\ nil) \\
&= 1 + (1 + (0)) = 2.
\end{aligned}$$

map–1. For the function *map* its initial definition

$$\begin{aligned}
map \quad = \quad \lambda f.\lambda x. \quad & if\ null\ x \\
& then\ () \\
& else\ (f(car\ x)) : (map\ f(cdr\ x))
\end{aligned}$$

is to be re-written as:

$$\begin{aligned}
map = (\lambda m.\lambda f.\lambda x.&if\ null\ x \\
&then\ () \\
&else\ (f(car\ x)) : (m\ f(cdr\ x)))\ \ map.
\end{aligned}$$

It follows from this, that

$$\begin{aligned}
map \quad = \quad Y(\lambda m.\lambda f.\lambda x. \quad & if\ null\ x \\
& then\ () \\
& else\ (f(car\ x)) : (m\ f(cdr\ x))).
\end{aligned}$$

Checking. Checking for $f = square,\ x = (2,3)$ (outline):

$$
\begin{aligned}
&map\ square\ (2,3) = \\
&\quad = (\lambda m \lambda f \lambda x.if\ null\ x \\
&\qquad\quad then\ () \\
&\qquad\quad else\ (f(car\ x)) : (m\ f(cdr\ x)))(Y(\ldots))square\ (2,3) \\
&\quad = (square\ 2) : ((Y(\ldots))\ square\ (3)) \\
&\quad = (square\ 2) : ((\lambda m.\lambda f.\lambda x.\ldots)(Y(\ldots))\ square\ (3)) \\
&\quad = (square\ 2) : ((square\ 3) : ((Y(\ldots))\ square\ ())) \\
&\quad = (square\ 2) : ((square\ 3) : ()) \\
&\quad = (4,9).
\end{aligned}
$$

In this case the symbol ':' is used to denote an infix form of the function $list$, therefore, as a notational agreement, is accepted, that $x : (y : (z : ())) \equiv \langle x, y, z \rangle \equiv (x, y, z)$.

Exercises

Exercise 9.1. Write a function, implementing the algorithm of disassembling into basis I, K and S.

Hint. As known, all the λ-terms can be avoided by replacing with the constructions, formed of combinators I, K and S, and combinator I can be replaced by the construction SKK.

The function, forming a combination, can be denoted, e.g., by `combine`. Now, for instance, the function, which eliminates the identifier x from an expression E, can be determined in a form of self referencing definition. Next, using a fixed point theorem, this definition can be transformed to a standard form by eliminating an occurrence of `extract` from the right part of the definition:

```
def rec extract x E
        if identifier E
        then if E = x
             then 'I'
             else  combine 'K' E
        else if lambdaexp E
             then extract x (extract (bv E)(body E))
             else let F = extract x (rator E)
                  let A = extract x (rand E)
                      combine 'S' (combine F A)
```

Exercise 9.2. Write the program, which transforms a λ-term into combinatory form, checking the occurrence of a variable in the operator or operand:

1) if the operator does not depend on x, then combinator B is introduced:
$$Bfgx = (f)(gx);$$

2) if the operand does not depend on x, then combinator C is introduced:
$$Cfgx = (fx)g;$$

3) if neither operator, nor operand does not depend on x, then the combinator is not introduced:
$$fg = (f)(g);$$

4) if $g = I$, then
$$\begin{aligned} SfIx &= (fx)(Ix) = fxx = Wfx, \\ BfIx &= (f)(Ix) = fx; \end{aligned}$$

5) if $f = K$, then
$$\begin{aligned} SKgx &= (Kx)(gx) = Ix, \\ CKgx &= (Kx)(g) = Ix. \end{aligned}$$

Hint. See, for instance, [5], p. 45-46.

Chapter 10

Function $list1$

This chapter describes the abilities of constructing a parameterized function-object. Assigning the particular values to the arguments – and, as this particular value, can be the functions, – a whole family of definitions of particular functions can be derived.

10.1 Theoretical background

The function $list1$ is defined as follows:

$$\begin{aligned} list1\ a\ g\ f\ x \quad = \quad & if\ (null\ x) \\ & then\ a \\ & else\ g(f(car\ x))(list1\ a\ g\ f(cdr\ x)). \end{aligned}$$

This is a sample of a function of rather general kind, from which, given a particular value of parameters, the whole sequence of "simpler" functions can be generated. It could be noted, that many of functions, acting over the lists, have some "shared part". A question is, if is it possible for the class of functions over lists, to determine such a generalized function, from which, by different choice of parameters, the particular representatives of the class of functions can

be generated. As one of such generalized functions, the function $list1$ is presented below.

The main construction in use is a *list*, which can be either empty or non-empty. In the last case it has a "head" and "tail", which, in turn, can be the lists. The following operations can be performed over lists:

$$\begin{array}{lcl} null & : & \text{list} \to \text{boolean}, \\ car & : & \text{non-empty list} \to (\text{list} + \text{atom}), \\ cdr & : & \text{non-empty list} \to \text{list}, \\ list & : & (\text{atom} + \text{list}) \to (\text{list} \to \text{list}). \end{array}$$

These operation are interrelated as follows:

$$\begin{array}{rcl} null\ () & = & true, \\ null\ (list\ x\ y) & = & false, \\ car\ (list\ x\ y) & = & x, \\ cdr\ (list\ x\ y) & = & y, \\ list\ (car\ z)(cdr\ z) & = & z. \end{array}$$

In addition, use an abbreviation:

$$list\ x\ y = x : y,$$

and, thus, for $n \geq 2$ apply this agreements to the construction below:

$$\langle x_1,\ x_2,\ x_3, \ldots, x_n \rangle = x_1 : (x_2 : (x_3 : (\ldots\ x_n) : \ldots () \ldots)).$$

10.2 Tasks

Task 10.1. Study the properties of the function

$$\begin{array}{rcl} list1\ a\ g\ f\ x & = & if\ (null\ x) \\ & & then\ a \\ & & else\ g(f(car\ x))(list1\ a\ g\ f(cdr\ x)). \end{array}$$

Using the following definitions:

$$Ix = x,\ Kxy = x,\ postfix\ x\ y = append\ y\ (ux),$$

where (ux) is a list, which contains a singular element x, express the following functions:

(a) $length$, $sumsquares$, $reverse$, $identity$;

(b) sum, $product$, $append$, $concat$, map.

Task formulation. Use the following definitions of the functions:

(a) $ux = x : (),$

$length\ x = if\ null\ x\ then\ 0\ else\ 1 + length\ (cdr\ x),$

$sumsquares\ x = if\ null\ x$
$\quad then\ ()$
$\quad else(square(car\ x))\ +\ sumsquares(cdr\ x),$

$reverse\ x = if\ null\ x\ then\ ()$
$\quad else\ append(reverse(cdr\ x))(ux),$

$identity\ x = x,$

$square\ x = if\ null\ x\ then\ 0\ else\ x\ \times\ x;$

(b) $sum\ x = if\ null\ x$
$\quad then\ 0$
$\quad else\ (car\ x)\ +\ sum(cdr\ x),$

$product\ x = if\ null\ x$
$\quad then\ 1$
$\quad else\ (car\ x)\ \times\ product(cdr\ x),$

$append\ x\ y = if\ null\ x$
$\quad then\ y$
$\quad else\ list\ (car\ x)(append(cdr\ x)y),$

$concat\ x = if\ null\ x$
$\quad then\ ()$
$\quad else\ append(car\ x)(concat(cdr\ x)),$

$map\ f\ x = if\ null\ x$
$\quad then\ ()$
$\quad else\ (f(car\ x)) : (map\ f(cdr\ x)).$

Perform steps of the algorithms for the following samples:

$$\begin{aligned} sum\ (1,\ 2,\ 3,\ 4) &= 10, \\ product\ (1,\ 2,\ 3,\ 4) &= 24, \\ append\ (1,\ 2)(3,\ 4,\ 5) &= (1,\ 2,\ 3,\ 4,\ 5), \\ concat\ ((1,\ 2),\ (3,\ 4),\ ()) &= (1,\ 2,\ 3,\ 4), \\ map\ square\ (1,\ 2,\ 3,\ 4) &= (1,\ 4,\ 9,\ 16). \end{aligned}$$

Solution. Assume, that

$$postfix\ x\ y\ =\ append\ y\ (ux).$$

*list*1−1. Consider case (a):

$$\begin{aligned} length &= list1\ 0\ plus\ (K\ 1), \\ sumsquares &= list1\ 0\ plus\ square, \\ reverse &= list1\ (\ )\ postfix\ I, \\ identity &= list1\ (\ )\ list\ I. \end{aligned}$$

*list*1–2. Consider case (b):

$$\begin{aligned} sum &= list1\ 0\ plus\ I, \\ product &= list1\ 1\ multiply\ I, \\ append\ x\ y &= list1\ y\ list\ I\ x, \\ concat &= list1\ (\,)\ append\ I, \\ map\ f &= list1\ (\,)\ list\ f. \end{aligned}$$

To complete this solution, we need to substitute the parameters and perform the detailed computations. In addition, it is assumed that the steps of checking the samples should be performed.

10.3 Functor-as-object

Concluding, pay your attention to some features of the method of solving the task. First of all, the function *list*1 is a functional (and even a functor). Having defined this function, actually, a significantly more volume of job, than needed, was performed. More precisely, *list*1 reveals a significant dependency on parameters: varying the parameters, we result in a whole family of particular functions, every of which has a rather general presentation. By these means, a definition of *list*1 establishes a *notion*, or *concept*. As a concept is given by the description, then the *intensional* is determined. Selecting out different values of parameters, or referencing the *assignments*, actually, we obtain a whole family of individual functions. Counting the elements of this family, we obtain an *extensional* of the concept *list*1.

The functions, similar to *list*1, represent in programming an important idea, which has a distinctive name 'functor-as-object'. As could be seen, a program, composed of such objects, has a high degree of generality.

Exercises

Exercise 10.1. Determine the types of arguments of $list1$.

Hint. Consider a definition of $list1$ and will carry it into effect step by step.

1) Apply the function $(list1\ a\ g\ f)$ to a *list*. Assume, that it has the following structure:

   > list either is an empty element (null), or has a first element (head) and the rest of elements (tail), which, in turn, is a list.

   Assume, that this list has a type A. Thus,

   $$(list1\ a\ g\ f) \in (A \to B);$$

2) conclude, that arguments a should be taken from B;

3) the function f is applied to elements of A, and its values range C:

   $$f \in (A \to C);$$

4) thus, the function g should be applied to elements of C, and an effect of applying $(list1\ a\ g\ f)$ to elements of A ranges B; from this it follows, that

   $$g \in (C \to (B \to B)).$$

More details of analysis see in [5], p. 110-111.

Chapter 11

Isomorphism of c.c.c. and ACS

In this chapter we start to move deeper in the mathematical abstractions and will conform the operator thinking to combinatory thinking. Note, that there is the only operator in combinatory logic – *operator of application*, or the operator of acting of one object on another. The induced system of computations is called the *applicative computational system*, or ACS. This system conforms to a traditional operator computational system, which is represented by a special object – the *cartesian closed category*, or c.c.c.

11.1 Theoretical background

An idea of constructing a functional language, which contains no variables, is based on using of combinatory logic. Refusing of variables, a developer of programming language begins to use in operations arbitrary objects, which, in constructing a program, are allowed to be applied to each other. Evidently, this is the most purified form of using the objects.

As known, for proper using the objects, it is needed to stay within some variant of combinatory logic, which plays a role of shell theory. A set of combinators from this shell forms the "set of instructions" of an abstract computational system. It seems, that this set is not restricted by anything. In this case its mathematical properties stay unrevealed and potentially contain some form of contradiction.

Select as a shell theory the category theory. Fix in it a set of combinators, which have completely safe mathematical behavior: produce a cartesian closed category. Note, that in a cartesian closed category (c.c.c.) the functions are understood as operators, acting on its operands, which are written by positions. On the other hand, in applicative computational system there is the only operation – operation of application, which is interpreted as an acting of one object to another. The question is, are there any losses as a result of transition from a system of notions and definitions of c.c.c. to the system of notions and definitions of ACS and visa versa.

Use the following notational agreement:

$$[x, y] \equiv \lambda r.rxy,$$
$$< f, g > \equiv \lambda t.[f(t), g(t)] \equiv \lambda t.\lambda z.z(f(t))(g(t)).$$

The fact, that f is a mapping from A to B (in an intensional sense) is denoted by $f : A \to B$. Take into use the following mappings:

$$\begin{aligned} h &: A \times B \to C \quad x : A, y : B, \\ \Lambda_{ABC}h &: A \to (B \to C), \\ \Lambda_{ABC} &: (A \times B \to C) \to (A \to (B \to C)), \\ k &: A \to (B \to C), \\ \varepsilon_{BC} &: (B \to C) \times B \to C, \\ \varepsilon\circ < k \circ p, q > &: A \times B \to C, \\ p &: A \times B \to A, \quad q : A \times B \to B. \end{aligned}$$

In the following these mappings will often be used without explicit indication of their types, only if there is no ambiguity in reasoning.

11.2 Tasks

Task 11.1. As in the Karoubi's shell it is derivable, that:

$$h = \varepsilon\circ < (\Lambda h) \circ p, q >,$$

and is derivable:

$$k = \Lambda(\varepsilon\circ < k \circ p, q >),$$

then, first of all, from the previously given definitions it is needed to derive both of the equalities by yourself.

Solution.

$\Lambda-1$. At first, show, how to construct a translation of expressions from operator form to applicative one, i.e. find such a function h', that $h[x, y] = h'xy$. This can be obtained in the following way:

$$\begin{array}{rcl} h & = & \varepsilon\circ < (\Lambda h) \circ p,\ q >; \\ h[x,y] & = & (\varepsilon_{BC}\circ < (\Lambda h) \circ p,\ q >)[x,y] \\ & = & \varepsilon_{BC}(< (\Lambda h) \circ p,\ q > [x,y]) \\ & = & \varepsilon_{BC}[((\Lambda h) \circ p)[x,y],\ q[x,y]] \\ & = & \varepsilon_{BC}[(\Lambda h)(p[x,y]),\ q[x,y]] \\ & = & \varepsilon_{BC}[((\Lambda h)x), y] \\ & = & (((\Lambda h)x)y) = \Lambda\ h\ x\ y \\ & \equiv & h'\ x\ y. \end{array}$$

The computations above need in proper type assignment for the objects[1]. Hence, $h'\ x\ y = (\Lambda h)x\ y$.

$\Lambda-2$. At second, show, how to construct a translation from applicative form to operator form, i.e. find the function k' such,

[1]The expression $((\Lambda h)x)$ has the type $B \to C$. The expression y has been assigned type B. The expression $(((\Lambda h)x)y)$ obtains the type C.

that $k\,x\,y = k'[x,\,y]$. This can be obtained in the following way:

$$\begin{array}{rcl} k & = & \Lambda(\varepsilon\circ < k\circ p, q >); \\ kxy & = & \Lambda(\varepsilon\circ < k\circ p, q >)xy \\ & = & (\lambda u.\lambda v.(\varepsilon\circ < k\circ p, q >)[u,\,v])xy \\ & = & \varepsilon\circ < k\circ p,\; q > [x, y] \\ & \equiv & k'[x, y]. \end{array}$$

Thus, $k' = \varepsilon\circ < k\circ p,\; q >$ and $\Lambda k' = k$. It is left to note, that by '=' is denoted the relation of $\eta\xi$-convertibility: it is reflective, symmetric and transitive. Hence, it is a relation of equivalency. The last circumstance allows to make a conclusion, that

$$(A \times B \to C) \cong (A \to (B \to C)),$$

where symbol '$\cong$' is to be read as 'isomorphic'.

Chapter 12

Currying

The n-ary functions-operators, used in an operator programming, in the combinatory logic have the images as the objects, which inherit all their essential properties.

12.1 Theoretical background

12.1.1 Operators and functions

In a programming, it is often needed to establish a distinction between the notion of 'operator' and the notion of 'function'. In the first case the algebraic ideas are followed, when any operator is in advance assumed to have arity (the known number of arguments, called operands). This number of operands is known beforehand, and it is connected with a kind of particular operator. The arity also reveals itself in a manipulation with the functions, which though have an arbitrary character, but for every of them its number of arguments is known in advance. This is an old traditional way to consider the computations and creation of calculi. A relative opposing the symbol of function or operator, on one hand, to the symbols of arguments on other hand, is put in its basis. It is assumed silently, that the *objects*

do exist, but they are not on equal rights: the objects-operators are used by separate rules, and objects-operands — by other. The most often assumption in use deals with a replacement of one objects by others. In a framework of first order formalizations, the operands can be replaced by other objects, while operators can not. All of this has a sense of restriction, laid on the substitution in this systems. The first order systems are called the systems with a *restricted principle of comprehension*, because a particular accepted definition of substitution operation implements and idea of comprehension.

12.1.2 Comprehension

During the manipulation with arbitrary in reality functions, the reasoning concerning the computations should be carry out in terms of applying a symbol of function to the corresponding symbol of argument. Even more homogeneity in treating the functions and arguments can be achieved, if consider them as objects — without any additional reserve, — and reduce the study of a computation process to reasonings of acting (applying) of one object to another. In this case it is possible do not superimpose the burdensome restrictions on executing of a substitution, that gives rise to higher order formalisms. In case the order of such a theory is not restricted, then this is a theory with *unrestricted principle of comprehension*.

12.1.3 Connection between operators and functions

The disparate connections between both of the kinds of systems can be established, revealing the potential possibilities of both the approaches. In particular, consider a question, how by means of ACS (using operators of application and abstraction), to express an intensional representation of 2-ary, 3-ary, . . ., n-ary functions, considered

as operators. Use the following notational agreement:

$$\begin{array}{rcl} [x,y] & = & \lambda r.rxy, \\ h & : & A \times B \to C, \\ Curry_{ABC} & : & (A \times B \to C) \to (A \to (B \to C)). \end{array}$$

Hence, h is assumed to be the usual two placed operator, while $Curry$ is a transformation from operator to applicative notation[1]:

$$\begin{array}{rcl} Curry_{ABC}h & = & \lambda xy.h[x,y], \\ \lambda xy.h[x,y] & : & A \to (B \to C). \end{array}$$

12.2 Tasks

Task 12.1. Consider a family of functions h:

$$\begin{array}{l} h_2 : A \times B \to C, \\ h_3 : A \times B \times C \to D, \\ h_4 : A \times B \times C \times D \to E, \\ \ldots : \ldots . \end{array}$$

Find the family of mappings:

$$Curry_{ABC},\ Curry_{(A\times B)CD},\ Curry_{(A\times B\times C)DE},\ \ldots,$$

which make currying of these functions, i.e. transform them into an applicative form.

Solution. As an example, consider the currying of h_3 and h_4 .

$Curry{-}1$. Indeed, let $h_3 : (A \times B) \times C \to D$. Then $\Lambda_{(A \times B)CD}h_3 = \lambda xy.h_3[x,y] : A \times B \to (C \to D)$. Now it can be assumed that $\Lambda_{(A \times B)CD}h_3 = h_2'$, and therefore the

[1] In theoretical studies, instead notation of '$Curry$' is often used the notation 'Λ'; thereafter this last notation will be used.

next idea is to replace the first variable by the ordered pair of variables, i.e.

$$\begin{aligned}\Lambda_{AB(C\to D)}(\Lambda_{(A\times B)CD}h_3) &= \\ &= \lambda uv.(\Lambda_{(A\times B)CD}h_3)[u,v] \\ &= \lambda uv.(\lambda xy.h_3[x,y])[u,v] \\ &= \lambda uv.(\lambda y.h_3[[u,v],y]) \\ &= \lambda uvy.(h_3[[u,v],y]) : A\to\ (B\to\ (C\to\ D)) \\ &= (\Lambda_{AB(C\to D)}\ \circ\ \Lambda_{(A\times B)CD})\ h_3.\end{aligned}$$

Curry–2. Let now $h_4 : A\times B\times C\times D\to E$, where it is assumed, that

$$A\times B\times C\times D = (A\times B\times C)\times D = ((A\times B)\times C)\times D.$$

Then consider the transformation of currying step by step.

Step 1:

$$\Lambda_{((A\times B)\times C)DE}h_4 = \lambda xy.h_4[x,y] : ((A\times B)\times C)\to(D\to E).$$

Step 2:

$$\begin{aligned}\Lambda_{((A\times B)C(D\to E)}(\Lambda_{((A\times B)\times C)DE}h_4) &= \\ &= \lambda uv.(\Lambda_{((A\times B)\times C)DE}h_4)[u,v] \\ &= \lambda uvy.h_4[[u,v],y] : A\times B\to(C\to(D\to E)).\end{aligned}$$

Step 3:

$$\begin{aligned}\Lambda_{AB(C\to\ (D\to\ E)}(\lambda uvy.h_4[[u,v],y]) &= \\ &= \lambda\overline{x}\,\overline{y}.(\lambda uvy.h_4[[u,v],y])[\overline{x},\overline{y}] \\ &= \lambda\overline{x}\,\overline{y}vy.h[[[\overline{x},\overline{y}],v],y].\end{aligned}$$

In discussing the solution, obtained in (*Curry*-1) and (*Curry*-2), it could be noted, that the currying functions are as follows:

$$\Lambda_{(A\times B\times C)DE} = \Lambda_{AB(C\to(D\to E))}\circ\Lambda_{(A\times B)C(D\to E)}\circ\Lambda_{(A\times B)\times C)DE}.$$

For a purpose of finding out the solution in a general case, rewrite this equality as follows:

$$\Lambda_{(A_1\times A_2\times A_3)A_4B} = \\ = \Lambda_{A_1A_2(A_3\to(A_4\to B))} \circ \Lambda_{(A_1\times A_2)A_3(A_4\to B)} \circ \Lambda_{((A_1\times A_2)\times A_3)A_4B}.$$

Exercises

Exercise 12.1. It is left to the reader to derive corresponding equality for n-ary functions.

Hint. This can be done by induction on the number of argument places.

Exercise 12.2. Establish a connection between carried and uncurried functions.

Solution. For any function

$$f : [D_1 \times D_2 \times \cdots \times D_n] \to D$$

there is a curried function, which is equivalent to this function

$$Curry\ f : D_1 \to [D_2 \to [\ldots [D_n \to D] \ldots]]$$

Using the λ-notations, it is not difficult to conclude, that

$$Curry\ f = \lambda x_1 x_2 \ldots x_n . f(x_1, x_2, \ldots, x_n)$$

Next, $Curry$, in turn, is a higher order function:

$$Curry : [[D_1 \times D_2 \times \cdots \times D_n] \to D] \\ \to [D_1 \to D_2 \to \cdots \to D_n \to D] \\ Curry = \lambda f . \lambda x_1 x_2 \ldots x_n . f(x_1, x_2, \ldots, x_n) \\ = \lambda f x_1 x_2 \ldots x_n . f(x_1, x_2, \ldots, x_n)$$

Example 12.1.

$$\begin{aligned} plusc &= \lambda x_1 x_2. + x_1\ x_2 \\ &= \lambda x_1 x_2 . plus(x_1, x_2) \\ &= (\lambda f. \lambda x_1 x_2 . f(x_1, x_2)) plus \\ &= (\lambda f x_1 x_2 . f(x_1, x_2)) plus \\ &= Curry\ plus \end{aligned}$$

For the function $Curry$ it can be possible to find the $unCurry$ with a reversed effect:

$$\begin{aligned} unCurry &: [D_1 \to D_2 \to \cdots \to D_n \to D] \\ &\qquad \to [[D_1 \times D_2 \times \cdots \times D_n] \to D] \\ unCurry &= \lambda f. \lambda (x_1, x_2, \ldots, x_n). f x_1 x_2 \ldots x_n \end{aligned}$$

Example 12.2.

$$\begin{aligned} plus &= \lambda (x_1, x_2). + [x_1, x_2] \\ &= \lambda (x_1, x_2). plusc\ x_1\ x_2 \\ &= (\lambda f. \lambda (x_1, x_2). f\ x_1\ x_2) plusc \\ &= unCurry\ plusc. \end{aligned}$$

This example is generalized without any difficulty and results in the following.

Statement 12.1. If $Curry\ h\ =\ k$, then:

$$\begin{aligned} unCurry(Curry\ h) &= h \\ Curry(unCurry\ k) &= k \end{aligned}$$

Chapter 13

Karoubi's shell

A special category, called Karoubi's shell, allows laconic expression of all the available knowledge store, concerning operators, in terms of combinatory logic. In addition, the types are also encoded by the objects. By this way an embedding of typed application into type free programming environment is performed.

13.1 Theoretical background

A research work in programming often starts with a choice of the host theory, i.e. the shell theory, within which it is suitable to represent and study newly constructed mechanisms. By need of establishing and support of *type system*, a necessity of using such a shell theory arises, that allows to establish and consider various ideas, concerning the types. Possibly, within a shell it is better to avoid any a priori idea of types. In other words, a type free theory arises to deal with, and a good sample of this kind is the untyped λ-calculus.

In this chapter the establishing of connection of category theoretic notions with the notions of untyped λ-calculus is studied. Let $\mathcal{L}$ be a set of the terms of some calculus of λ-conversion. Karoubi's shell for $\mathcal{L}$, denoted by $\mathcal{C}(\mathcal{L})$, will be assumed to be a category, defined as

follows. Let

$$a \circ b \equiv \lambda x.a(bx)$$

for $a,\ b$ belonging to $\mathcal{L}$, where '$\circ$' is a sign of composition of the functions.

In addition, use the following notations:

$$\begin{aligned} \{a \in \mathcal{L} | a \circ a = a\} &- \text{set of objects in a category,} \\ \{f \in \mathcal{L} | b \circ f \circ a = f\} &- \text{set of morphisms, } Hom(a,b), \\ id &- \text{identity morphism, } id\, a = a \\ f \circ g &- \text{composition of morphisms.} \end{aligned}$$

13.2 Tasks

Task 13.1. Show, that $\mathcal{C}(\mathcal{L})$ is a category. This is left to verify by the reader, making a concordance with any form of definition of a category.

Task formulation. Assume the following definitions, needed for studying the objects. Remind, that in this case the necessity of carrying out the following equalities is presupposed:

$$\begin{aligned} A = \lambda x.A(A(x)) \quad &= A \circ A, && (A) \\ F = \lambda x.B(f(A(x))) &= B \circ f \circ A. && (f) \end{aligned}$$

1. Cartesian product:

$$A \times B = \lambda u \lambda z.z(A(u(\lambda x \lambda y.x)))(B(u(\lambda x y.y))).$$

2. Projections onto the first and second element respectively:

$$\begin{aligned} p_{AB} &= \lambda u.(A \times B)(u)(\lambda x \lambda y.x), \quad p_{AB} : A \times B \to A; \\ q_{AB} &= \lambda u.(A \times B)(u)(\lambda x \lambda y.y), \quad q_{AB} : A \times B \to B. \end{aligned}$$

3. Couple of functions:

$$< f, g >= \lambda t \lambda z.z(f(t))(g(t)) = \lambda t.[f(t), g(t)],$$
$$f : C \to A, \;\; g : C \to B, \;\; < f, g >: C \to (A \times B).$$

4. A set of mappings (a functional space):

$$(A \to B) = \lambda f.B \circ f \circ A.$$

5. Application (acting by a function on an argument):

$$\varepsilon_{BC} = \lambda u.C(u(\lambda xy.x)(B(u(\lambda xy.y)))),$$
$$\varepsilon_{BC} : (B \to C) \times B \to C.$$

6. The function of currying, i.e. the translation of the "usual" functions into applicative form, is called in honor of H. Curry (Remind one more: this function is often denoted by '$Curry$'; in the current context it is assumed, that $Curry \equiv \Lambda$, i.e. the currying function is denoted by 'Λ'):

$$\Lambda_{ABC} h = \lambda x \lambda y.h(\lambda z.z(x)(y)),$$
$$h : (A \times B) \to C, \;\; \Lambda_{ABC} h : A \to (B \to C).$$

It is needed to prove the following:

- Properties of the projections:

$$p_{AB} \circ < f, g >= f, \;\; q_{AB} \circ < f, g >= g,$$
$$< p_{AB} \circ h, q_{AB} \circ h >= h.$$

- Let $h : (A \times B) \to C, \;\; k : A \to (B \to C)$. Then

$$\varepsilon \circ < (\Lambda h) \circ p, q >= h,$$
$$\Lambda(\varepsilon \circ < k \circ p, q >) = k,$$

where $\Lambda = \Lambda_{ABC}, p = p_{AB}\,, \varepsilon = \varepsilon_{BC}, q = q_{AB}$.

Solution. A proof is reducible to the verifying the properties of the introduced objects.

$\mathcal{C}(\mathcal{L})$–1. Note, that the mapping $h : (A \times B) \to C$ has a translation into the term of λ-calculus, with the following equality:

$$h = \lambda x.C(h((A \times B)(x))),$$

where $x = [x_1, x_2]$. This is an immediate corollary of the equality (f).

$\mathcal{C}(\mathcal{L})$–2. Determine a combinatory characteristic h:

$$\begin{aligned} h[x_1, x_2] &= C(h((A \times B)[x_1, x_2])) \\ &= C(h(\lambda z.z(A([x_1, x_2]K))(B([x_1, x_2](KI))))) \\ &= C(h(\lambda z.z(A\ x_1)(B\ x_2))) = C(h[A\ x_1, B\ x_2]). \end{aligned}$$

Thus, $h[x_1, x_2] = C(h[A\ x_1, B\ x_2])$.

$\mathcal{C}(\mathcal{L})$–3. Again, pay attention to a necessity of taking into account the following equalities:

$$\begin{aligned} x &= \lambda z.a(x(\mathbf{1}\ z)) = A \circ x, \\ f &= \lambda z.B(f(A\ z)) = B \circ f \circ A, \end{aligned}$$

where $\mathbf{1} = \lambda y.y = I$.

$\mathcal{C}(\mathcal{L})$–4. It is not difficult to see (prove yourself!), that

$$(A \times B) = \lambda u.[A(u\ K), B(u(K\ I))],$$

where $K = \lambda xy.x$, $I = \lambda x.x$. Then a straightforward computation gives the following combinatory characteristic of a cartesian product:

$$\begin{aligned} (A \times B)[u, v] &= \lambda z.z(A([u, v]K))(B([u, v](K\ I))) \\ &= \lambda z.z(A\ u)(B\ v) \\ &= [Au, Bv]. \end{aligned}$$

$\mathcal{C}(\mathcal{L})$–5. Verification of properties of the projections leads to the following characteristics:

$$p_{AB}([u,v]) = (A \times B)[u,v]K = [Au, Bv]K = A\,u,$$
$$q_{AB}([u,v]) = (A \times B)[u,v](K\,I) = [A\,u, B\,v](K\,I) = B\,v.$$

$\mathcal{C}(\mathcal{L})$–6. Considering the ordered pair $[f,x]$, show how using the mapping ε_{BC} result in an application of f to x:

$$\begin{aligned}\varepsilon_{BC}([C \circ f \circ B, B \circ x]) &= \varepsilon_{BC}([f,x])\\ &= C([C \circ f \circ B, B \circ x]K(B([C \circ f \circ B, B \circ x](K\,I))))\\ &= C((C \circ f \circ B)(B(B \circ x)))\\ &= C(f(B\,x))\\ &= (C \circ f \circ B)(x)\\ &= f(x).\end{aligned}$$

Note a necessity of taking into account properties of a composition.[1] Essentially, a simple equality:

$$\Lambda_{ABC}\,h\,x\,y = h([x,y])$$

will be proved below (see chapter 12, where a currying function is defined).

$\mathcal{C}(\mathcal{L})$–7. Let t be represented by an ordered pair, i.e. $t = [t_1, t_2]$. Then

$$\begin{aligned}(\varepsilon \circ <(\Lambda\,h) \circ p, q>)t &= \varepsilon[(\Lambda\,h)(p\,t), q\,t]\\ &= \varepsilon[(\lambda xy.h[x,y])(pt), qt]\\ &= \varepsilon[(\lambda xy.h[x,y])t_1, t_2]\\ &= \varepsilon[\lambda y.h[t_1,y], t_2]\\ &= (\lambda y.h[t_1,y])t_2\\ &= h[t_1,t_2].\end{aligned}$$

This is a derivation of a characteristic equality:

$$\varepsilon \circ <(\Lambda\,h) \circ p, q> = h.$$

[1] Remind, that $C \circ f = f$, $B \circ x = x$.

$\mathcal{C}(\mathcal{L})$–8. Now derive the second characteristic equality, where $\#(t_1) = A$, $\#(t_2) = B$, $\#(t) = A \times B$, and $t = [t_1, t_2]$, i.e. t is represented by an ordered pair; by '#' the function of 'evaluating the type' is denoted:

$$\begin{aligned}\Lambda(\varepsilon\circ < k \circ p, q >)t_1\ t_2 &= (\varepsilon\circ < k \circ p, q >)([t_1, t_2]) \\ &= \varepsilon[k(p\ t), q\ t] = \varepsilon[kt_1, t_2] \\ &= (\lambda u.C(uK(B(u(K\ I)))))[k\ t_1, t_2] \\ &= C(k\ t_1(B(t_2))) \\ &= C(k\ t_1\ t_2) \\ &= k\ t_1\ t_2.\end{aligned}$$

By this[2] the characteristic equality

$$\Lambda(\varepsilon\circ < k \circ p, q >) = k.$$

is proved.

All the introduced equalities actually take place, and this completes a proof.

Exercises

Exercise 13.1. Give the representation of a metadata object, using combinators.

Solution. Assume, that a reader is familiar with the technique of variable objects (see, e.g., chapter 23 of the current volume). The constructing of metadata object will be done step by step, using a comprehension principle.

(1) Using a comprehension. For given the data objects, a comprehension generates the metadata object. Next, a superscript is used for indicating a relative level of generality. Thus, metaobject of level

[2] Take in mind, that $B(t_2) = t_2$.

$j+1$ is generated by the objects of level j. Note, that comprehension involves the description, so that metaobject z^{j+1} is identified by x^{j+1}.

$$x^{j+1} \equiv \mathbf{I}z^{j+1} : \underbrace{[\ldots[}_{j+1 \text{ times}} \; D]\ldots]\forall x^j : \underbrace{[\ldots[}_{j \text{ times}} D]\ldots](z^{j+1}(x^j) \leftrightarrow \Phi^j)$$

(2) Encoding a comprehension. Let be the following:

• truth values $[\,]$ represent a *variable domain*;

• the function g is a *predicate* function $g : \mathcal{U} \to [\,] \equiv [\mathcal{U}]$, hence, a *predicate concept* is the object

$$c \equiv \lambda A.\lambda g.x^2 \circ g \circ A \in [\mathcal{U}];$$

• j-th concept is determined by the following:

$$\begin{aligned} x^j \quad &\equiv \mathbf{I}z^j : \underbrace{[\ldots[}_{j \text{ times}} D]\ldots]\forall x^{j-1} : \underbrace{[\ldots[}_{j-1 \text{ times}} D]\ldots](z^j(x^{j-1}) \leftrightarrow \Phi^{j-1}) \\ &\equiv \{x^{j-1} \mid \Phi^{j-1}\} \\ &\subseteq \{x^{j-1} \mid x^{j-1} : A \to x^j\} \\ &= H_{x^j}(A) \equiv [\mathcal{U}]_A; \end{aligned}$$

• variable domain $[\mathcal{U}]_A$ is encoded by the expression

$$\lambda g.x^{j+1} \circ g \circ A;$$

• augmenting by the 'instances of a time' results in:

$$c \equiv \lambda A.(\underbrace{(\lambda h.[\,] \circ h \circ A)}_{[\,]_A} \circ c \circ \underbrace{(\lambda x^{j-1}.x^j \circ x^{j-1} \circ A)}_{\mathcal{U}_A});$$

(3) The final steps of encoding. To complete the encoding of comprehension, assume the following:

• fixing A and varying the parameters f, B results in a *predicate concept*

$$c \in [\mathcal{U}]_A, \;\; c_f \subseteq \mathcal{U}_B;$$

• data object c_f is defined as a *variable predicate*

$$c_f \equiv \{t \in \mathcal{U}_B \mid [t/x^{j-1}](\| \overline{\Phi}^j \|_f (B))\}$$

for the evolvent $f : B \to A$;
• metaobject c_f is encoded by

$$c_f \in \lambda c.[\mathcal{U}]_B \circ (c \circ f) \circ B.$$

Note. The usage of Karoubi' shell attracts the same ideas as the considered in chapter 10 in analysis of function $list1$. Actually, λ-calculus gives a wide range of various terms. Among them, only those are selected, a syntax form of which is specially restricted. A set of these restrictions, listed in the beginning of the current chapter, determines an *assignment*. A type free λ-calculus is taken as a *concept* theory (general shell), which is observed as a computation theory. As a result of its applying to the assignment, an *individual* theory is generated, which in turn has the properties of a shell. In particular, a special class of computations – categorical computations, – are representable within it. In this case a question of comparing the expressive power of the concept theory and the individual theory has a mathematical nature.

Chapter 14

Products and projections

A finalization of the started process of embedding can be achieved by introducing the ordered tuples of objects. As turns out, the applicative computations permit to get their representation.

14.1 Theoretical background

A problem of embedding the objects of one theory into other theory arises in considering the intensionally determined mappings, when it is needed to perform their translation in applicative (functional) language.

To fulfil this task, a small store of given definitions is needed:

$$K = \lambda xy.x,\ I = \lambda x.x,\ [x, y] = \lambda r.rxy$$

(ordered pair).

14.2 Task

Task formulation. Obtain the term of λ-calculus, corresponding to a cartesian product of n objects. In addition, determine such n terms,

that are the projections.

Hint. In case of $n = 2$ obtain

$$A_0 \times A_1 = \lambda u.[A_0(u\ K), A_1(u(K\ I))],$$
$$\pi_0^2 = \lambda u.(A_0 \times A_1)(u)K,$$
$$\pi_1^2 = \lambda u.(A_0 \times A_1)(u)(K\ I).$$

Solution. First of all we will try to find out a regularity in constructing the term.

$\times\pi{-}1$. Consider the case of $n = 3$.

$$\begin{aligned} A_0 \times A_1 \times A_2 &= (A_0 \times A_1) \times A_2 \\ &= \lambda u.[(A_0 \times A_1)(u\ K), A_2(u(K\ I))] \\ &= \lambda u.[(\lambda v.[A_0(v\ K), A_1(v(KI))])(uK), A_2(u(K\ I))] \\ &= \lambda u.[[A_0((u\ K)K), A_1((u\ K)(K\ I))], A_2(u(K\ I))]. \end{aligned}$$

Indeed, by a straightforward computations it can be derived, that

$$((A_0 \times A_1) \times A_2)[[x_0, x_1], x_2] = [[A_0(x_0), A_1(x_1)], A_2(x_2)].$$

To establish corresponding projections, let the following rather simple statement be formulated, and perform its verification.

Statement 14.1.

$$\pi_0^3 = \lambda u.(A_0 \times A_1 \times A_2)(u)(K)(K),$$
$$\pi_1^3 = \lambda u.(A_0 \times A_1 \times A_2)(u)(K)(K\ I),$$
$$\pi_2^3 = \lambda u.(A_0 \times A_1 \times A_2)(u)(KI).$$

Proof.

$$\begin{aligned} \pi_0^3[[x_0, x_1], x_2] &= [[x_0, x_1], x_2]K\ K \\ &= K[x_0, x_1]x_2K \\ &= [x_0, x_1]K \\ &= x_0; \\ \pi_1^3[[x_0, x_1], x_2] &= [[x_0, x_1], x_2]K(K\ I) \\ &= K[x_0, x_1]x_2(K\ I) = x_1; \\ \pi_2^3[[x_0, x_1], x_2] &= [[x_0, x_1], x_2](K\ I) \\ &= (KI)[x_0, x_1]x_2 = I\ x_2 = x_2. \end{aligned}$$

Indeed, for a kind of projection, defined above, the family of objects π_j^3, $j = 0, 1, 2$ has a behavior of projections. □

$\times\ \pi{-}2$. Consider the case of $n = 4$.

$$\begin{aligned} A_0 \times A_1 \times A_2 \times A_3 &= (A_0 \times A_1 \times A_2) \times A_3 \\ &= \lambda u.[(A_0 \times A_1 \times A_2)(u\ K), A_3(u(K\ I))] \\ &= \lambda u.[[[A_0(((u\ K)K)K), A_1(((u\ K)K)(K\ I))], \\ &\qquad A_2((u\ K)(K\ I))], A_3(u(K\ I))]. \end{aligned}$$

Statement 14.2.

$$\begin{aligned} \pi_0^4 &= \lambda u.(A_0 \times A_1 \times A_2 \times A_3)(u)(K)(K)(K), \\ \pi_1^4 &= \lambda u.(A_0 \times A_1 \times A_2 \times A_3)(u)(K)(K)(K\ I), \\ \pi_2^4 &= \lambda u.(A_0 \times A_1 \times A_2 \times A_3)(u)(K)(K\ I), \\ \pi_3^4 &= \lambda u.(A_0 \times A_1 \times A_2 \times A_3)(u)(K\ I). \end{aligned}$$

Proof.

$$\begin{aligned} \pi_0^4[[[x_0, x_1], x_2], x_3] &= [[[x_0, x_1], x_2], x_3]K\ K\ K = x_0, \\ \pi_1^4[[[x_0, x_1], x_2], x_3] &= [[[x_0, x_1], x_2], x_3]K\ K(K\ I) = x_1, \\ \pi_2^4[[[x_0, x_1], x_2], x_3] &= [[[x_0, x_1], x_2], x_3]K(K\ I) = x_2, \\ \pi_3^4[[[x_0, x_1], x_2], x_3] &= [[[x_0, x_1], x_2], x_3](K\ I) = x_3. \end{aligned}$$

□

Now a generalization of the product will be done.

Statement 14.3 *(generalization)*. By induction it can be proved, that:

$$\begin{aligned} &\pi_0^1 = I,\ \pi_0 = \lambda y.y\ K, \pi_1 = \lambda y.y(K\ I); \\ &\pi_n^{n+1} = \lambda y.y(K\ I), i \leq 1;\ \pi_j^{n+1} = \pi_j^n \circ \pi_0, 0 \leq j \leq n-1. \end{aligned}$$

Proof. Left to a reader. □

Hint. List some considerations, concerning the proof of generalization. The particular cases:

$n = 1$ (ordered 2-tuples).
$\pi_1^2 = \pi_1, \pi_0^2 = \pi_0^1 \circ \pi_0 = I \circ \pi_0 = \pi_0.$

$n = 2$ (ordered 3-tuples).
$\pi_2^3 = \lambda y.y(KI) = \pi_1,$
$\pi_1^3 = \pi_1^2 \circ \pi_0 = \pi_1 \circ \pi_0,$
$\pi_0^3 = \pi_0^2 \circ \pi_0 = \pi_0^1 \circ \pi_0 \circ \pi_0.$

$n = 3$ (ordered 4-tuples).
$\pi_3^4 = \pi_1,$
$\pi_2^4 = \pi_2^3 \circ \pi_0 = \pi_1 \circ \pi_0,$
$\pi_1^4 = \pi_1^3 \circ \pi_0 = \pi_1^2 \circ \pi_0 \circ \pi_0 = \pi_1 \circ \pi_0 \circ \pi_0,$
$\pi_0^4 = \pi_0^3 \circ \pi_0 = \pi_0^2 \circ \pi_0 \circ \pi_0 = \pi_0^1 \circ \pi_0 \circ \pi_0 \circ \pi_0$
$= \pi_0 \circ \pi_0 \circ \pi_0.$

Particular cases of projections of cartesian product in a transformed form are as follows:

$n = 2.$
$\pi_1^2 = \lambda y.y(K\ I),$
$\pi_0^2 = \lambda y.y\ K.$

$n = 3.$
$\pi_2^3 = \lambda y.y(KI),$
$\pi_1^3 = \lambda y.\pi_1(\pi_0\ y)$
$= \lambda y.\pi_1(y\ K)$
$= \lambda y.(y\ K)(K\ I),$
$\pi_0^3 = \lambda y.\pi_0(\pi_0\ y)$
$= \lambda y.\pi_0(y\ K)$
$= \lambda y.(yK)K.$

$n = 4$.

$$\pi_3^4 = \lambda y.y(KI),$$
$$\pi_2^4 = \lambda y.\pi_1(\pi_0\ y) = \lambda y.\pi_1(y\ K) = \lambda y.(y\ K)(K\ I),$$
$$\pi_1^4 = \lambda y.\pi_1(\pi_0(\pi_0\ y)) = \lambda y.\pi_1(\pi_0(y\ K))$$
$$= \lambda y.\pi_1((y\ K)K) = \lambda y.y((y\ K)K)(K\ I),$$
$$\pi_0^4 = \lambda y.\pi_0(\pi_0(\pi_0\ y)) = \lambda y.\pi_0(\pi_0(y\ K))$$
$$= \lambda y.\pi_0((y\ K)K) = \lambda y.y((y\ K)K)K.$$

Non-formal consideration of the solved problem is reduced to the following reasons. Ordered n-tuples are well known, in particular, they are used to constitute the relations of a relational data base. Remind, that a data manipulation language of relational data base management systems deals with the sets of n-tuple as particular *objects*, called relations. An often used query to data base is to cut the available relations, and can be reduced to projection operations. To execute these operations, the implemented query language is needed.

In this case the ACS is used as a programming system. More exact reasoning assumes the assumptions, that, as a shell system, the Karoubi's shell is used (see chapter 13). Both the objects-as-products and objects-as-projections are built in this shell. In turn, they consist of the objects-as-combinators. The programming is performed completely in terms of objects.

14.3 Product and cartesian closed category

In this section restrict the consideration by giving the definition a cartesian closed category, or c.c.c. This definition is important for better understanding the matters, covered in chapter 11, to reading of which it is recommended to go back one more time.

Let $\mathcal{C}$ be a category. The identity morphism of the object A will be denoted by id_A.

$\mathcal{C}$ is called a *cartesian closed category*, if the following conditions are valid:

1) in $\mathcal{C}$ there is a *terminal object* T such, that for any object $A \in \mathcal{C}$ there is the unique morphism $!_A : A \to T$;

2) for any objects $A_1, A_2 \in \mathcal{C}$ there is the object $A_1 \times A_2$, their *Cartesian product*, equipped with morphisms $p_i : A_1 \times A_2 \times A_i$, the *projections*, such, that for any $f_i : C \to A_i$ for $i = 1, 2$ there is the only morphism $\langle f_1, f_2 \rangle : C \to A_1 \times A_2$ such, that $p_i \circ \langle f_1, f_2 \rangle = f_i$;

3) for $A, B \in \mathcal{C}$ there is the object $B^A \in \mathcal{C}$, *power*, $B^A \equiv A \to B$ with the morphism

$$\varepsilon = \varepsilon_{AB} : B^A \times A \to B,$$

such, that for any $f : C \times A \to B$ there is the only

$$\Lambda f : C \to B^A,$$

satisfying the equality $f = \varepsilon \circ (\Lambda f \times id_A)$.

A discussion of fruitfulness of this definition needs to be familiar with the chapters 11-12, as well as with the chapters 19-22, in which one of the most promising applications of c.c.c. to constructing the abstract machine is given.

Chapter 15

Embedding Lisp into ACS

Non-trivial application – significant and, essentially, complete fragment of the known programming system Lisp (*Lis*t *p*rocessing), – is embedded into an applicative computational system.

15.1 Theoretical background

One of the most difficult problems in developing the user (language) interface is in proper and suitable embedding of the application into programming environment. The application of an object approach, and, naturally, an object-oriented programming, as known, promote the success in its solving. In practice, a special set of the classes, which exports the *methods* for applied programs, is defined.

In mathematic, this mode is known as an *embedding*, when in a host theory, named a *metatheory*, the objects, a set of which constitutes the *embedded theory*, are constructed. Metatheory is assumed as a shell, and the objects of embedded theory can be accommodated by need, without violating a framework of the metatheory. Consider a using of this method by example. As a metatheory, the type free combinatory logic will be used. It will be assumed as a shell of the objects. The system of objects with their characteristic equalities,

which implement the interface of Lisp, i.e. constitute a functionally complete universe of discourse around the *lists* and *atoms*, will be used as an embedded system. By a list of objects, as usually, we mean a finite sequence of objects, among which the other lists can be.

Language Lisp is, essentially, the type free language. Its main constructions are the same as the constructions of type free λ-calculus. It is well known, that these constructions are expressible by means of combinatory logic. The aim of this study is to establish the combinatory characteristics of some functions of programming language.

For this purpose, the $\eta\xi$-calculus of λ-conversions will be used. The postulates of a relation of conversion '*conv*' (denoted by the symbol '=') are listed below:

$$(\alpha) \qquad \lambda x.a = \lambda z.[z/x]a, \qquad (\beta) \qquad (\lambda x.a)b = [b/x]a,$$

$$(\nu) \qquad \frac{a = b}{ac = bc}, \qquad (\mu) \qquad \frac{a = b}{ca = cb},$$

$$(\tau) \qquad \frac{a = b;\ \ b = c}{a = c} \qquad (\sigma) \qquad \frac{a = b}{b = a} \qquad (\rho) \qquad a = a$$

$$(\eta) \qquad \lambda x.bx = b,\ x \notin b. \qquad (\xi) \qquad \frac{a = b}{\lambda x.a = \lambda x.b}$$

15.2 A main task

Task 15.1. Using the combinators, represent the following set of functions of the Lisp-system:

$$\{Append,\ Nil,\ Null,\ List,\ Car,\ Cdr\}. \qquad (Lisp)$$

Task formulation. Consider the properties of these functions of Lisp language.

- A concatenation of two lists is constructed by the function *Append*. This function has a property of *associativity*:

$$A \frown B \frown C = (A \frown B) \frown C, \qquad (Append)$$

where A, B, C are arbitrary lists, symbol '$\frown$' is an infix form of notation of the function *Append*.

- Empty list is denoted by $< >$ and is equal to the object Nil. Evidently,

$$A \frown <> = <> \frown A = A. \qquad (Nil), (<>)$$

- The function $Null$ recognizes an empty list:

$$Null\ A = \begin{cases} \overline{1}, & \text{if } A = Nil, \\ \overline{0}, & \text{otherwise.} \end{cases} \qquad (Null)$$

- The function $List$ constructs from atom a list of length 1:

$$List\ x =< x >, \qquad (List)$$

where x is an atom, and $< x_1,\ x_2,\ \ldots\ , x_n >$ is a list of length n.

- The function Car selects the first element from a list:

$$Car < x_1,\ x_2,\ \ldots\ ,\ x_n >= x_1. \qquad (Car)$$

- The function Cdr deletes the first element from a list:

$$Cdr < x_1,\ x_2,\ \ldots\ ,\ x_n >=< x_2,\ \ldots\ , x_n > . \qquad (Cdr)$$

Using these properties, the following axiom schemes are formulated:

$$\begin{aligned}
Append\ a\ (Append\ b\ c) &= Append(Append\ a\ b)c, && (15.1)\\
Append\ Nil\ a &= Append\ a\ Nil, && (15.2)\\
Null\ Nil &= \overline{1}, && (15.3)\\
Null(Append(List\ a)b) &= \overline{0}, && (15.4)\\
Car(Append(List\ a)b) &= a, && (15.5)\\
Cdr(Append(List\ a)b) &= b, && (15.6)
\end{aligned}$$

where a, b, c are any objects. Prove, that the axioms (15.1)–(15.6) are derivable in $\eta\xi$-calculus of λ-conversion.

Solution. Perform a successive translation of the intensional equalities (15.1)–(15.6) into the terms and formulae of combinatory logic.

Lisp–1. It will be shown, that the function *Append* corresponds to the object B with a combinatory characteristic (B) : $Babc = a(bc)$ (axiom scheme (15.1)):

$$\begin{aligned}
Ba(Bbc)\,x &= a(Bbcx) && (\text{by } (B))\\
&= a(b(cx)) && (\text{by } (B))\\
&= Bab(cx) && (\text{by } (B))\\
&= B(Bab)c\,x. && (\text{by } (B))
\end{aligned}$$

Using the rule of transitivity (τ), it can be derived:

$$Ba(Bbc)\,x = B(Bab)c\,x.$$

In $\eta\xi$-calculus it is derivable, that for a variable x:

$$\frac{z_1x = z_2x}{z_1 = z_2},$$

or, in a linear notation, $z_1x = z_2x \Rightarrow z_1 = z_2$, i.e., assuming

$z_1 = Ba(Bbc)$ $z_2 = B(Bab)c$, obtain the following:

$$\begin{array}{lll}
(1) & z_1x = z_2x \Rightarrow \lambda x.z_1x = \lambda x.z_2x, & (\text{ by } (\xi)) \\
(2) & \lambda x.z_1x = z_1, & (\text{ by } (\eta)) \\
(3) & z_1 = \lambda x.z_1x, & (\text{ by } (\sigma), (2)) \\
(4) & \lambda x.z_2x = z_1, & (\text{ by } (\tau), (1), (3)) \\
(5) & z_2 = \lambda x.z_2x, & (\text{ by } (\eta)) \\
(6) & z_1 = z_2. & (\text{ by } (\tau), (4), (5))
\end{array}$$

Thus, the axiom scheme (15.1) is verified.

Lisp–2. The axiom scheme (15.2) will be proved, taking, that $Nil \leftrightarrow I$ is valid[1], while (I) : $Ia = a$.

$$\begin{array}{lcll}
BIa\,x & = & I(ax) & (\text{ by } (B)) \\
 & = & ax & (\text{ by } (I)) \\
 & = & a(Ix) & (\text{ by } (I)) \\
 & = & BaI\,x. & (\text{ by } (B))
\end{array}$$

As $BIax = BaIx$, then $BIa = BaI$. This conclusion is established by the method, similar to those, used in deriving the previous axiom. Since it will be involved rather often, the corresponding rule is to be especially formulated:

$$(\nu^{-1}) : \qquad \frac{ux = vx}{u = v}.$$

This rule turns out working in the case, when x is a variable. If this rule matches with one of the monotonicity rules (ν), then it can be observed, that its premise and conclusion have been changed their positions. From this reason (in case, when x is a variable) this rule (ν^{-1}) can be named as a rule of *reversed monotonicity*.

[1] The symbol '$\leftrightarrow$' denotes a one-to-one correspondence.

Lisp–3. The axiom scheme (15.3) will be re-written as the equality $\overline{1} = Null\ Nil$ (by the rule (σ)), where $Nil = I$, $\overline{1}$ is a numeral with a combinatory characteristic $\overline{1}ab = ab$, or, in terms of λ-calculus, $\overline{1} = \lambda xy.xy$. Note, that

$$\begin{array}{ll} (D): & Dabc = cab, \\ (\overline{0}): & \overline{0}ab = b, \text{ or } \overline{0} \leftrightarrow \lambda xy.y. \end{array}$$

It is needed to be taken into consideration, that $KI \leftrightarrow \overline{0}$, i.e. $KI = \overline{0}$. Now, an object, corresponding to the function $Null$ will be found:

$$\begin{array}{lcll} \overline{1} & = & \lambda xy.xy & (\text{ by Df. } \overline{1}) \\ & = & \lambda x.x & (\text{ provable } \lambda xy.xy = \lambda x.x) \\ & = & I & (\text{ by Df. } I) \\ & = & KI(K(K\overline{0})) & (\text{ by scheme } (K)) \\ & = & I(KI)(K(K\overline{0})) & (\text{ by scheme } (I)) \\ & = & D(KI)(K(K\overline{0}))I & (\text{ by scheme } (D)) \\ & = & D\overline{0}(K(K\overline{0}))I. & (\text{ by scheme } (\overline{0})) \end{array}$$

Comparing the obtained expression $D\overline{0}(K(K\overline{0}))I = \overline{1}$ and the scheme $Null\ Nil = I$ (or, more rigorously, scheme $Null\ Nil = \overline{1}$), conclude, that $D\overline{0}(K(K\overline{0}))I \leftrightarrow Null\ Nil$.

Lisp–4. The following transformations are to be performed:

$$\begin{array}{lcll} \overline{0} & = & K\overline{0}(b\overline{0}) & (\text{ by scheme } (K)) \\ & = & K(K\overline{0})a(b\overline{0}) & (\text{ by scheme } (K)) \\ & = & Da(b\overline{0})(K(K\overline{0})) & (\text{ by scheme } (D)) \\ & = & B(Da)b\overline{0}(K(K\overline{0})) & (\text{ by scheme } (B)) \\ & = & D\overline{0}(K(K\overline{0}))(B(Da)b). & (\text{ by scheme } (D)) \end{array}$$

Compare the derived expression $D\overline{0}(K(K\overline{0}))(B(Da)b) = \overline{0}$ with the axiom scheme (15.4): $Null(Append(List\ a)b) = \overline{0}$. If take into consideration, that $D\overline{0}(K(K\overline{0})) \leftrightarrow Null$, $B \leftrightarrow Append$, then $D \leftrightarrow List$.

Lisp–5. The same way as above, find an object, corresponding to the function Car:

$$\begin{array}{rcll} a & = & Ka(bc) & (\text{ by scheme } (K)) \\ & = & Da(bc)K & (\text{ by scheme } (D)) \\ & = & B(Da)bcK & (\text{ by scheme } (B)) \\ & = & DcK(B(Da)b). & (\text{ by scheme } (D)). \end{array}$$

Evidently, that $DcK \leftrightarrow Car$.

Lisp–6. Using the same way, construct an object, corresponding to the function Cdr:

$$\begin{array}{rcll} b\,z & = & I(bz) & (\text{ by scheme } (I)) \\ & = & KIa(bz) & (\text{ by scheme } (K)) \\ & = & Da(bz)(KI) & (\text{ by scheme } (D)) \\ & = & B(Da)bz(KI) & (\text{ by scheme } (B)) \\ & = & (\lambda xy.xy(KI))(B(Da)b)\,z. & (\ (\beta), (\sigma)) \end{array}$$

Since

$$b\,z = (\lambda xy.xy(K\ I))(B(Da)b)\,z,$$

then

$$(\lambda xy.xy(K\ I))(B(Da)b) = b$$

for a variable z (the reversed monotonicity rule is applied), i.e.

$$\lambda xy.xy\overline{0} \leftrightarrow Cdr.$$

Answer. The results of representing the main functions of the programming system Lisp are arranged in a table below:

N° by order	Function of Lisp	Object of combinatory logic or λ-calculus
1	*Append*	B
2	*Nil*	I
3	*Null*	$D\overline{0}(K(K\overline{0}))$
4	*List*	D
5	*Car*	DcK
6	*Cdr*	$\lambda xy.xy\overline{0}$

15.3 Concluding remarks

A solution of the formulated problem, undoubtedly, is done by the method of embedding. In can be re-formulated in terms of objects.

The system of equalities (15.1)–(15.6), as a set, is considered as an *assignment*, for which a shell theory is individualized. This shell theory is a concept, while a result of individualizing will be dependent on a choice of the concept. For instance, select as a shell the $\eta\xi$-calculus of λ-conversions. Then every of the concept objects

$$Append, \qquad Nil, \qquad Null, \qquad List, \qquad Car, \qquad Cdr$$

will result in, under performing the assignment, a corresponding individual object

$$B, \qquad I, \qquad D\overline{0}(K(K\overline{0})), \qquad D, \qquad DcK, \qquad \lambda xy.xy\overline{0}.$$

The individual objects constitute an individual theory, that is similar to Lisp language (and is a shell). The $\eta\xi$-calculus of λ-conversions reveals one remarkable feature: both the concept objects and individual objects relatively the assignment (15.1)–(15.6) remain within the same universe.

Chapter 16

Supercombinators

Supercombinators constitute a purely object programming system, embedded into combinatory logic. This immediately satisfies the needs in denotational computation of instruction of programming language, when a functional meaning of the program is expressed by the objects. It is essentially, that a computation begins with some beforehand known set of instructions. During an evaluation of the program, the instructions, which are needed in a computation, but are beforehand unknown, arise in dynamics, and they are fixed in the programming system in addition.

16.1 Theoretical background

There are two approaches to use the supercombinators for implementation of applicative programming languages. In the first of them a program is compiled by the fixed set of supercombinators (in a non-optimal variant they are S, K, I) with beforehand known definitions. But the second approach, in which the definitions of supercombinators are generated by a program itself in a process of compiling, is primarily considered.

16.1.1 Notion of supercombinator

Definition 16.1. Supercombinator $\$S$ of arity n is the lambda-expression

$$\$S \stackrel{\text{def}}{=} \lambda x_1.\lambda x_2.\ldots.\lambda x_n.E,$$

or, equally, the abstraction of a kind:

$$\$S \stackrel{\text{def}}{=} [x_1].[x_2].\ldots.[x_n].E,$$

where E is not an abstraction. Thus, all the "leading" symbols of abstraction $[\cdot].$ are related only to x_1, x_2, ..., x_n, and at the same time the following conditions are valid:

(1) $\$S$ does not contain any free variables;

(2) every abstraction in E is a supercombinator;

(3) $n \geq 0$, i.e. the availability of symbols '$[\cdot].$' is not necessary.

Supercombinator redex is the application of a supercombinator to n arguments, where n is its arity. A substitution of the arguments in the body of supercombinator instead of free occurrences of corresponding formal parameters is called a *reduction* of the supercombinator.

This definition could be compared with a definition of combinator. In other words, we can say, that the combinator is such a lambda-abstraction, which does not contain any free occurrences of the variables. Some of the lambda-expressions are the combinators, and some of the combinators are the supercombinators.

Example 16.1. The expressions

```
3,  + 2 5,   [x].x,  [x].+ x x,  [x].[y].xy
```

are the super combinators.

Example 16.2. The following terms are not the supercombinators:

`[x].y` — (the variable `y` has a free occurrence),
`[y].- y x` — (the variable `x` has a free occurrence).

Example 16.3. The term `[f].f([x].f x 2)` is a combinator, because all the variables (`f` and `x`) are bound, but is not a supercombinator, because the variable f is free in the inner abstraction and condition (2) of the definition 16.1 is violated. In accordance with this definition, the combinators S, K, I, B, C are the supercombinators. Hence, SK-machine, represented in a theory of categorical computations, implements one of the methods of using the supercombinators.

Supercombinators of arity 0 are the *constant applicative forms* (CAF).

Example 16.4. The expressions:

a) `3`, b) `+ 4 6`, c) `+ 2`

are the CAFs.

The point c) shows, that CAF can be the function, though it does not contain the abstractions. Since CAF has no symbol of the abstraction, the code is not compiled for them.

Exercise 16.1. Show, that the following expressions are the supercombinators:

`1 0`, `[x].+ x 1`, `[f].([x].+ x x)`.

Exercise 16.2. Explain, why the following expressions are not the supercombinators:

`[x].x y z`, `[x].[y].+ (+ x y) z`.

Exercise 16.3. Give an example of the combinator, which is not a supercombinator.

16.1.2 Process of compiling

The actual programs contain a significant number of the abstractions. The program is to be transformed so that to contain the only supercombinators. Accept, that the names of the supercombinators start with the symbol '$', e.g.:

```
$XY = [x].[y].- y x.
```

To stress the features of supercombinatros, this definition is rewritten as:

```
$XY x y = - y x.
```

The selected strategy is in a transforming the abstraction, which should be compiled into:

(i) the set of supercombinator definitions,

(ii) the evaluated expression.

This will be written by:

```
Definitions of supercombinators
  ..............................
  ..............................
  ------------------------------
      Evaluating expression
```

Example 16.5. The expression $([x].[y].-\ y\ x)3\ 4$ can be represented by:

```
$XY x y = - y x
---------------
    $XY 3 4
```

Example 16.6. The expression ($\$XY$ 3) is not a redex and cannot be evaluated. Thus, the definitions of supercombinators are determined as a set of rewriting rules. The reduction is in rewriting the expression, which matches a left part of the rule, replacing it by the expression, which is the right part. These systems are known as term rewriting systems.

Exercise 16.4. Is it possible to evaluate the expressions:

```
$XY 5,  $XY 5 7,  $XY 3 4 7?
```

16.1.3 Transformation to supercombinators

Supercombinators are easy to compile. Give a description of the *algorithm* of transforming the abstraction into combinators. Consider a program, which does not contain any supercombinator:

```
([x].([y].+ y x) x) 4.
```

Select the most inner abstraction, i.e. such an abstraction that has no other abstractions:

```
([y].+ y x).
```

It has a free occurrence of the variable `x`, therefore this abstraction is not a supercombinator.

(1) The following supercombinator is obtained by a simple transformation — usual β-reduction:

```
([x].[y].+ y x)x.
```

(2) Substitute it in the source program:

```
([x].([w].[y].+ y w) x x)4.
```

(3) Assign to this supercombinator the name `$Y`.

(4) Now it can be seen, that `[x].`-abstraction is also a supercombinator. It is assigned the name `$X` (the abstraction is compiled) and put into compiled code:

```
$Y w y = + y w
  $X x = $Y x x
---------------
      $X 4
```

The resulting program can be executed, using a reduction of supercombinators:

```
$X 4 --> $Y 4 4 = + 4 4 = 8.
```

Thus, an algorithm, transforming the abstractions into supercombinators, is as follows:

LOOP-*while*: is an abstraction?

(1) select any abstraction, which *has no* other abstractions,

(2) shift all the free in this abstraction variables as extraparameters,

(3) assign to this abstraction a name (e.g., `$X34`),

(4) replace an occurence of the abstraction in the program by the name of supercombinator, which should be applied to free variables,

(5) compile the abstraction and assign a name to this compiled code.

END-*while*

During these transformations a volume of this program increased. This is a payment for the simplicity of reduction rules. Note that this transformation procedure reduces the source program to the following layout:

```
... Definitions of supercombinators ...
---------------------------------------
                    E
```

Since the expression `E` is the evaluated expression of the highest level, then it contains no free variables. It can be considered the a supercombinator of arity 0, i.e. CAF:

```
... Definitions of supercombinators ...
                  $Prog = E
---------------------------------------
                    $Prog
```

The procedure of transforming the abstractions into supercombinators is called *lambda lifting*, because all the abstractions are lifted to the higher level.

Exercise 16.5. Transform and execute the following programs:
1) `([x].([y].- y x) x) 5`,
2) `([z].+ z(([x].([y].× y x) x) 4)) 2`.

16.1.4 Eliminating redundant parameters

Consider a simple optimization of lambda-lifting algorithm. Let:

```
[x].[y].- y x.
```

be written. Although it is a supercombinator, apply to it the lambda-lifting algorithm.

(1) Select the innermost abstraction `[y].- y x`. The variable `x` has a free occurrence. Shift it as an extraparameter:
`([x].[y].- y x) x.`

Let:

```
$Y = [x].[y].- y x.
```

Thus:

```
$Y x y = - y x
--------------
   [x].$Y x
```

(2) Let `$X = [x].$Y x`. Then:

```
$Y x y = - y x
  $X x = $Y x
--------------
      $X
```

(3) In this case the definiton of `$X` is simplified to `$X = $Y` (by η-reduction).

Thus, the supercombinator `$X` is redundant and can be replaced by `$Y`:

```
$Y x y = - y x
--------------
      $Y
```

This results in the availability of two optimizations:

1) eliminating the redundant parameters from the definitions by η-reduction,

2) eliminating the redundant definitions.

16.1.5 Ordering of the parameters

In all the programs considered above the order of lifting the variables as extraparameters was arbitrary. For instance, consider the program:

```
..........
   (...
([x].[z].+ y (× x z))
   ...),
```

where '. . .' denotes the outer for the `[x].`-abstraction context. Let us start with the lambda-lifting algorithm.

(1) Select out the innermost abstraction

```
[z].+ y(× x z).
```

This abstraction is not a supercombinator, because it contains two free variables `x` and `y`. At the next step of algorithm the free variables are to be shifted as extraparameters.

A question is, in what order should the variables be arranged: it is possible at first put the `x`, and then `y`, or at first put `y`, and then `x`. Both the variant are performed.

Variant 1.

(2) Lift the variables, arranging them in the order `x, y`:

```
([x].[y].[z].+ y(× x z))x y.
```

(3) Assign to obtained supercombinator the name:

```
$S = ([x].[y].[z].+ y(× x z)).
```

(4) Substitute `$S` into the source program:

```
$S x y z = + y(× x z)
---------------------
        (...
   ([x].$S x y)
        ...)
```

The expression `[x].$S x y` is not a supercombinator, therefore, in turn, the lambda-lifting algorithm should be applied to it.

(2) Lift the free variable `y`:

```
([y].[x].$S x y)y.
```

(3) Assign to a supercombinator the name `$T y x = $S x y`.

(4) Substitute the combinator into the program:

```
$T y x = $S x y
---------------
      $T y.
```

Turn to the main algorithm, in which now obtain:

```
$S x y z = + y(× x z)
   $T y x = $S x y
   (...  $T y ...).
```

Variant 2.

(2) Lift the variables, arranging them in the order `y, x`:

```
([y].[x].[z].+ y(× x z)) y x.
```

(3) Assign to the obtained supercombinator a name:

```
$S = ([y].[x].[z].+ y(× x z)).
```

(4) Substitute $\$S$ into the source program:

```
$S y x z = + y(× x z)
------------------------
        (...
   ([x].$S y x)
        ...)
```

The expression `[x].$S y x` is not a supercombinator, because it contains an occurrence of free variable `y`. Apply the lifting algorithm to `[x].$S y x`.

(1) The innermost abstraction matches the whole program.

(2) Lift the variable `y`: `([y].[x].$S y x)y`.

(3) Now the supercombinator above is to be named:
`$T = [y].[x].$S y x`.

(4) Substitute the combinator into the program:

```
$T y x = $S y x
---------------
      $T y
```

Turn now to the main algorithm. Derive, that:

```
$S x y z = + y (× x z)
   $T y x = $S y x
----------------------
   (... $T y ...)
```

In accordance with the rule of elimination the redundant parameters (see subsection 16.1.4) we obtain: `$T = $S`, therefore `$T` can be eliminated. Then the compiled code is as follows:

```
$S y x z = + y(× x z)
--------------------
       $S y
```

In the first variant such an otimization is impossible. In the initial program

```
(... ([x].[z].+ y(× x z)) ...)
```

there are two bound variables: `x` and `z`. In the second variant an ordering of the free variables is done at the step (2) so, that in the obtained supercombinator the bound in a program variables `x` and `z` are at the last place: `$S y x z`. Only in this case the code can be optimized. Thus, the free variables should be arranges such a way, that the bound variables were put in the tail places in a list of supercombinator parameters.

Any abstraction is assigned by a lexical number of its level, which is determined by the number of outer symbols of the abstraction.

Example 16.7. In the expression

```
([x].[y].+ x(× y y))
```

the operator of `[x].`-abstraction is at the level 1, and of `[y].`-abstraction is at the level 2.

Give the formulation of the rules, allowing to introduce a lexical number of the level:

1) a lexical number of the abstraction is greater by one of the number of its outer abstractions; if there are no such abstractions, the the number is equal to 1;

2) the lexical number of a variable is the number of the abstraction, bounding this variable; if the number of `x` is less than the number of `y`, then is said, that `x` is more free than `y`;

3) a lexical number of a constant is equal to 0.

To improve the abilities for optimizing, the extraparameters should be arranged by increasing of their lexical numbers.

16.1.6 The lambda-lifting with a recursion

Note, that the lambda-abstractions, as a rule, have no names. To the contrast, the supercombinators are named. Besides this, a supercombinator can refer to itself. This means, that recursive supercombinators are implemented directly, without using the fixed point

combinator Y. Certainly, the recursive definitions can be transformed into the non-recursive ones, using Y, by to do this the additional rules are to be introduced.

Example 16.8. To make `$F` non-recursive, the following definitions sould be intriduced:

```
$F = Y $F1                          }
$F1 E x = $G (F(- x 1)) 0.          }        (*)
```

An additional definition is marked by the symbol '*'. Since `Y` should be reduced, then such a definition of `$F` involves more of the reductions, than a recursive version does.

Notation:

```
$S1 x y = B1
S2 f = B2
...
E
```

is equl to the expression:

```
letrec
    $S1 = [x].[y].B1
    $S2 = [f].B2
...
in
    E.
```

It means, that `E` contains `$S1, $S2, ...`, the recursive definitions of which are given in `letrec`. The lambda-lifting algorithm proceeds the same way, as previously: the expressions, containing in `letrec`, are treated the same way as any other expressions. Nevertheless, a question is, what a lexical level number should be assigned to the variables, bound in `letrec`. Since these variables are evaluated, when nearest outer abstraction is applied to its argument, then their lexical number is the same as the number of this abstraction. In case of

absence of outer abstractions the lexical number equals 0. Such a number is assigned to the constants and supercombinators. Within `letrec`, which contains no abstractions, it cannot be any free variables, besides those variables, which already have been defined in `letrec`.

Such a `letrec` is the combinator. To transform it to supercombinator, it is necessary to make a lambda-lifting, eliminating all the inner abstractions. The variables, bound in `letrec` of level 0, will not be shifted as extraparameters, because the constants (remind, that they are of level 0) are not lifted.

Example 16.9. Consider the program, resulting in an infinite list of ones:

```
------------------
letrec x = cons 1 x
in x
```

In this program the `letrec` is at the level 0, and there are no abstractions, hence, `x` is already a supercombinator:

```
$x = cons 1 x
-------------
x
```

Example 16.10. Consider a recursive function of factorial:

```
-----------------------------------
letrec fac = [n].IF (= n 0) 1
                    (× n (fac(- n 1)))
in fac 4
```

In this case `letrec` has the number 0, and there are no abstractions within `[n].`-abstraction. Hence, `fac` is a supercombinator:

```
$fac n = IF (= n 0) 1 (× n(fac(- n 1)))
$Prog = $fac 4
---------------------------------------
$Prog
```

Exercise 16.6. Compile the program:

```
-----------------------------------------
let
     inf = [v].(letrec vs = cons v vs in vs)
in
     inf 4
```

Hint: `let` means, that in the expression `inf 4` the function `inf` contains the definition, which have been written within `let`. The function `inf v` returns an infinite list of symbols `v`.

16.1.7 Execution of the lambda-lifting algorithm

Consider the program, adding the first 100 natural numbers:

```
SumInts m = sum (count 1)
            where
            count n = [], n > m
                    = n:count(n + 1)
     sum [] = 0
     sum (n:ns) = n + sum ns
-----------------------------------
SumInts 100
```

`SumInts` is a composition of the functions `sum` and `count`: at first, the function `count` is applied to 1, and next its result is used as an input of the function `sum`. The function `count` works as follows:

```
count 1 = 1:count 2 (because both n = 1, m = 100,
                     and condition n > m do not hold)
        = 1:2:(count 3) (executed count 2)
...
        = 1:2: ... :100:(count 101)
        = 1:2: ... :100:[] (since
                            n = 101, m = 100,
                            then n > m, (count 101 = []) )
```

Now the list `1:2:3: ... :100:[]` is transmitted to the function `sum`. In the second defining equality for `sum` the argument `(n:ns)` denotes a list, in which the first element is the number `n`, and its "tail" is a list of the numbers `ns`:

```
sum 1:2:3: ... :100:[] = 1 + sum 2:3: ... :100:[]
                       ...
    = 1 + 2 + 3 + ...  + 100 + sum[]
    = 1 + 2 + 3 + ...  + 100 + 0 (because sum [] = 0)
```

Write this function in terms of abstractions, using `letrec`:

```
letrec
     SumInts
       [m].letrec
            count = [n].IF (> n m) NIL
                           (cons n (count (+ n 1) ))
            in sum (count 1)
       sum = [ns].IF (= ns NIL) 0
                     (+ (head ns)(sum (tail ns)) )
  in SumInts 100
```

In the above: `NIL` is an empty list `[]`, `head` is the function, returning the first element of a list, `tail` is the function, returning the tail of a list (a list without the first element). The variables `SumInts` and `sum` have the number 0, however, `SumInts` contains an inner `[n].`-abstraction with free variables `m` and `count`. It is necessary to execute the lambda-lifting algorithm and to "lift" these variables.

(1) The inner abstraction is the following:

```
[n].IF (> n m) NIL (cons n (count(+ n 1) ))
```

(2) Lift the variables `count` and `m` in this order (because bound in a source program variable `m` should be at the last place):

```
([count].[m].[n].IF (> n m) NIL
                    (cons n (count (+ n 1)))) count m
```

(3) Assign to the derived combinator the name `$count`:

```
$count count m n = IF (> n m) NIL
                      (cons n (count (+ n 1)))
```

(4) Replace an occurrence of the `[n].`-abstraction in the program by the construction `$count count m n`:

```
$count count m n = IF (> n m) NIL
                      (cons n (count (+ n 1)))
-----------------------------------------------
letrec
    SumInts
         = [m].letrec
              count = $count count m
              in sum (count 1)
    sum = [ns].IF (= ns NIL) 0
            (+ (head ns)(sum (tail ns)))
in SumInts 100
```

(5) In the expressions `SumInts` and `sum` there are no inner abstractions, their level is equal to 0, hence, they are supercombinators. By immediate applying the lifting to these expressions, and adding the supercombinator `$Prog`, obtain the final result:

```
$count count m n = IF (> n m) NIL
                      (cons n(count (+ n 1)))
$sum ns = IF (= ns NIL) 0 (+ (head ns)
                             ($sum (tail ns)))
$SumInts m = letrec count = $count count m
         in $sum (count 1)
$Prog = $SumInts 100
-------------------------------------------
$Prog
```

Exercise 16.7. Compile the program, which applies the function `f = SQUARE` to each of the elements of a list of the natural numbers from 1 to 5:

```
----------------------------------------
apply m = fold SQUARE (constr 1)
          where constr n = [], n > m
                         = n:constr(n + 1)
fold f[] = []
fold f(n:ns) = fn:fold f ns
```

16.1.8 Other ways of lambda-lifting

An approach, used in the previous subsections, is not the only way of a lambda-lifting of the recursive functions. There is an algorithm, which generates the supercombinators for the data structures rather than the functions. Assume, that there is a program, containing a recursive function `f` with the free variable `v`:

```
-------------------------------------
( ...
    letrec f = [x].( ...  f ... v ... )
    in ( ... f ... )
... )
```

The recursive combinator `$f` is generated by `f`, however, with this, an abstraction is generated for the variable `v`, rather than for the function `f`: all the occurrences of `v` are replaced by `$f v`. The replacement results in:

```
-------------------------------------
$f v x = ... ($f v) ... v ...
( ...
    ( ... ($f v) ... )
  ... )
```

Consider an execution of this algorithm, using the example from subsection 16.1.6. A source program is as follows:

```
-------------------------------------------------
letrec
     SumInts
          [m].letrec
               count = [n].IF (> n m) NIL
                       (cons n (count (+ n 1)) )
               in sum (count 1)
          sum
               = [ns].IF (= ns NIL) 0
                         (+(head ns)(sum(tail ns)) )
     in SumInts 100
```

Lift the `[n].`-abstraction, abstracting the free variable `m`, but not the `count`, and replacing all the occurrences of `count` by the expression `($count m)`:

```
-------------------------------------------------
$count m n = IF (> n m) NIL
                        (cons n ($count m (+ n 1)))
letrec
    SumInts = [m].sum($count m 1)
    sum = [ns].IF (= ns NIL) 0
                  (+(head ns)(sum(tail ns)))
    in
         SumInts 100
```

There are two calls of `count` in the initial program: in the `[n].`-abstraction and in the definition of `SumInts`; both of these calls are replaced by `($count m)`. Now it is clear, that both `SumInts`, and `sum` are the supercombinators, hence, they could be lambda-lifted:

```
$count m n = IF (> n m) NIL
                        (cons n ($count m (+ n 1)))
$sum ns = IF (= ns NIL) 0
                        (+(head ns)($sum(tail ns)))
$SumInts m = sum ($count m 1)
$Prog = $SumInts 100
-------------------------------------------------
$Prog
```

A main advantage of this method relatively the previous one is as follows. In the example from subsection 16.1.6 a recursive call of `$count` in the supercombinator `$count` was performed via its parameter, denoted by `count`. In the new method a call of the supercombinator `$count` is done immediately. The compiler, built on this method, works more efficiently.

Exercise 16.8. Try to make a compiling of the program of exercise 16.7 by the method given above.

16.1.9 Full lazyness

Consider the function `f = [y].+ y (sqrt 4)`, where `sqrt` is the function of square root. Every time, when this function is applied to its argument, the subexpression `(sqrt 4)` is to be newly evaluated one more time. However, independently on a value of the argument `y`, the expression `(sqrt 4)` is reduced to `2`. Hence, it would be desirable not to perform repeatedly an evaluation of these constant expressions, but, evaluating it once, use the saved result.

Example 16.11. Consider the program:

```
--------------------
f = g 4
g x y = y + (sqrt x)
(f 1) + (f 2)
```

Its notation in terms of the lambda-expressions gives:

```
----------------------------
letrec f = g 4
    g = [x].[y].+ y (sqrt x)
    in + (f 1) (f 2)
```

In evaluating this expression the following result is obtained:

```
+ (f 1) (f 2) -->
--> + (. 1) (. 2)
       .-----.--->  (([x].[y].+ y (sqrt x)) 4)
-->  + (. 1)(. 2)
        .----.---> ([y].+ y (sqrt 4))
-->  + (. 1)(+ 2 (sqrt 4))
        .--------> ([y].+ y (sqrt 4))
-->  + (. 1)4
        .--------> ([y].+ y (sqrt 4))
-->  + (+ 1 (sqrt 4))4
-->  + (+ 1 2)4
-->  + 3 4
-->  7
```

In this example the subexpression `(sqrt 4)` is evaluated twice, on every application the expression `[y].(sqrt 4)` is treated as generated, in dynamics, constant subexpression of the `[y].`-abstraction. The same effect is observed when the supercombinators are used. The considered expression is compiled as follows:

```
$g x y = + y (sqrt x)
$f = $g 4
$Prog = + ($f 1)($f 2)
----------------------
$Prog
```

The reduction of this is the following:

```
$Prog -->  + (. 1)(. 2)
                .----.---> ($g 4)
-->  + (. 1)(+ 2 (sqrt 4))
        .---> ($g 4)
-->  + (. 1) 4
        .---> ($g 4)
-->  + (+ 1 (sqrt 4))4
-->  + (+ 1 2)4
-->  + 3 4
  7
```

And in this case the subexpression `(sqrt 4)` is evaluated twice as well. After writing these examples, having an introductory nature, give a formulation of the main problem, to overcome which the efforts are needed:

> after binding all its variables, every expression should be evaluated, at most, once.

This property of evaluation the expressions is called as *full lazyness*.

Exercise 16.9. Consider the following program:

```
f = g 2
g x y = y * (SQUARE x)
(f 3) * (f 1)
```

a) Write this program in terms of abstractions and evaluate.
b) Compile this program and evaluate.

16.1.10 Maximal free expressions

To get the full lazyness there is no necessity to make the evaluation of those expressions, which contain no (free) occurrences of the formal parameter.

Definition 16.2. An expression E is called the *proper* subexpression of F, if and only if E is a subexpression of F and E is not the same as F.

Definition 16.3. Subexpression E of the abstraction is assumed *free* in the lambda-abstraction L, if all the variables of E are free in L.

Definition 16.4. . A *maximal free expression*, or MFE, in L is such a free expression, that is not a proper subexpression of the other free expression in L.

Example 16.12. In the abstractions below the MFEs are underlined:

$$\begin{array}{ll} (1) & ([x].\underline{sqrt\ x}), \\ (2) & ([x].x\ \underline{(sqrt\ 4)}\ ) \\ (3) & ([y].[x].\underline{+(*\ y\ y)}\ x). \end{array}$$

To obtain a full lazyness, the maximal free abstractions are not to be evaluated in performing the β-reductions.

Example 16.13. Turn back to a function from the example 16.11:

```
--------------------
letrec f = g 4
    g = [x].[y].+ y (sqrt x)
    in + (f 1) (f 2)
```

A sequence of reductions begins, as in this example:

```
+ (f 1)(f 2) -->
--> + (. 1)(. 2)
       .----.---> (([x].[y].+ y (sqrt x)) 4)
-->  + (. 1)(. 2)
        .----.---> ([y].+ y (sqrt 4))
```

(here: the expression `(sqrt 4)` is MFE in `[y].`-abstraction, hence, in applying the `[y].`-abstraction to its argument `(sqrt 4)` it should not evaluate:

```
-->  + (. 1)(+ 2 .)
        .--------.---> ([y].+ y (sqrt 4)) )
```

Evaluation has a pointer to `(sqrt 4)` in the body of abstraction:

```
+ (. 1)(+ 2 .)
   .--------.--->([y].+ y 2)
--> + (. 1)4
       .---> ([y].+ y 2)
--> + (+ 1 2)4
--> + 3 4
 7
```

In this case `(sqrt 4)` is evaluated only once.

Exercise 16.10. Evaluate the expression from exercise 16.9, using MFE.

16.1.11 Lambda-lifting with MFE

An algorithm, using MFE, is distinct from the lambda-lifting algorithm above in that, it does not make the abstractions of variables, but of MFE. Turn back to an example under consideration. A function `g` contains the abstraction:

```
[x].[y].+ y (sqrt x)
```

Give an exemplifying of the algorithm in this situation.

(1) The most inner abstraction is

```
[y].+ y (sqrt x).
```

(2) The expression `(sqrt x)` is MFE. Shift MFE as an extraparameter:

```
([sqrtx].[y].+ y (sqrtx))(sqrt x),
```

where `sqrtx` is a name of extraparameter. Substitute the obtained expression into source abstraction:

```
[x].([sqrtx].[y].+ y sqrtx)(sqrt x).
```

(3) Assign a name to obtained supercombinator:

```
$g1 = [sqrtx].[y].+ y sqrtx
---------------------------
[x].$g1 (sqrt x)
```

(4) An available `[x].`-abstraction is also a supercombinator, that is named and that is corresponded to a compiled code:

```
$g1 sqrtx y = + y sqrtx
$g x = $g1 (sqrt x)
$f = $g 4
$Prog = + ($f 1)($f 2)
----------------------
$Prog
```

The additional supercombinator is obtained, because a possibilty to use η-reductions is lost due to abstracting on `(sqrtx)` instead abstracting on `x`.

However, this effect is compensated by full lazyness. Hence,

> to obtain full lazyness it is needed to abstract MFE, but not the free variables, using the generated abstractions as extraparameters.

The algorithm above is a fully lazy lambda-lifting.

Exercise 16.11. Make a fully lazy lambda-lifting of a program, considered in exercise 16.9.

16.1.12 Fully lazy lambda-lifting with *letrec*

Consider the program:

```
-------------------------------------
let f = [x].letrec fac = [n].( ...  )
               in + x (fac 100)
in
    + (f 3)(f 4)
```

The algorithm from subsection 16.1.6 compiles it in:

```
$fac fac n = ( ...  )
$f x = letrec fac = $fac fac
          in + x (fac 100)
+ ($f 3)($f 4)
```

The function `fac` is locally defined in the body of function `f` and, hence, the expression `(fac 100)` cannot be lifted as MFE from a body of `f`. This means, that `(fac 100)` will be evaluated every time, when `$f` is executed. Thus, a property of full lazyness is lost.

The overcoming of this is rather easy: it is enough to note, that the definition of `fac` does not depend on `x`, hence, `letrec` for `fac` can be lifted:

```
-------------------------
letrec
    fac = [n].( ...  )
    in let
    f = [x].+ x (fac 100)
in
    + (f 3)(f 4)
```

Now nothing prevents the applying of fully lazy lifting, that will generate a fully lazy program:

```
$fac n = ( ...  )
$fac100 = $fac 100
$f x = + x $fac100
$Prog = + ($f 3)($f 4)
----------------------
$Prog
```

In this case some optimizing is done: an expression, which has no free variables, is not abstracted, but is named and became the supercombinator – `$fac100`.

Thus, a strategy is implemented by two stages:

1) shifting `letrec-` (and `let-`) definitions as "far" as possible,

2) using a fully lazy lambda-lifting.

Exercise 16.12. Execute a fully lazy lambda-lifting of the program:

```
let
    g = [x].letrec el = [n].[s].(IF (= n 1)(head s)
                                    (el (- n 1)(tail s)) )
            in (cons x (el 3 (A,B,C)))
in
    (cons (g R)(g L))
```

(here: `R` and `L` are the constants).

16.1.13 Compound example

This subsection contains a more detailed example exposing fully lazy lambda-lifting[1]:

```
-----------------------------------------------
SumInts n = foldl + 0 (count 1 n)
count n m = [],   n > m
```

[1] An advanced study of lazy evaluations with supercombinators see in [105]. This book includes the abstract machine that supports all the spectrum of possibilities.

```
count n m = n:count (n + 1) m
fold op base [] = base
foldl op base (x:xs) = foldl op (op base x) xs
```

This program in terms of abstractions is written as:

```
-----------------------------------------------------
letrec
     SumInts = [n].foldl + 0 (count 1 n)
     count = [n].[m].IF (> n m) NIL
                        (cons n (count (+ n 1) m))
     foldl = [op].[base].[xs].IF (= xs NIL) base
             (foldl op (op base (head xs))(tail  xs))
in SumInts 100
```

In this program:

(1) an inner abstraction is `([xs]. ... )`,

(2) maximal free expressions are the expressions `(fold op)`, `(op base) base`.

They are shifted as extraparameters `p`, `q` and `base` respectively:

```
$R1 p q base xs = IF (= xs NIL) base
                     (p (q (head xs)) (tail xs))
-----------------------------------------------------
letrec
     SumInts = [n].foldl + 0 (count 1 0)
     count = [n].[m].IF (> n m) NIL
                          (cons n (count (+ n 1) m))
     foldl = [op].[base].$R1 (foldl op) (op base) base
in
     SumInts 100
```

In this program an inner abstraction is '`[base].`'. Maximal free expressions are the expressions `($R1(foldl op))` and `op`, that are shifted as `r` and `op` respectively:

```
$R1 p q base xs = IF (= xs NIL) base
         (p (q (head xs))(tail xs))
$R2 r op base = r (op base) base
-------------------------------------------------
letrec
     SumInts = [n].foldl + 0 (count 1 n)
     count = [n].[m].IF (> n m) NIL
                        (cons n (count (+ n 1) m))
     foldl = [op].$R2($R1 op)op
in
     SumInts 100
```

(3) all the definitions of `letrec` are supercombinators, because a lifting of all the inner abstractions is done. With all of these notions, a final result is as follows:

```
$SumInts n = $foldlPlus0 ($count1 n)
$foldlPlus0 = $foldl + 0
$count1 = $count 1
$count n m = IF (> n m)NIL(cons n ($count (+ n 1) m))
$foldl op = $R2 ($R1 ($foldl op))op
$Prog = $SumInts 100
$R1 p q base xs = IF (= xs NIL) base
                     (p (q (head xs))(tail xs))
$R2 r op base = r (op base) base
-------------------------------------------------------
$Prog
```

Concluding a consideration, note that in this case the parameter `op` cannot be eliminated in `$foldl`, because it is used twice in a right part.

16.2 Task

Task 16.1. Write the following definition in initial language using supercombinators:

$$el\ n\ s = if\ \ n = 1\ \ then \quad (hd\ s)$$
$$else\ \ el\ \ (n-1)\ \ (tl\ s).$$

The function el selects n-th element form a sequence s.

Solution. In terms of λ-calculus a definition of the function has the following form:

$$el = Y(\lambda el.\lambda n.\lambda s.if\ (=\ n\ 1)\ (hd\ s)(el\ (-\ n\ 1)\ (tl\ s))). \qquad (el)$$

Remind an algorithm of translation the λ-expressions into applicative form. Take an arbitrary λ-expression $\lambda V.E$.

$SC{-}1$. A body of the initial expression is transformed into the applicative form by a recursive call of the compiler. This results in: $\lambda V.E'$.

$SC{-}2$. The variables, which are free in the λ-expression, are identified by the letters P, Q, ..., R, next the λ-expression receives a prefix 'λ', bounding all these variables. The result is the following:

$$\lambda P.\lambda Q.\ldots.\lambda R.\lambda V.E'.$$

The resulting expression is wittingly a supercombinator, because E' is an applicative form, all the free variables P, Q, ..., R in which are bound.

$SC{-}3$. Denote the generated combinator by $alpha$ and assign to it a definition (defining equality):

$$alpha\ P\ Q\ \ldots\ R\ V \rightarrow E'.$$

An initial λ-expression is replaced by the form

$$(alpha\ P\ R\ \ldots\ R).$$

In a connection with expression E, the application can be reduced. The expression E is related to V, and every free variable has its own value in E'. Hence, the applicative form is actually equal to the initial λ-expression. Now a step-by-step translations will be determined.

1) Consider the most inner λ-expression:

$$\lambda s.if\ (=\ n\ 1)\ (hd\ s)\ (el\ (-\ n\ 1)\ (tl\ s)).$$

Its free variables are n and el, hence the combinator $alpha$ is introduced by a definition:

$$alpha\ n\ el\ s \rightarrow if\quad \begin{array}{l}(=\ n\ 1)\ (hd\ s)\\ (el\ (-\ n\ 1)\ (tl\ s)).\end{array} \qquad (alpha)$$

2) Replace all the λ-expressions by $(alpha\ n\ el)$, therefore,

$$el = Y(\lambda el.\lambda n.alpha\ n\ el).$$

3) Repeat the steps 1) and 2) for $\lambda n.alpha\ n\ el$. Introduce the combinator $beta$ by a definition:

$$beta\ el\ n \rightarrow alpha\ n\ el, \qquad (beta)$$

from which follows, that

$$el = Y(\lambda el.beta\ el).$$

4) Repeat the steps 1) and 2) for $(\lambda el.beta\ el)$. Introduce the combinator $gamma$ by a definition:

$$gamma\ el \rightarrow beta\ el, \qquad (gamma)$$

from which we obtain, that

$$el = Y\ gamma.$$

Answer. $el = Y\ gamma.$

Test.

1. What are the aims of using the supercombinators in implementing a reduction?

2. What are the features of a languages of the constant applicative forms (CAF)?

3. What are the features of the combinators S, K, I, that allow their direct using in a GR-machine (a machine of graph reduction)?

Exercise 16.13. Using the supercombinators, write the expression:

$$fac\, n \;=\; if\ n = 0\ then\ 1\ else\ n \;\times\; (fac(n-1)).$$

16.3 Answers to exercises

16.1
`1 0` contains no free variables (see Df. 16.1, p.(1)) and abstraction symbols (see Df. 16.1, p.(3)); `[x].+ x 1` contains a variable `x`, which is bound (see Df. 16.1, p.(1)); `+ x 1` contains no abstractions (see Df. 16.1, p.(2)); `[f].f([x].+ x x)` contains no free variables (see Df. 16.1, p.(1)), `[x].+ x x` is a supercombinator (see Df. 16.1, p.(2)).

16.2
`[x].x y z` contains free variables `y` and `z`, point (1) of Definition 16.1 is violated; `[x].[y].+ (+ x y) z` contains free variable `z`.

16.3
For instance, `[x].[y].x y([z].z x)`.

16.4

The expression `$XY 5` cannot be evaluated, because it is not a redex; `$XY 5 7` can be evaluated, `$XY 3 4 7` can be evaluated.

16.5

1) `([x].([y].- y x)x)5`:

(1) the innermost abstraction is `[y].- y x`;

(2) lift `x` as an extraparameter `([x].[y].- y x)x` and substitute it into initial program `([x].([v].[y].- y v) x x)5`;

(3) assign to this supercombinator the name `$Y`:

```
$Y v y = - y v
--------------
([x].$Y x x)5
```

(4) `[x].`-abstraction is also a supercombinator, which can be named and assigned to the compiled code:

```
$Y v y = - y v
$X x = $Y x x
--------------
$X 5
```

The derived program is executed as follows:

```
$X 5 = $Y 5 5 = - 5 5 = 0.
```

2) `([z].+ z (([x].([y]. * y x) x) 4)2`:

(1) inner abstraction: `[y]. * y x`;

(2) lift `x` as an extraparameter:

```
([x].[y]. * y x) x
([z].+ z(([x].([w].[y]. * y w) x x)4))2;
```

(3)

```
$Y w y = * y w
------------------------
([z].+ z([x].$Y x x) 4) 2
```

(4) `[x].`-abstraction is a supercombinator:

```
$Y w y = * y w
$X x = $Y x x
------------------
([z].+ z ($X 4)) 2
```

Now it is obvious that `[z].`-abstraction is a supercombinator:

```
$Y w y = * y w
$X x = $Y x x
$Z z = + z ($X 4)
-----------------
$Z 2
```

An execution of the program:

```
$Z 2 = + 2 ($X 4) = + 2 ($Y 4 4) =
     = + 2 (* 4 4) = + 2 16 = 18.
```

16.6

`inf` is at a level 0 and contains no inner abstractions. Hence, `inf` is already a supercombinator:

```
$inf v = letrec vs = cons 0 vs in vs
$Prog = $inf 4
-----------------------------------
$Prog
```

16.7

Write this program in terms of abstractions:

```
-----------------------------------------------------
letrec apply = [m].letrec constr = [n].(IF > n m) NIL
                              (cons n (constr (+ n 1)))
    in fold SQUARE (constr 1)
       fold = [f].[ns].IF (= ns NIL) NIL
           (cons (f (head ns)) (fold (f (tail ns))) )
    in apply 5
```

(1) inner abstraction has a form:

```
[n].IF (> n m) NIL (cons n (constr (+ n 1)));
```

(2) lift the variables `constr` and `m` in this order:

```
[constr].[m].[n].IF (> n m) NIL
                (cons n (constr (+ n 1))) constr m;
```

(3) assign to this combinator the name `$constr`:

```
$constr constr m n = IF (> n m) NIL
                        (cons n(constr(+ n 1)));
```

(4) replace an occurrence of `[n].`-abstraction in the program by the expression `($constr constr m n)`;

```
$constr constr m n = IF (> n m) NIL
                        (cons n (constr (+ n 1)))
-------------------------------------------------
letrec
     apply = [m].letrec
                         constr = $constr constr m
          in fold SQUARE (constr 1)
     fold = [f].[ns].IF (= ns NIL) NIL
        (cons (f (head ns)) (fold f (tail  ns)) )
in apply 5
```

(5) the expressions `apply` and `fold` are supercombinators:

```
$constr constr m n = IF (> n m) NIL
                         (cons m (constr(+ n 1)))
$fold f ns = IF (= ns NIL) NIL
                (cons (f (head ns))
                      (fold (f (tail ns))))
$apply m = letrec constr = $constr constr m
          in $fold SQUARE (constr 1)
$Prog = $apply 5
-----------------------------------------------
$Prog
```

16.8

Lift `[n].`-abstraction, abstracting the variable `m`, but not the `constr`, and replace all the occurrences of `constr` by `($constr m)`:

```
$constr m n = IF (> n m) NIL
                         (cons n ($constr m (+ n 1)))
-----------------------------------------------------
letrec
     apply = [m].fold (SQUARE) ($constr m 1)
     fold = [f].[ns].IF (= ns NIL) NIL
           (cons (f (head ns)) (fold (f (tail ns))) )
in apply 5
```

In this case both `apply`, and `fold` are the supercombinators:

```
$constr m n = IF (> n m) NIL
                         (cons n ($constr m (+ n 1)))
$fold f ns = IF (= ns NIL) NIL (cons
                                (f (head ns))
                                ($fold f (tail ns)) )
```

```
$apply m = $fold SQUARE ($constr m 1)
$Prog = $apply 5
-----------------------------------------------------
$Prog
```

16.9

a) Notation as the abstraction is the following:

```
--------------------------------
letrec f = g 2
    g = [x].[y]. * y (SQUARE x) in * (f 3)(f 1)
```

Evaluation of expression:

```
* (f 3)(f 1) -->
-->  (. 3)(. 1)
       .----.---> ([x].[y]. * y (SQUARE x))2
--> * (. 3)(. 1)
        .----.---> ([y]. * y (SQUARE 2))
--> * (. 3)(* 1 (SQUARE 2))
        .---> ([y]. * y (SQUARE 2))
--> * (. 3) 4
        .---> ([y]. * y (SQUARE 2))
--> * ( * 3 (SQUARE 2)) 4
--> * 12 4
--> 48.
```

b) This expression is compiled into:

```
$g x y = * y (SQUARE x)
$f = $g 2
$Prog = * ($f 3) ($f 1)
-----------------------
$Prog
```

A sequence of reductions:

```
$Prog -->  (* (. 3)(. 1))
                .-----.---> ($g 2)
      --> * (. 3)(* 1 (SQUARE 2))
              .---> ($g 2)
      --> * (. 3) 4
              .--->  ($g 2)
      -->  * (* 3 (SQUARE 2)) 4
      -->  * (* 3 4) 4
      -->  * 12 4
      -->   48.
```

16.10

```
* (f 3)(f 1) -->
-->  * (. 3)(. 1)
       .----.---> (([x].[y].* y (SQUARE x)) 2)
--> * (. 3)(* 1 .)
      .--------.---> ([y].* y (SQUARE 2))
--> * (. 3) 4 (* 1 .)
      .-----------.---> ([y].* y 4)
--> * (. 3) 4
      .---> ([y].* y 4)
--> * (* 3 .) 4
          .--->  4
--> * 12 4
-->   48
```

The expression `--> (SQUARE 2)` is evaluated once.

16.11

The function g contains an abstraction:

```
[x].[y]. * y (SQUARE x)
```

Next:

(1) the innermost abstraction is

```
[y]. * y (SQUARE x);
```

(2) `(SQUARE x)` is MFE, which is shifted as an extraparameter:

```
([SQUAREx].[y]. * y (SQUAREx)) (SQUARE x)
```

Substitute this expression into the abstraction:

```
[x].([SQUAREx].[y]. * y (SQUAREx)) (SQUARE x)
```

(3) assign to obtained supercombinator the name:

```
$g1 = [SQUAREx].[y]. * y SQUAREx
---------------------------------
[x].$g1 (SQUARE x)
```

(4) `[x].`-abstraction is also a supercombinator, which is named by `$g` and assigned to the compiled code:

```
$g1 SQUAREx y = * y SQUAREx
$g x = $g1 (SQUARE x)
$f = $g 2
$Prog = * ($f 3)($f 1)
------------------------------
$Prog
```

16.12

This program is compiled into:

```
$el el n s = (IF (= n 1)(head s)(el (- n 1)(tail s)))
$g x = letrec el = $el el
                in (cons x (el 3 (A,B,C)) )
-----------------------------------------------------
(cons ($g R)($g L))
```

Since a definition of `el` does not depend on `x`, the `letrec` can be shifted for `el`:

```
-----------------------------------------------------
letrec el = [n].[s].(IF (= n 1)(head s)
                                (el(- n 1)(tail s)))
in let g = [x].cons x (el 3 (A,B,C))
in (cons (g R) (g L))
```

As a result of applying the lambda-lifting obtain:

```
$el n s = (IF (= n 1)(head s)(el (- n 1)(tail s)))
$el3 (A,B,C) = $el 3 (A, B, C)
$g x = cons x $el3 (A,B,C)
$Prog = cons ($g R)($g L)
-----------------------------------------------------
$Prog
```

New combinators, the parameters of which are the MFEs, are generated in a process of compiling a programm. In other words, compiler performs the *assigning* by that way of indicating the objects in a program source code, which is implemented, generating the individual objects. Since the generated objects are the combinators, then it is safety to add them to shell system as the new instructions. All the generated combinators give rather individualized representation of the source program: this is a system of objects.

Possibly, it would be better to explain in terms of assignment of a shell system (λ-calculus) with the set of MFEs. Then, as a *concept*, λ-calculus gives a system of *individuals* (of compiled programs): they form an equivalency class (are converted to each other) relatively the criteria of an optimum.

Chapter 17

Lazy implementation

Whenever the objects are generated in dynamics "on fly", then an efficiency of the resulting code could be lost for the multiple evaluations of the same object. The mechanisms of lazy evaluation allow to avoid this inefficiency: if the value of an object is obtained once, then namely this value will be used in the future.

17.1 Tasks

Task 17.1. For a sample definition

$$el\ n\ s\ =\ \underline{if}\ n = 1\ then\ (hd\ s)\ else\ el\ (n-1)\ (tl\ s)\ \underline{fi},$$

where the function el returns n-th element of a list (finite sequence) s, find the supercombinators, resulting in a fully lazy implementation, and perform the particular case $el\ 2$.

Task formulation. Some definitions are to be introduced. As turns out, the expressions which are repeatedly evaluated can be easily identified. This is true for any subexpression of λ-expression, which does not depend on bound variable. Such expressions are called free expressions of λ-expression (similar to the notion of free variable).

Free expressions that are not a part of any other bigger free expression are called maximal free expressions of a λ-expression (MFE).

Consider a scheme of translation. The maximal free expressions of every λ-expression are transformed into the parameters of the corresponding combinator. Consider a scheme of translating maximal free expressions into the parameters of corresponding combinator.

Note. At first, we establish that this translation scheme is valid, i.e. the actual combinators are derived and every λ-expression is replaced by the applicative form. Consider an applicability of this new scheme to the λ-expression, which body is an applicative form. Such a combinator that is so generated should conform the definition, because its body is to be an applicative form and is to contain no free variables. Body of a combinator will wittingly be an applicative form because it is, in turn, generated from the applicative form (from a body of source λ-expression) by substituting new parameter names instead of some of the expressions. It cannot contain any free variables, because a free variable should be a part of some maximal free expression and, therefore, will be eliminated as a part of parameter. All of this confirms that an actual combinator is constructed. The final result which replaces a source λ-expression, is a new combinator, that is applied to maximal free expressions, each of them came already into being as an applicative form.

Solution. Turn back to the initial task.

lazy-1. Let maximal free expressions for the initial expression

$$\lambda s.el\ n\ s = \lambda s.if\ (=\ n\ 1)(hd\ s)(el(-\ n\ 1)(tl\ s))$$

be

$$p \equiv (if(=\ n\ 1)) \quad \text{and} \quad q \equiv (el(-\ n\ 1)).$$

Therefore, a new combinator *alpha* is defined by

$$alpha\ p\ \ q\ s \rightarrow p\ (hd\ s)\ (q\ (tl\ s)).$$

Note, that in this case

$$\lambda s.alpha\ p\ \ q\ s = alpha\ p\ \ q\ \ \rightarrow \lambda s.p\ (hd\ s)\ (q\ (tl\ s)).$$

Since, now

$$\begin{array}{rcl} \lambda s.el\ n\ s & = & \lambda s.if\ (=\ n\ 1)(hd\ s)(el(-\ n\ 1)(tl\ s)) \\ & = & \lambda s.p\ (hd\ s)\ (q\ (tl\ s)) = alpha\ p\ \ q = el\ n, \end{array}$$

then

$$el\ n = alpha\ \underbrace{(if\ (=\ n\ 1))}_{p}\ \underbrace{(el\ (-\ n\ 1))}_{q}.$$

From this the following result is derived: a definition of the function el is the expression

$$el = Y\ (\lambda el.\lambda n.alpha\ (if\ (=\ n\ 1))\ (el\ (-\ n\ 1))).$$

Continuing this process, we obtain the additional combinators $beta$ and $gamma$, defined by:

$$\begin{array}{rcl} beta\ el\ n & \rightarrow & \underbrace{alpha\ (if\ (=\ n\ 1))\ (el(-\ n\ 1))}_{el\ n}, \\ gamma\ el & \rightarrow & beta\ el, \end{array}$$

from where conclude, that the expression el is equal to the expression $(Y\ gamma)$, i.e. $el = Y\ gamma$, that coincides with the previous result[1].

lazy−2. Consider now a particular case, when $(el\ 2)$ is applied:

$$\begin{array}{rl} el\ 2 & \rightarrow \underbrace{(Y\ gamma)}_{el}\ 2 \\ & \rightarrow gamma\ el\ 2 \\ & \rightarrow beta\ el\ 2 \\ & \rightarrow alpha\ (if\ (=\ 2\ 1))\ (el\ (-\ 2\ 1)). \end{array}$$

[1] Indeed, $gamma\ el = el$, from where $el = (\lambda f.(gamma\ f))\ el$. By a fixed point theorem (see Theorem 9.1 on p. 123) $el = Y(\lambda f.(gamma\ f)) = Y\ gamma$.

Any time, when $(el\ 2)$ is used, the same copies of free variables are involved, hence, they are evaluated only once. In fact, the following reduction is obtained:

$$el\ 2 \rightarrow alpha\ if\text{-}FALSE\ (alpha\ if\text{-}TRUE\ (el\ (-\ 1\ 1))).$$

In more details:

$$\begin{aligned} el\ 2 &\rightarrow alpha\ (if\ (=\ 2\ 1))\ (el\ (-\ 2\ 1)) \\ &\rightarrow alpha\ (if\text{-}FALSE)\ (el\ 1) \\ &\rightarrow alpha\ (if\text{-}FALSE)\ (alpha\ (if\ (=\ 1\ 1))\ (el(-\ 1\ 1))) \\ &\rightarrow alpha\ (if\text{-}FALSE)\ (alpha\ (if\text{-}TRUE)\ (el\ (-\ 1\ 1))) \end{aligned}$$

The scheme considered above leads to the suboptimal combinators.

Exercises

Exercise 17.1. For the expression

$$fac\ n\ =\ if\ n = 0\ then\ 1\ else\ n \times (fac(n-1))$$

perform a fully lazy implementation, using the supercombinators, and execute $fac\ 3$.

Chapter 18

Permutation of parameters

Using of combinators opens the abilities to generate optimized program code, in passing the synthesis of resulting object making an analysis of order of possible replacement of the formal parameters by the actual ones.

18.1 Task

Task 18.1. For the initial definition:

$$el = Y\ (\lambda el.\lambda n\lambda s.IF\ (=\ n\ 1)\ (hd\ s)\ (el\ (-\ n\ 1)\ (tl\ s)))$$

derive an expression with optimal (by two criteria) ordering of supercombinator parameters.

Task formulation. Remind, that two main criteria of the optimal ordering of supercombinator parameters are the number of MFEs and maximal elimination of redundant parameters. The given definition should be successfully optimized by both of the criteria.

Note. For maximizing the length and minimizing the number of MFEs of the currently nested λ-expression, all the MFEs in λ-expression being compiled, that in the same time are free expressions

in the next nested λ-expression, should appear before MFE, having no such property. An optimal parameter ordering by this criterion can be formulated as follows. All E_i are free expressions of the λ-expression, which has to be compiled, however it can be free expression of one or more number of nested λ-expressions. The innermost λ-expression, in which the expression E_i is not free, will be called a *generating* λ-expression. If generating λ-expression E_i includes the generating λ-expression E_j, then in an optimal ordering E_i is a predecessor of E_j. From this it does not follow, that an optimal ordering is with a necessity unambiguously determined, because the expressions with the same generating λ-expression can occur in any order. Nevertheless, any ordering, conforming this condition, is optimal, as any other one.

Solution.

opt−1. First of all take into account the result of solving the task 17.1 on p. 211.

opt−2. Assume the applicative form

$$(alpha\ (hd\ s)\ n\ (tl\ s)),$$

in which the parameters of *alpha* can be arranged in any order. In case the immediately nested λ-expression binds n, then maximal free expressions (MFEs) of this form is as follows:

$$(alpha\ (hd\ s))\ \ (tl\ s).$$

In case a reordering of the form

$$(alpha\ (hd\ s)\ (tl\ s)\ n),$$

is performed, then MFE is unique:

$$(alpha\ (hd\ s)\ (tl\ s)).$$

opt−3. If, on the other hand, an immediately nested λ-expression binds s, then an optimal parameter ordering is

$$(alpha\ n\ (hd\ s)\ (tl\ s)),$$

from where the only MFE is $(alpha\ n)$.

opt−4. Having obtained an optimal ordering by one criterion, consider the next of them. As the "only" parameters the following ones

$$alpha,\quad beta,\quad el,\quad hd,\quad tl.$$

can be accepted.

A mission of the compiler is to arrange parameters so that to achieve a maximal elimination of redundant parameters. The only case in which compiler has a choice is when a combinator is directly defined as a call of other one. As an example consider the reduction:

$$beta\ p\ q\ r\ s \rightarrow alpha\ \ldots\ s\ \ldots\ .$$

There are no redundant parameters besides the last one, but the last parameter of a combinator should always be a bound variable of the λ-expression, from which a derivation was done. In this case parameter s is a bound variable of the λ-expression, that immediately includes $alpha$. If the parameters have already been optimally ordered by the rules above, then all the parameters, in which s participates, are shifted to the end of its list of parameters. If there is the only such a parameter, and this is s, then s becomes a redundant parameter $beta$ and can be eliminated. By this reason a call $alpha$ should be determined in form of

$$(alpha\ E_1\ \ldots\ E_n\ s),$$

where s has no occurrences in any of the expressions E_i.

This means, that all E_i are free in *beta*, that spreads over $(alpha\ E_1\ \ldots\ E_n\ s)$. Hence, in fact, if there are any E_i, then *beta* should be defined by reduction

$$beta\ p\ s \to p\ s,$$

where p corresponds to $(alpha\ E_1\ \ldots\ E_n\ s)$.

If *alpha* has the only parameter s, then *beta* should be defined by

$$beta\ s \to s.$$

In the first case *beta* equals to the combinator I. The λ-expression, generating *beta*, is replaced by an application

$$(beta\ (alpha\ E_1\ \ldots\ E_n\ s)),$$

i.e. by $(I\ (alpha\ E_1\ \ldots\ E_n\ s))$. Besides that, *beta* can be entirely omitted, and λ-expression is replaced immediately by $(alpha\ E_1\ \ldots\ E_n\ s)$.

In the second case *beta* is equal to *alpha*. It can be seen that optimal ordering, obtained by the second criterion, also conforms to the first one and, besides that, significantly simplifies a mission to find the redundant parameters.

opt−5. The considered expression *el* is defined by equality:

$$el = Y(\lambda el.\lambda n.\lambda s.IF\ (=\ n\ 1)\ (hd\ s)\ (el(-\ n\ 1)\ (tl\ s))).$$

The innermost λ-expression has two MFEs:

$$(IF\ (=\ n\ 1))\ \ (el\ (-\ n\ 1)).$$

Assume them as the parameters p and q. Both of these MFEs have the same generating λ-expression, hence, their ordering is immaterial, and *alpha* can be defined by reduction:

$$alpha\ p\ q\ s \to p\ (hd\ s)\ (q\ (tl\ s)),$$

therefore,

$$el = Y(\lambda el.\lambda n.alpha\ (IF\ (=\ n\ 1))\ (el(-\ n\ 1))).$$

It appears now, that the next λ-expression has the only MFE and this is el. Hence, $beta$ is defined by reduction

$$beta\ el\ n \rightarrow alpha\ (IF\ (=\ n\ 1))\ (el\ (-\ n\ 1)),$$

according which

$$el = Y(\lambda el.beta\ el).$$

Next, the reduction

$$gamma\ el \rightarrow beta\ el,$$

should be applied, and, since combinator $gamma$ is equal to combinator $beta$, then $gamma$ should not be generated.

Finally, obtain that $gamma$ is equal to $(Y\ beta)$.

Exercises

Exercise 18.1. Derive the expression with optimal ordering of super-combinators for the definition:

$$fac\ n\ =\ IF\ n = 0\ then\ 1\ else\ n \times (fac(n-1)).$$

Test

Try to give answers to the following questions.

1. Define the notion of 'lazy evaluation in λ-calculus' and 'full lazyness' in the language of CAFs and indicate a connection of them.

2. Determine a meaning of 'MFE of λ-expression'.
3. What is the final result of replacing the initial λ-expression in a fully lazy implementation of supercombinators?
4. What are the advantages of using the rules of optimization?
5. Give a formulation of two main criteria of optimization.
6. What are the consequences of the ordering of parameters of supercombinators?

Chapter 19

Immediate computations

In this chapter a method of evaluating expressions is revised using systematic construction of the set of both syntax and semantic equalities, that implement considered paradigm of object-oriented computations.

19.1 Task

Task 19.1. For the expression:

$$let\ x\ =\ plus\ in\ x\ (4, (x\ where\ x = 3));;$$

construct λ-expression and expression of a combinatory logic, and evaluate them.

Solution.

dc−1. Write the expression:

$$let\ x\ =\ plus\ in\ x\ (4, (x\ where\ x = 3));;$$

dc−2. Re-write it in a form of λ-expression:

$$M\ =\ (\lambda x.x(4,\ (\lambda x.x)3)) + .$$

dc–3. Make the preparations to translate this expression into a language of combinatory logic (CL), i.e. perform the currying:

$$N \;=\; (\lambda x.x\ 4\ ((\lambda x.x)3))\oplus,$$

where the symbol of addition '$\oplus$' is distinct from the symbol '$+$'. In the meanwhile put aside a discussion of their differences till the topics on the expression evaluation using categorical abstract machine will be considered.

dc–4. Remind that a procedure of translating into CL is defined by induction:

$$\begin{array}{rlcl} (i) & \lambda x.x & = & I, \\ (ii) & \lambda x.c & = & Kc,\ c \not\equiv x, \\ (iii) & \lambda x.PQ & = & S(\lambda x.P)(\lambda x.Q). \end{array}$$

Thus,

$$\begin{array}{rcl} N & = & (\lambda x.x\ 4((\lambda x.x)3))\oplus \\ & = & S(\lambda x.x\ 4)(\lambda x.((\lambda x.x)3))\oplus \\ & = & S(S(\lambda x.x)(\lambda x.4))(S(\lambda x.(\lambda x.x))(\lambda x.3))\oplus \\ & = & S(SI(K\ 4))(S(\lambda x.I)(K3))\oplus \\ & = & S(SI(K\ 4))(S(K\ I)(K\ 3))\ \oplus\ . \end{array}$$

dc–5. Application of this procedure results in

$$N = S(SI(K\ 4))(S(K\ I)(K\ 3))\oplus,$$

where $Sxyz = xz(yz),\ Kxy = x,\ Ix = x.$

dc–6. Using any of the ways to evaluate it, results in the number '7':

$$\begin{array}{l} S(SI(K4))(S(KI)(K3))\oplus \rightarrow \\ \qquad \rightarrow (SI(K\ 4)\oplus)(S(K\ I)(K\ 3)\oplus) \\ \qquad \rightarrow (I\oplus)(K\ 4\oplus)(S(K\ I)(K\ 3)\oplus) \\ \qquad \rightarrow \oplus(K\ 4\oplus)(S(K\ I)(K\ 3)\oplus) \\ \qquad \rightarrow \oplus 4(S(K\ I)(K\ 3)\oplus) \rightarrow \oplus 4(K\ I\oplus)(K3\oplus) \\ \qquad \rightarrow \oplus 4(I(K\ 3\oplus)) \rightarrow \oplus 4\ 3 \rightarrow 7. \end{array}$$

It is easy to verify this result by an immediate β-reduction of the λ-expression:

$$M = (\lambda x.x(4,(\lambda x.x)3))+ \to +(4,(\lambda x.x)3) \to +(4,3) \to 7,$$

(pay attention one more time to the using of the addition symbol: '+' is used instead of '$\oplus$').

Exercises

Exercise 19.1. Express by the combinators K and S the object I with a combinatory characteristic $Ia = a$.

Hint. Use the equality $I = \lambda z.z$. Derive the expression $\lambda z.z = (\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$ and remind that $K = \lambda xy.x$, $S = \lambda xyz.xz(yz)$.

Answer. $I = SKK$.

Exercise 19.2. For the expression:

$$let\ x\ =\ \pi/2\ in\ let\ z\ =\ sin\ in\ sqr(z\ x);;$$

construct the expression of combinatory logic and evaluate it.

Hint.

1. λ-expression is the following:

$$\begin{aligned}(\lambda x.(\lambda z.sqr(z\ x))sin)\pi/2 = \\ = (\lambda x.sqr(sin\ x))\pi/2, \qquad (\beta)\\ = sqr(sin\ \pi/2). \qquad (\beta)\end{aligned}$$

2. Use a fact that

$$f(gx) = (f \circ g)x = S(KS)K\ f\ g\ x.$$

Answer. $S(KS)K\ sqr\ sin\ \pi/2 = 1$.

Test

1. Write the combinatory characteristics of the standard combinators K, I, S, B, W, C.

2. What are the reasons to use combinatory logic to implement β-reductions?

Chapter 20

de Bruijn's encoding

In this chapter the method of rewriting the bound variables (formal parameters) is introduced, which allows to avoid the collisions of binding while replacing the formal parameters by the actual ones. This way of rewriting is called as *de Bruijn's encoding* and allows, in fact, use λ-calculus on the same rights as combinatory logic.

20.1 Tasks

Task 20.1. For the expression:

$$let\ x\ =\ plus\ in\ x\ (4, (x\ where\ x = 3));;$$

construct the λ-expression and expression of de Bruijn's encoding, and evaluate the last expression using $SECD$-machine.

Task formulation. It is known, that performing the λ-conversions leads to the collisions of variables. E.g., "direct" execution of β-reduction for $(\lambda xy.x)y$ could give $\lambda y.y$:

$$(\lambda xy.x)y = \lambda y.y,$$

that is completely inappropriate, because:

$$\begin{array}{rcl}(\lambda xy.x)y & \overset{(\alpha)}{=} & (\lambda uv.u)y \overset{(\beta)}{=} (\lambda v.y) \\ & \neq & (\lambda y.y) = I.\end{array}$$

Note, that in a closed term the significantly important knowledge of a variable is the depth of its binding, i.e. the number of symbols λ between the variable and its binding λ (excepting the last operator). Then the variable is replaced by the number, which, however, should be distinguished from the usual natural number. To distinguish the numbers, replacing the variables, from the usual natural numbers the first of them will be called de Bruijn's numbers. Now, e.g., for

$$P = \lambda y.(\lambda xy.x)y$$

de Bruijn's encoding is of a form:

$$\lambda.(\lambda\lambda.\underline{1})\underline{0}.$$

Say, the rule (β), when applied to this expression, results in $\lambda\lambda.\underline{1}$, and it is not necessary to transform $\lambda xy.x$ into $\lambda xv.x$, which eliminates the collision. Main topic is to describe a meaning of the expressions. This depends on the associations between the identifiers and their values, i.e. on an environment. Thus, evaluation of M is the function $\|M\|$, which associates the value with an environment. Consider the usual semantic equalities, where the application of a function to its argument is represented just by writing the symbol of function followed by the symbol of argument:

$$\begin{array}{rcl}\|x\|env & = & env(x), \\ \|c\|env & = & c, \\ \|(M\,N)\|env & = & \|M\|env\ (\|N\|env), \\ \|\lambda x.M\|env\ d & = & \|M\|env\ [x \leftarrow d],\end{array}$$

where:

$env(x)$	-	value of x in the environment env;
c	-	constant, denoting the value, also be called c, what is in accordance with the usual practice;
$env[x \leftarrow d]$	-	environment env, where x is replaced by the value d, i.e. a *substitution* of d in place of x in env has been done.

In general, de Bruijn's formalism can be considered by the same way as a combinatory logic with a suitable adaptation of the rules. To transit from the usual λ-expressions to encoding the variables by de Bruijn's numbers, consider the needed rules and agreements.

Let environment env be of form

$$(\ldots((),\ w_n)\ldots,\ w_0),$$

where the value w_i is associated with de Bruijn's number i. This assumption uses rather strict restrictions. The environments, where the evaluation of expressions is performed, are assumed bound by the structures, not by the arrays. This choice is closely linked to conforming the demands of efficiency. First of all, this choice leads to a simple machine description:

$$\begin{aligned}
\|\underline{0}\|(env, d) &= d,\\
\|\underline{(n+1)}\|(env, d) &= \|\underline{n}\|env,\\
\|c\|env &= c,\\
\|M\ N\|env &= \|M\|env(\|N\|env)\\
\|\lambda.M\|env\ d &= \|M\|(env, d).
\end{aligned}$$

The interest is not of the values themselves, but the values from point of view the evaluations they support. In a combinatory approach it is stressed, that the value of, e.g., $M\ N$ is a combination of the values of M and N.

The following three combinators are introduced:

$$\begin{array}{ll} \mathcal{S} & \text{of arity 2,} \\ \Lambda & \text{of arity 1,} \\ ' & \text{of arity 1} \end{array}$$

and infinitely many of combinators $n!$ in that sense that:

$$\begin{array}{rcl} \|\underline{n}\| & = & n!, \\ \|c\|env & = & c, \\ \|M\ N\| & = & \mathcal{S}(\|M\|, \|N\|), \\ \|\lambda.M\| & = & \Lambda(\|M\|). \end{array}$$

From this it is easy to establish a procedure of transition from semantic equalities to purely syntactic ones:

$$\begin{array}{rcl} 0!(x,y) & = & y, \\ (n+1)!(x,y) & = & n!x, \\ ('x)y & = & x, \\ \mathcal{S}(x,y)z & = & xz(yz), \\ \Lambda(x)yz & = & x(y,z) \end{array}$$

These rules are near to SK-rules: the first three of them indicate the "forgetting" of an argument property (similar to combinator K); the fourth rule is the uncurried version of rule S; the fifth rule is exactly the currying, i.e. the transformation of a function on two arguments into the function on the first argument, which in turn is a function on the second argument.

Introduce also a coupling combinator $< \cdot, \cdot >$, where

$$\|(M,N)\| =< \|M\|, \|N\| >,$$

and equip it by the pick outers (projections) Fst and Snd. Introduce in addition the composition operator '$\circ$' and a new command ε. Consider $\mathcal{S}\ (\cdot,\cdot)$ and $n!$ as the abbreviations for '$\varepsilon\circ < \cdot,\cdot >$' and

'$Snd \circ Fst^n$' respectively, where $Fst^{n+1} = Fst \circ Fst^n$. List now all the combinatory equalities:

$$\begin{array}{rl} (ass) & (x \circ y)z = x(yz), \\ (fst) & Fst(x,y) = x, \\ (snd) & Snd(x,y) = y, \\ (dpair) & < x, y > z = (xz, yz), \\ (ac) & \varepsilon(\Lambda(x)y, z) = x(y,z), \\ (quote) & ('x)y = x, \end{array}$$

where $(dpair)$ is a connection between pairing and coupling, and (acc) is a connection between composition and application. It can be observed, that $\mathcal{S}(x,y)z = \varepsilon(xz, yz)$. By this a consideration of operators Fst, Snd and ε becomes homogeneous. In addition, the following equality:

$$'M = \Lambda(M \circ Snd),$$

is valid, from where it follows that

$$('x)yz = xz.$$

Solution.

$DB{-}1$. Using the syntax and semantic rules, it can be derived for $M = (\lambda x.x(4, (\lambda x.x)3))+$:

$$\begin{array}{rcl} M' & = & \|M\| = \|(\lambda.\underline{0}(4, (\lambda.\underline{0})3)) + \| \\ & = & \mathcal{S}(\|\lambda.\underline{0}(4, (\lambda.\underline{0})3)\|, \| + \|) \\ & = & \mathcal{S}(\Lambda(\|\underline{0}(4, (\lambda.\underline{0})3)\|), \| + \|) \\ & = & \mathcal{S}(\Lambda(\mathcal{S}(0!, \|(4, (\lambda.\underline{0})3)\|)), \| + \|) \\ & = & \mathcal{S}(\Lambda(\mathcal{S}(0!, <' 4, \mathcal{S}(\Lambda(0!), '3) >)), \Lambda(+ \circ Snd)). \end{array}$$

$DB{-}2$. Now the evaluation by P. Landin's method for $SECD$-machine will be performed, i.e. M should be evaluated by applying M' to the environment. Currently, an environment is

empty, because the term is closed. In evaluating M' the strategy of the leftmost and at the same time innermost expression will be applied. To abbreviate denote:

$$A = \mathcal{S}(0!, <' 4, B >), B = \mathcal{S}(\Lambda(0!), 3).$$

Consider a complete sequence of reductions:

$$(\Lambda(A), \Lambda(+ \circ Snd))() \to \varepsilon(\Lambda(A)(), \Lambda(+ \circ Snd)()) \to A\, env$$

here an abbreviation $env \equiv ((), \Lambda(+ \circ Snd)())$ is used:
$\to \varepsilon(0!\, env, <' 4,\ B > env)$
$\to \varepsilon(\Lambda(+ \circ Snd)(), ('4\, env, B\, env))$
$\to \varepsilon(\Lambda(+ \circ Snd)(), (4, B\, env))$
$\to \varepsilon(\Lambda(+ \circ Snd)(), (4,\ \varepsilon(\Lambda(0!)env, '3\, env)))$
$\to \varepsilon(\Lambda(+ \circ Snd)(), (4,\ \varepsilon(\Lambda(0!)env, 3)))$
$\to \varepsilon(\Lambda(+ \circ Snd)(), (4, 0!(env, 3)))$
$\to \varepsilon(\Lambda(+ \circ Snd)(), (4, 3)) \to (+ \circ Snd)((), (4, 3))$
$\to +(Snd((), (4, 3))) \to +(4, 3) \to 7.$

Evidently, that this result is the same as the result, obtained in the immediate evaluation of the expression.

Exercises

Exercise 20.1. Construct de Bruijn's expressions for the λ-expressions below.
1) $\lambda y.yx$. *Answer.* $\|\lambda y.yx\| = \Lambda(\mathcal{S}(0!, 1!))$.
2) $(\lambda x.(\lambda z.zx)y)((\lambda t.t)z)$.

Hint. Denote the initial expression by Q and use the expression $R = \lambda zxy.Q$. Then, using a tree of representation of R, we can write it in the form: $R' = (\lambda.(\lambda.\underline{01})\underline{1})((\lambda.\underline{0})\underline{2})$, and by simple replacement of 'λ' by 'Λ', '$\circ$' by '$\mathcal{S}$', '$\underline{n}$' by '$n!$' can obtain de Bruijn's encoding for Q.

Answer. $Q_{DB(z,x,y)} = \mathcal{S}(\Lambda(\mathcal{S}(\Lambda(\mathcal{S}(0!, 1!))1!)), \mathcal{S}(\Lambda(0!), 2!))$ (the ordering of names in the subscript of Q corresponds to the order of their binding in an intermediate expression R and allows to restore the initial definition with corresponding free variables).

Chapter 21

Abstract machine: CAM

In this chapter a special version of computation theory, called *categorical abstract machine*, is constructed. To achieve the goals a special version of combinatory logic – the categorical combinatory logic is considered. It is represented by a set of combinators, every of which has its own meaning as the instruction in a programming system. This embeds in a combinatory logic one more useful application – a programming system, based on cartesian closed category. This, one more time, allows at a new level revise the connection of operator and applicative style of programming.

21.1 Theoretical background

Abbreviation 'CAM' is used for 'Categorical Abstract Machine'. As can be seen, this name has itself a voluminous sense.

21.1.1 CAM structure

At first make a substantiation for the term 'categorical'. Composition and identity are used in a *category* (identity mapping serves for a purpose of optimizing), product equipped with coupling $< \cdot, \cdot >$, and

projections Fst and Snd is added in a *cartesian category*. Currying Λ, applying ε and exponentiation, i.e., the means to construe the functional spaces, are added in a *cartesian closed category* (c.c.c.).

The rules listed in the previous chapter allow to evaluate the categorical terms of a form M', which are constructed from the combinators mentioned earlier by applying a term to the environment that is constituted from a set of various components:

1) combinators of application and coupling are not used in M';

2) application arises when $M'()$ has been written, i.e. whenever a categorical term is applied to an (empty) environment;

3) constructing of a couple is performed any time the rule $(dpair)$ is used.

Machine instructions

The machine instructions of CAM are constructed as follows:

> static operators are accepted as the basic machine instructions.

Note at first, that any redex in the rules, used for the reduction of $M'()$, is of form Mw, where w is a value, i.e. term is in a normal form relatively the rules in use. A term M is transformed by de Bruijn, i.e. it is de Bruijn's encoding. Next:

1. term M is considered as a code, acting on w, where M is constituted from the elementary components;

2. projections Fst and Snd are easily added to the set of instructions: Fst acts on the value (w_1, w_2), giving an access to the first generated element of the pair, in case it is assumed, that a value is represented by a binary tree;

3. for couples an action of $< M, N >$ on w is considered in accordance with the equality:

$$< M, N > w = (Mw, Nw).$$

The actions of M and N on w should be performed independently, and their results are joined in a tree, the root of which is a couple, and leaves are the obtained values w_1 and w_2. In a sequential machine, first, the evaluation is executed, e.g., of M. This results in w_1. Before beginning to execute w, it should be saved in a memory to have an ability to restore w in evaluating N, when w_2 is obtained. At last, w_1 and w_2 are gathered into couple, but it is assumed, that the value w_1 has been saved, and this is executed namely in the same time, when w is restored.

Machine structure

From the considerations above it follows a notion of *machine structure*:

T – term as a structured value, e.g., a graph;
C – code;
S – stack, or dump (auxiliary memory).

The *machine state* is a triple $< T,\ C,\ S >$.

It follows from the above, that *code* for $< M, N >$ is the following sequence of instructions:

> '<' , followed by a sequence of instructions, corresponding to code M, followed by ',', followed by a sequence of instructions, corresponding to code N, followed by '>'.

21.1.2 Instructions

Note, that categorical abstract machine (CAM) makes only the symbolic transformations, no compiling is performed. This is achieved by the following instructions.

Instructions $<$, $>$

Consider an action of the instructions '$<$', ',', '$>$':

$<$: pushes the term onto the top of the stack,
, : swaps the term and the top of the stack,
$>$: makes a couple from the top of the stack and the term, replaces the term by the couple just built, and pops the stack.

The particular syntax which is used for making a couple, exactly corresponds to constituting the control instructions. These instructions should replace a construction of coupling. Thus, the combining of evaluations of M and N into the evaluation $< M, N >$ is achieved.

Instructions Fst Snd $\Lambda(C)$

Respectively machine structure, the projections Fst and Snd can be described more exactly:

Fst : expects a term (s, t) and replaces it by s,
Snd : expects a term (s, t) and replaces it by t.

The code for $n!$ is constructed from n and the instruction 'Fst', followed by the instruction 'Snd'. Any additional efforts are not needed, if the following agreement is used:

take $x\,y$, or $x|y$ to denote a composition, which in usual mathematical practice is denoted by $y \circ x$. In currying, the code for $\Lambda(M)$ is $\Lambda(C)$, where C is the code for M. An action of 'Λ' is described as follows:

$\Lambda(C)$: replaces s by $C : s$, where C is the code encapsulated in Λ;

notation $C : s$ is the abbreviation of '$\Lambda(M)s$'. From the rewriting rules point of view, the action of 'Λ' is none, since $\Lambda(M)w$

is a value as soon as w is a value, hence it can not be rewritten. In terms of actions this is reformulated as:

the action of $\Lambda(M)$ on w is $\Lambda(M)w$.

The description of the command 'Λ' results in building a closure as in the $SECD$-machine. Indeed, as stressed by the notation $C : s$, the couple is handled as a value, and this couple is built from the code, corresponding to a body of λ-expression, and the value, that represents a declaration environment of the function, described by the abstraction.

Instruction $'$

The additional constants will be used. They are needed as the basic constants, when a code for $'C$ is $'(C)$ with the following action:

$'C$: replaces the term by the encapsulated constant C.

For the constructions like built-in function, e.g., for the symbol of addition, the encoding as above is used:

the code for $'(op)$ is $\Lambda((op))$, where (op) is the instruction Snd, followed by 'op'.

Instruction ε

Turn back to the application operation from the λ-calculus. It can be written as $\varepsilon\circ < M, N >$. In case a rewriting rule is needed, then the following equality is used:

$$(\varepsilon\circ < M, N >)w = \varepsilon(Mw, Nw) \qquad (= \varepsilon[Mw, Nw]).$$

The last notation includes the brackets and is used in the notations of combinatory logic.

Let (Mw, Nw) be evaluated as (w_1, w_2), that is the action of the code, associated to $< M, N >$. The instruction 'ε' is still left which gets $w_1 = \Lambda(P)w_1'$ and returns $\varepsilon(\Lambda(P)w_1', w_2) = P(w_1', w_2)$.

In terms of CAM, the code corresponding to $\varepsilon \circ < M, N >$, is the code for $< M, N >$ followed by 'ε' with the following effect:

> ε gets the term $(C : s, t)$, replaces it by (s, t), and establishes prefix C for the rest of a code.

CAM working cycle

Before building a complete list of instructions, remind the notational agreements for the instruction lists and stack elements:

an empty list L is denoted by $[\,]$; denotation $[e_1; \ldots; e_n]$ is used for the list with n elements $e_1, \ldots, e_n$;

$a.L$ writes a in the head of L;

$L1@L2$ appends $L2$ to $L1$.

The following mnemonic notations are used for convenience and listed below depending on their action:

Fst	Snd	$<$	,	$>$	ε	Λ	$'$
car	cdr	$push$	$swap$	$cons$	app	cur	$quote$

The Table 21.1 is represented to describe the working cycle of CAM. Its left side contains the initial states ("old" states), and its right side contains the resulting ones ("new" states). In fact, CAM is observed as the λ-calculus with explicit products.

In fact, this table describes a programming system with a few initial commands (instructions). Note, that among them there is no conditional evaluation (conditional branching). This instruction should be added using some agreements on the ways of the representation, how does categorical abstract machine work. Besides that in

Table 21.1: CAM working cycle

Initial configuration			Resulting configuration		
Term	Code	Stack	Term	Code	Stack
(s,t)	$car.C$	S	s	C	S
(s,t)	$cdr.C$	S	t	C	S
s	$(quoteC).C$	S	C	C	S
s	$(curC).C1$	S	$(C:s)$	$C1$	S
s	$push.C$	S	s	C	$s.S$
t	$swap.C$	$s.S$	s	C	$t.S$
t	$cons.S$	$s.S$	(s,t)	C	S
$(C:s,t)$	$app.C1$	S	(s,t)	$C@C1$	S

evaluating recursive definitions, the (implicit) fixed point operator is to be used. The additional agreements are to be introduced to achieve this goals.

21.2 Tasks

Task 21.1. For the expression:

$$let\ x\ =\ plus\ in\ x(4,(x\ where\ x=3));;$$

give its notation in terms of "categorical code" and write the program of its evaluation in terms of CAM-instructions.

Solution.

$CAM-1$. Use the mathematical notation, which stresses a relation with the rewriting rules. Let A, B be assumed as the notations of codes of A and B respectively, $+$ is an abbreviation for $plus$, $\mathcal{S}(x,y) \equiv \mathcal{S}[x,y] = \varepsilon\circ < x,y >, \oplus \equiv (Snd+) : ()$.

The resulting evaluations are listed in Table 21.2. Note, that evaluations start with an empty environment, i.e. in the position of term, is written (). The initial term is represented by de Bruijn's encoding. In the starting point of computations, the stack, or dump ("auxiliary memory") is also assumed as empty [].

Thus, CAM let us obtain the awaited result by one more way, which could be easy implemented. To achieve this goal, take in mind that the operation + is not a proper CAM-instruction, but is a built-in function of a host programming system.

Exercises

Exercise 21.1. Give a representation for the machine instructions of CAM to evaluate the expression:

$$let\ x = 3\ in\ (op(7, x)\ where\ op\ =\ sub).$$

Hint. Start with deriving a corresponding λ-expression

$$(\lambda x.(\lambda op.op\ 7\ x)sub)3,$$

then, using the postulates of the λ-calculus convert it to the form

$$(\lambda op.op(7, (\lambda x.x)3))sub$$

and obtain de Bruijn's encoding:

$$\mathcal{S}(\Lambda(\mathcal{S}(0!, <'\ 7, \mathcal{S}(\Lambda(0!), '3)\ >)), \Lambda(sub \circ Snd)).$$

Now, after rewriting this code by CAM instructions and applying the needed transformations, a final answer can be obtained.

Answer. 4.

Table 21.2: CAM computations

Term	Code	Stack
$()$	$< \Lambda(A), \Lambda(Snd+) > \varepsilon$	$[]$
$()$	$\Lambda(A), \Lambda(Snd+) > \varepsilon$	$[()]$
$A : ()$	$, \Lambda(Snd+) > \varepsilon$	$[()]$
$()$	$\Lambda(Snd+) > \varepsilon$	$[A : ()]$
$\ldots$	$\ldots$	$\ldots$
$\oplus$	$> \varepsilon$	$[A : ()]$
$(A : (), \oplus)$	ε	$[]$
$((), \oplus)$	$< Snd, <' 4, B >> \varepsilon$	$[]$
$((), \oplus)$	$Snd, <' 4, B >> \varepsilon$	$[((), \oplus)]$
$\oplus$	$, <' 4, B >> \varepsilon$	$[((), \oplus)]$
$((), \oplus)$	$<' 4, B >> \varepsilon$	$[\oplus]$
$((), \oplus)$	$'4, B >> \varepsilon$	$[((), \oplus); \oplus]$
4	$, B >> \varepsilon$	$[((), \oplus); \oplus]$
$((), \oplus)$	$>> \varepsilon$	$[4; \oplus]$
$((), \oplus)$	$(Snd),' 3 > \varepsilon >> \varepsilon$	$[((), \oplus); 4; \oplus]$
$Snd : ((), \oplus)$	$, 3 > \varepsilon >> \varepsilon$	$[((), \oplus); 4; \oplus]$
$((), \oplus)$	$3 > \varepsilon >> \varepsilon$	$[Snd : ((), \oplus); 4; \oplus]$
3	$> \varepsilon >> \varepsilon$	$[Snd : ((), \oplus); 4; \oplus]$
$(Snd : ((), \oplus), 3)$	$\varepsilon >> \varepsilon$	$[4; \oplus]$
$(((), \oplus), 3)$	$Snd >> \varepsilon$	$[4; \oplus]$
3	$>> \varepsilon$	$[4; \oplus]$
$(4, 3)$	$> \varepsilon$	$[\oplus]$
$(\oplus, (4, 3))$	ε	$[]$
$((), (4, 3))$	$Snd+$	$[]$
$(4, 3)$	$+$	$[]$
7	$[]$	$[]$

Chapter 22

Optimizing CAM-computations

A cartesian closed category gives the additional abilities to optimize the resulting program code. Besides the properties of combinatory logic used as a shell, the special categorical equalities, taken from a cartesian closed category as from an application, are applicable.

22.1 Task

Task 22.1. For the expression:

$$let\ x = 5\ in\ let\ z\ y = y + x\ in\ let\ x = 1\ in\ (zx) \times 2;;$$

compile CAM-program and, possibly, optimize it.

Solution.

opt−1. Write the initial expression:

$$P\ =\ let\ x = 5\ in\ let\ z\ y = y + x\ in\ let\ x = 1\ in\ (zx) \times 2.$$

Table 22.1: Evaluation of a substitution

Term	Code	Stack
s	$push.(cur\ C).swap.C1@[cons;\varepsilon]$	S
s	$(cur\ C).swap.C1@[cons;\varepsilon]$	$s.S$
$C:s$	$swap.C1@[cons;\varepsilon]$	$s.S$
s	$C1@[cons;\varepsilon]$	$(C:s).S$
w	$[cons;\varepsilon]$	$(C:s).S$
$(C:s,w)$	$[\varepsilon]$	S
(s,w)	C	S

Its reduction to the λexpression results in:

$$P \;=\; (\lambda x.(\lambda z.(\lambda x.(zx) \times 2)1)(\lambda y.y + x))5.$$

Note, that this notation leads to a simple optimizing in a compiling time. This is possible, because the code, corresponding to the expression $(\lambda.M)N$, is the instruction '$push$', followed by'$cur\ C$' (where C is the code of M), followed by '$swap$', followed by the code $C1$ for N, followed by '$cons$' and 'ε'. Assuming, that an evaluation of $C1$ on the term s results in the value w, write the main steps of a computation:

$$\text{code of } ((\lambda.M)N) \;=\; push.cur\ C.swap.C1@[cons;\varepsilon],$$
$$C \;=\; \text{code of } (M), \quad C1 \;=\; \text{code of } (N).$$

Table 22.1 represents the evaluation.

Note, that

$$\|(\lambda.M)N\| \;=< \Lambda(\|M\|), \|N\| > \varepsilon.$$

Consider the means of deriving the optimized code. Besides the possible evaluation of expressions relatively the environment, as was already considered, it is possible, using the categorical combinatory logic, rather naturally simulate the β-reduction. To achieve this goal, a set of rules is used, which are distinct from the above, and only the pure "categorical" combinators are used, i.e. both coupling and applying are eliminated. A starting point is the rule:

$$(Beta) \qquad \varepsilon\circ < \Lambda(x), y >= x\circ < Id, y > .$$

A validity of this rule is proved as follows:

$$\begin{aligned}(\varepsilon\circ < \Lambda(x), y >)t \;&=\; \varepsilon(< \Lambda(x), y > t)\\ &= \varepsilon(\Lambda(x)t, yt)\\ &= x(t, yt) = x(Id\, t, yt)\\ &= (x\circ < Id, y >)t,\end{aligned}$$

from where the principle of comprehension $(Beta)$ can be derived. Note, that $Id\, x \;=\; x$. By the rule $(Beta)$ the expression

$$let\; x \;=\; N\; in\; M$$

is associated with the code

$$push.skip.swap@C1@cons.C,$$

where *skip* replaces Id (with no action). The action of the construction $[push.skip.swap]$ is the same as of $[push]$, i.e. the considered optimization has a sound ground.

opt−2. Show, that the optimization of $[push.skip.swap]$ by replacing $[push]$ is valid. Indeed, in terms of λ-calculus we obtain:

$$< f, g >= \lambda t.\lambda r.r(ft)(gt) \;=\; \lambda t.(ft, gt),$$

and also obtain:

$$< g >= \lambda t.\lambda r.r(t)(gt) = \lambda t.(t, gt) = \lambda t.(Id\, t, gt).$$

Replacing f in the first equality by Id, results in:

$$< Id, g >=< g >,$$

that substantiates the introducing of the notation '$< \cdot >$' for a single expression (a special case). Next,

$$\begin{aligned} < Id, g > &= [push; skip; swap; g; cons], \\ < g > &= [push; g; cons]. \end{aligned}$$

From that the needed optimization is derived as an equality.

opt−3. Let an additional rule be introduced as an optimization rule for the code $[\oplus]$[1]. The proof for this optimization is not difficult. Indeed, the following equalities are valid:

$$[\oplus] \qquad \begin{aligned} \varepsilon \circ < \Lambda(+ \circ Snd), < M, N >> &= \\ &= (+ \circ Snd) \circ < Id, < M, N >> \\ &= + \circ < M, N >=< M, N > +. \end{aligned}$$

In more details:

$$\begin{aligned} (\varepsilon \circ < \Lambda(+ \circ Snd), < M, N >)t &= \\ &= \varepsilon(\Lambda(+ \circ Snd)t, < M, N > t) \\ &= (\Lambda(+ \circ Snd)t)(< M, N > t) \\ &= (+ \circ Snd)(t, < M, N > t) \\ &= (+(Snd(t, < M, N > t) \\ &= (+ \circ < M, N >)t, \end{aligned}$$

from which the needed equality is derived. In addition, a sequence $[cons; plus]$ can be replaced by $[plus]$, if '$plus$' is considered as a two placed function that takes as its arguments the

[1] It turns out, that e.g., the optimization can be done by compiling $M + N$ into the code $\langle M, N \rangle$, followed by '$plus$'.

term and the top of a stack. It should be noted, that an ability of the operation '*plus*' to be applied to its arguments is partially lost (because the λ-term $(\lambda x.\lambda y. + xy)$ is replaced by the two placed operator $x + y$, which needs both of the operands at once), but this lost of generality for operating is overcame by the ability to optimize the particular cases of three- and, generally, of n-placed functions. This can be grounded by the following rules (the case of three placed functions):

$$\begin{aligned}\lambda t.(ft, gt, kt) &= \lambda t\lambda r'.r'(\lambda r.r(ft)(gt))(kt) = \\ &= \lambda t.(< f, g > t, kt) = << f, g >, k >,\end{aligned}$$

therefore:

$$+(M, N, O) = + << M, N >, O > .$$

Indeed,

$$\begin{aligned}&\varepsilon\circ < \Lambda(+ \circ Snd), \varepsilon\circ < \Lambda(Snd), << M, N >, O >>> = \\ &= + \circ Snd\circ < Id, \varepsilon\circ < \Lambda(Snd), << M, N >, O >>> \\ &= + \circ (\varepsilon\circ < \Lambda(Snd), << M, N >, O >>>) \\ &= + \circ Snd\circ < Id, << M, N >, O >>) \\ &= +\circ << M, N >, O > .\end{aligned}$$

Thus, for initial task the following main steps of computation are distinguished:

s	$push.C1@cons.C$	S
s	$C1@cons.C$	$s.S$
w	$cons.C$	$s.S$
(s, w)	C	S

This optimized computation is based on the identity combinator, without which the combinatory logic in use is hardly be called the categorical one.

opt−4. Turn back to the CAM computations with the term P. Use the notation $x \mid y \equiv y \circ x$ to save writing of the compiled expression. In the following the abbreviations:

$$Fst = F,\ Snd = S,$$

will be used. To make all the writings closed, consider the derivations in details. A formulation of optimizing principle $(Beta)$ will be taken in the form:

$$(Beta) \qquad < \Lambda(X), Y > | \ \varepsilon = x \circ < Id, y > = < y > | \ x.$$

A formulation of optimizing principle $[\oplus]$ will be used in the form:

$$[\oplus] \qquad < \Lambda(+ \circ Snd), < M, N >> | \varepsilon = \\ = (+ \circ Snd) \circ < Id, < M, N >>.$$

De Bruijn's encoding gives:

$$P = (\lambda x.(\lambda z.(\lambda x.(z\ x) \times 2)1)(\lambda y.y + x))5,$$

next

$$\begin{aligned} P' = \ & (\lambda.(\lambda.(\underline{1}\ \underline{0}) \times 2)1)(\lambda.\underline{0} + \underline{1}))5 \\ & = (\lambda.(\lambda.(\lambda.\times((\underline{1}\ \underline{0}), 2))1)(\lambda.+(\underline{0}, \underline{1})))5. \end{aligned}$$

Next the evaluation of P' results in:

$$\|P'\| \ = <' \ 5 > | \ \|(\lambda.(\lambda(\underline{1}\ \underline{0}) \times 2)1)(\lambda.\underline{0} + \underline{1})\|;$$

$$\begin{aligned} \|(\lambda.(\lambda(\underline{1}\ \underline{0}) \times 2)1)(\lambda.\underline{0} + \underline{1})\| = & \\ = \quad & < \|\lambda.(\lambda.(\underline{1}\ \underline{0}) \times 2)1\|, \|\lambda.\underline{0} + \underline{1}\| > \varepsilon \\ = \quad & < \Lambda\|\lambda.(\underline{1}\ \underline{0}) \times 2)1\|, \|\lambda.\underline{0} + \underline{1}\| > \varepsilon \\ \overset{(Beta)}{=} \quad & < \|\lambda.\underline{0} + \underline{1}\| > | \ \ \|(\lambda.(\underline{1}\ \underline{0}) \times 2)1\|; \end{aligned}$$

$$
\begin{array}{rcl}
\|(\lambda.(\underline{1}\ \underline{0}) \times 2)1\| & = & <\|\lambda.(\underline{1}\ \underline{0}) \times 2)\|,'1> \varepsilon \\
& = & <\Lambda\|(\underline{1}\ \underline{0}) \times 2\|,'1> \varepsilon \\
& = & <'\ 1>|\ \|(\underline{10}) \times 2\|; \\
< \|\lambda.\underline{0} + \underline{1}\| > & = & <\Lambda\| + (\underline{0}, \underline{1})\| > \\
& = & <\Lambda <' +, < 0!, 1! >> \varepsilon > \\
& = & <\Lambda < \Lambda(+ \circ S), < 0!, 1! >> \varepsilon >; \\
& \overset{(\oplus)}{=} & <\Lambda(+\circ < 0!, 1! >) > \\
& = & <\Lambda(< 0!, 1! >| +) > \\
& = & <\Lambda(< S, F \mid S >| +) >; \\
\|(\underline{1}\ \underline{0}) \times 2)\| & = & <\| \times \|, < \|(\underline{1}\ \underline{0})\|,'2 >> \varepsilon \\
& = & <\Lambda(\times \circ S), < \|(\underline{1}\ \underline{0})\|,'2 >> \varepsilon \\
& \overset{(\oplus)}{=} & \times\circ < \|(\underline{1}\ \underline{0})\|,'2 > \\
& = & \times\circ << 1!, 0! > \varepsilon,'2 > \\
& = & \times\circ << F \mid S, S > \varepsilon,'2 > \\
& = & << F \mid S, S >| \varepsilon,'2 >| \times.
\end{array}
$$

To save writing use the abbreviations. Using the abbreviation for D, where $D = (< S, F \mid S >| +)$ and for C, where $C = < F \mid S, S >| \varepsilon$, obtain:

$$\|P'\| = <'\ 5 >|< \Lambda(D) >|<'\ 1 >|< C', 2 >|\ \times.$$

To save more writing with CAM-computations use the abbreviations:

$$B =< C,'2 >| \times,\ \ C =< F \mid S, S >| \varepsilon,\ \ D =< S, F \mid S >| +.$$

We pay attention, that these re-writings, evidently, have the connections with supercombinators. Indeed, every object that is introduced as a mathematical notational agreement, is the sequence of CAM-instructions. This sequence can be evaluated beforehand (compiled) with CAM, and, by need, this result of preliminary evaluation could be used in a proper place. It is not difficult to see, that these preliminary

computations should be better done not episodically, but using the "discipline" of computations[2].

The categorical abstract machine itself is relatively well balanced system of "mathematical" computations. These computations with a good evidence can be arranged in a table with three columns, which represents the current values of term, code and stack. Remind, that namely the triple <term, code, stack > is exactly a current state (of computation). Thus, a sequence of rows in the table of CAM-computations determines the sequence of states of the computation process[3] .

[2]The methods of optimizing the computations, when it is possible to select relatively independent parameters, are developed not only in different versions of "supercombinatory" programming, but are widely used in a functional programming. The approaches are distinct in semantics of evaluations: as a rule, a denotational semantic with "continuations" is used. It means, that for any well in some sense defined computation the "rest of program" is known.

[3]From the mathematical point of view a computation can be considered as a sequence of states, i.e. as the *process* in a rigorous sense of this term. This point of view is quite in a spirit of the theory of computations by D.S. Scott.

Consider the complete sequence of CAM-computations:

$()$	$<' 5 >< \Lambda(D) ><' 1 > B$	$[]$
$()$	$'5 >< \Lambda(D) ><' 1 > B$	$[()]$
(5)	$>< \Lambda(D) ><' 1 > B$	$[()]$
$((), 5)$	$< \Lambda(D) ><' 1 > B$	$[]$
$((), 5)$	$\Lambda(D) ><' 1 > B$	$[((), 5)]$
$(D : ((), 5))$	$><' 1 > B$	$[((), 5)]$
$(((), 5), D : ((), 5))$	$<' 1 > B$	$[]$
$(((), 5), D : ((), 5))$	$'1 > B$	$[\ldots]$
(1)	$> B$	$[\ldots]$
$(((), 5), D : ((), 5)); 1$	B	$[]$
$(((), 5), D : ((), 5)); 1$	$< C,' 2 > \times$	$[]$
$(((), 5), D : ((), 5)); 1$	$C,' 2 > \times$	$[\ldots; 1]$
$(((), 5), D : ((), 5)); 1$	$< F \mid S, S >\mid \varepsilon,' 2 > \times$	$[\ldots; 1]$
$(((), 5), D : ((), 5)); 1$	$F \mid S, S >\mid \varepsilon,' 2 > \times$	$[\ldots; \ldots; 1]$
$((D : ((), 5)))$	$, S >\mid \varepsilon,' 2 > \times$	$[\ldots; \ldots; 1]$
$(((), 5), D : ((), 5)); 1$	$S >\mid \varepsilon,' 2 > \times$	$[D : ((), 5)); \ldots]$
(1)	$> \varepsilon,' 2 > \times$	$[D : ((), 5)); \ldots]$
$((D : ((), 5)); 1)$	$\varepsilon,' 2 > \times$	$[\ldots]$
$(((), 5); 1)$	$D,' 2 > \times$	$[\ldots]$
$(((), 5); 1)$	$< S, F \mid S > +,' 2 > \times$	$[\ldots]$
$(((), 5); 1)$	$S, F \mid S > +,' 2 > \times$	$[(((), 5); 1); \ldots]$
(1)	$, F \mid S > +,' 2 > \times$	$[(((), 5); 1); \ldots]$
$(((), 5); 1)$	$F \mid S > +,' 2 > \times$	$[1; \ldots]$
(5)	$> +,' 2 > \times$	$[1; \ldots]$
$(1, 5)$	$+,' 2 > \times$	$[\ldots]$
(6)	$,' 2 > \times$	$[\ldots]$
$(\ldots)$	$'2 > \times$	$[6]$
(2)	$> \times$	$[6]$
$(6, 2)$	$\times$	$[]$
(12)	$[]$	$[]$

Exercises

Exercise 22.1. Write the optimized program for CAM-computations of evaluating the expression:

$$let\ x = 3\ in\ let\ z = x + y\ in\ fz\ where\ y = 1\ where\ f = sqr.$$

Hint. Introduce the λ-expression:

$$(\lambda x.(\lambda z.(\lambda f.fz)sqr)((\lambda y. + xy)1))3,$$

and obtain its de Bruijn's encoding:

$$(\lambda.(\lambda.(\lambda.\underline{0}\ \underline{1})sqr)((\lambda. +\ \underline{1}\ \underline{0})\ 1))3.$$

Next,

$$\begin{aligned}
\mathcal{S}\|\lambda.(\ldots)(\cdots), 3\| &= \mathcal{S}(\Lambda(\|(\ldots)(\cdots)\|),' 3);\\
\|(\ldots)(\cdots)\| &= \mathcal{S}(\|(\ldots)\|, \|(\cdots)\|);\\
\|(\ldots)\| &= \Lambda(\mathcal{S}(\Lambda(\mathcal{S}(0!, 1!)), \|sqr\|));\\
\|(\cdots)\| &= \mathcal{S}(\Lambda(\mathcal{S}(\mathcal{S}(\| + \|, 1!), 0!)),' 1);
\end{aligned}$$

In the expressions the subscripts will indicate a balance of paired parentheses, e.g.,

$$_0(\ldots)_0, \ldots, _6(\ldots)_6.$$

Write the following:

$$\begin{aligned}
&\mathcal{S}_0(\Lambda_1(\mathcal{S}_2(\Lambda_3(\mathcal{S}_4(\Lambda_5(\mathcal{S}_6(0!, 1!)_6)_5, \|sqr\|)_4)_3,\\
&\qquad \mathcal{S}_3(\Lambda_4(\mathcal{S}_5(\mathcal{S}_6(\| + \|, 1!)_6, 0!)_5)_4,' 1)_3)_2)_1,' 3)_0 \equiv R.
\end{aligned}$$

Checking is done by the straightforward computations:

$$\begin{aligned}
R\rho &\equiv \mathcal{S}_0(\ldots,' 3)_0\rho\\
&= (\Lambda_1(\ldots)_1\rho)('3\ \rho)\\
&= {}_1(\ldots)_1(\rho, 3)\\
&\equiv {}_1(\ldots)_1\rho';
\end{aligned}$$

$$
\begin{aligned}
{}_1(\ldots)_1\rho' &\equiv \mathcal{S}_2(\ldots\ ,\ \ldots)_2\rho' \\
&= \mathcal{S}_4(\ldots\ ,\ \ldots)_4(\rho', {}_4(\ldots)_4(\rho', 1));
\end{aligned}
$$

$$
\begin{aligned}
{}_4(\ldots)_4(\rho', 1) &= \mathcal{S}_5(\ldots\ ,\ \ldots)_5(\rho', 1) \\
&= \mathcal{S}_6(\ldots\ ,\ \ldots)_6(\rho', 1)1 \\
&= \Lambda(+ \circ Snd)(\rho', 1)\rho'\ 1 \\
&= (+ \circ Snd)((\rho', 1), 3)1 \\
&= +\ 3\ 1 = 4;
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_4(\ldots\ ,\ \ldots)_4(\rho', 4) &= \Lambda_5(\ldots)_5(\rho', 4)(\Lambda_5(\cdots)_5(\rho', 4)) \\
&= {}_5(\ldots)_5((\rho', 4), (\Lambda_5(\cdots)_5(\rho', 4))) \\
&\equiv \mathcal{S}(0!, 1!)(\ldots\ ,\ \ldots) \\
&= \Lambda_5(\cdots)_5(\rho', 4)4 \\
&= {}_5(\cdots)_5((\rho', 4), 4) \\
&\equiv (sqr \circ Snd)((\rho', 4), 4) \\
&= sqr(4) = 16.
\end{aligned}
$$

The derived expression R should be optimized by the rules $(Beta)$ and $[\oplus]$. To achieve this goal the following equality:

$$\mathcal{S}(x, y) = \varepsilon \circ < x, y > .$$

can be useful.

Answer. $sqr(4) = 16$.

Test

1. List the alternative ways to eliminate a collision of variables in the λ-expressions.

2. Give the reasons to introducing the composition operator.

3. Show connection and difference between the notions of ‘pair’ and ‘couple’.

Hint. It is possible to use the set theoretic definitions of these notions, assuming

$$f : D \to E, \quad g : D \to F.$$

From this it follows, that

$$\begin{aligned} \text{couple} &: \quad h =< f, g >: D \to E \times F; \\ \text{pair} &: \quad [f, g] = (f, g) : (D \to E) \times (D \to F). \end{aligned}$$

4. List three main approaches to implementation of the functional programming languages, which the notion of the categorical abstract machine is based on.

5. What are the basic machine instructions of CAM?

6. Describe the projections Fst and Snd in connection with a structure of machine.

7. Give the formulations of the main rules of optimization.

8. What are the advantages of using the principles $(Beta)$ and $[\oplus]$ in writing a CAM-program?

Chapter 23

Variable objects

In this chapter the general topics of mathematical representation of objects are considered. As it shown, the notion of functor-as-object allows in a laconic form to revise the main laws of object-oriented computations. In particular, an attention is paid to the changeable (variable) notions-concepts, that are the usual objects of combinatory logic but have the useful in programming properties. For instance, the variable concepts, without complications, allow to build not only the theory of computations, but a semantic of programming languages, and the data object models as well. The notion of data-as-object brings in the new extent of the computing derivations.

23.1 Models

In such applicative computational system as *untyped* λ-calculus one and the same object depending on the context can play both the role of an argument and the role of a function which acts on the argument(s). If the arguments are taken from the domain D, then the functions are to be taken from the domain $D \to D$, and, as they can change their roles, we need the isomorphism between the domain of arguments D

and the domain of mappings $D \to D$, i.e.:

$$D \simeq (D \to D),$$

but, in general, this is impossible. However, in a particular case when the domain $D \to D$ is restricted to the set of functions which are continuous in some topology on D, it is possible. As was shown by D. Scott ([112]), to build the semantic of λ-calculus which covers available in computer science and, in particular, in programming computational mechanisms, it is sufficient to take a category of *complete lattices* with continuous mappings. In this case it is possible to build the object D_∞, which is isomorphic to the object $D_\infty \to D_\infty$, i.e.:

$$D_\infty \simeq (D_\infty \to D_\infty),$$

that leads to development of the extensional model of λ-calculus.

23.1.1 Applicative structure

The applicative structure, being simple and conceptually transparent, gives a sound ground to formulate, develop and research the variety of data models.

Assume, that all the elements can be gathered into one set, the domain D, on which the only operation of *application* $(\cdot\,\cdot)$ is defined:

$$(\cdot\,\cdot) : D \times D \to D \; : \; \forall x, y \in D \, \exists z \in D . [x, y] \mapsto z \;\&\; z = (x\, y),$$

or, in other notations,

$$\frac{x \in D \qquad\qquad y \in D}{(x\, y) \in D}, \qquad (\cdot\,\cdot)$$

i.e. for any pair of elements x, y from D it can be built the $(x\, y)$, called the 'application of x to y'. This pair D and $(\cdot\,\cdot)$:

$$\mathcal{M} = (D, (\cdot\,\cdot))$$

is called an *applicative structure*.

Note that in the literature as a notation for explicit application instead of '$(\cdot\,\cdot)$' is often used 'ε':

$$\varepsilon \;:\; D \times D \to D \;:\; \forall x, y \in D\, \exists z \in D.[x, y] \mapsto z \;\&\; z = \varepsilon[x\, y] \equiv (x\, y),$$

This notation can be equally simplified to:

$$\varepsilon : D \times D \to D,\ \varepsilon : [x, y] \mapsto (x\, y);$$

then the applicative structure will be of form:

$$\mathcal{M} = (D,\ \varepsilon).$$

The applicative structure $\mathcal{M}$ is *extensional*, if the following condition is valid:

$$\frac{\forall a, b \in D\, \forall x \in D.ax = bx}{a = b}.$$

Note that for the structure $\mathcal{M} = (D,\ \varepsilon)$ instead of '$d \in D$' it is often written '$d \in \mathcal{M}$'.

Terms

Consider a class of all the *terms* in $\mathcal{M}$, denoted by $\mathcal{T}(\mathcal{M})$. This class is built by induction on complexity of the term is:

i) class of the terms contains all the *variables* $v_1, \ldots, v_i, \ldots$:

$$v_1, \ldots, v_i, \ldots \in \mathcal{T}(\mathcal{M});$$

ii) class of the terms contains all the *constants* c_a:

$$\frac{a \in \mathcal{M}}{c_a \in \mathcal{T}(\mathcal{M})};$$

iii) applicative structure is *closed* by the application operation, i.e.

$$\frac{M, N \in \mathcal{T}(\mathcal{M})}{(M\, N) \in \mathcal{T}(\mathcal{M})}.$$

Here: M, N are arbitrary terms from $\mathcal{T}(\mathcal{M})$.

Assignment

An *assignment* in the applicative structure $\mathcal{M}$ is the mapping ρ:

$$\rho : \text{variables} \to \mathcal{M},$$

i.e. $\rho(v_i) \in \mathcal{M}$ for any variable v_i.

Evaluation mapping

Begin with the fixing of applicative structure $\mathcal{M}$ and assignment ρ. The *evaluation*, or interpretation of a term M is the mapping:

$$\|\cdot\|\cdot : \text{terms} \times \text{assignments} \to \text{elements from } \mathcal{M}.$$

An evaluation of the terms from $\mathcal{T}(\mathcal{M})$ in $\mathcal{M}$ with the assignment ρ is built by induction on complexity of a term:

i) all the *variables* $x \in \mathcal{T}(\mathcal{M})$ are interpreted as:

$$\|x\|\rho = \rho(x),$$

where $\rho(x) \in \mathcal{M}$;

ii) all the *constants* $c_a \in \mathcal{T}(\mathcal{M})$ are interpreted as:

$$\|c_a\|\rho = a,$$

where $a \in \mathcal{M}$;

iii) the application of a term M to a term N is interpreted as:

$$\|(M\,N)\|\rho = (\|M\|\rho)(\|N\|\rho),$$

i.e., by a principle: ‘an evaluation of the application is an application of the evaluations’.

The fact, that equality $M = N$ of the terms $M, N \in \mathcal{T}(\mathcal{M})$ is *true in* $\mathcal{M}$ *with the assignment* ρ, is denoted by:

$$\mathcal{M}, \rho \models M = N$$

and has a meaning of the equality $\|M\|\rho = \|N\|\rho$.

As can be seen from a definition of the evaluation $\|M\|\rho$, it depends on the assignment of the values ρ on the set of free variables of M. Thus, in case of *closed* terms M, when $FV(M) = \varnothing$, the evaluation $\|M\|\rho$ does not depend on an assignment ρ, that is denoted by $\|M\|$.

Substitution

One of the most often used constructions in programming languages is a substitution. Its sense reveals in performing the assignments. Let the assignment ρ be fixed and take an element $a \in \mathcal{M}$. Then the construction 'assignment of a substitution a in place of the variable x' is denoted by '$\rho([a/x])$' and defined as:

$$\rho([a/x]) \stackrel{\text{def}}{=} \rho'(y)|_a^x \equiv \begin{cases} a, & \text{if } y \equiv x, \\ \rho(y), & \text{if } y \not\equiv x, \end{cases}$$

where y is a variable.

23.1.2 Typed models

Note, that initially the λ-calculus both in a type free and in a typed variants have been considered as the means or metatheory to formalize the notion of a *rule* or *process*. To use the types, the syntax of a typed language is to be introduced.

Types

Type symbols are defined by induction on complexity:

i) basic type o ia a type symbol;

ii) if σ and τ are the types, then (σ, τ) is a type:

$$\frac{\sigma - \text{type}, \qquad \tau - \text{type}}{(\sigma, \tau) - \text{type}}.$$

Variables

Every *variable* x_i is assigned a type σ, and this fact is written by one of the ways:

$$x_i^\sigma, \quad \text{or } x_i : \sigma, \quad \text{or } \#(x_i) = \sigma.$$

The objects s, corresponding to the λ-*terms*, which contain a set of *free variables* $FV(s)$ and a set of *bound variables* $BV(s)$, are assigned the types by induction on complexity:

i) a variable x_i^σ is the term of type σ, where $FV(x_i^\sigma) = \{x_i^\sigma\}$, $BV(x_i^\sigma) = \varnothing$;

ii) if s is a term of type (σ, τ), and t is a term of type σ, then $(s\,t)$ is the term of type τ:

$$\frac{s : (\sigma, \tau) \qquad t : \sigma}{(s\,t) : \tau}, \qquad (\cdot\,\cdot)$$

a set of free variables in the compound term is the union of the sets of free variables in the component terms:

$$FV((s,t)) = FV(s) \cup FV(t),$$

and a set of bound variables in the compound term is the union of the sets of bound variables in the component terms:

$$BV((s,t)) = BV(s) \cup BV(t);$$

iii) if s is a term of type τ, and y is a variable of type σ, then $\lambda y.s$ is the term of type (σ, τ):

$$\frac{y : \sigma \qquad s : \tau}{(\lambda y.s) : (\sigma, \tau)}, \qquad (\lambda \cdot . \cdot)$$

the variable y is excluded from a set of free in s variables:

$$FV((\lambda y.s)) = FV(s) - \{y\},$$

and the variable y is added to a set of bound in s variables:

$$BV((\lambda y.s)) = BV(s) \cup \{y\}.$$

Thus, metaoperator of *abstraction* $(\lambda \cdot . \cdot)$ binds one variable in the one term. The set of all the λ-terms is denoted by Tm.

Pre-structure

A *pre-structure* is the family

$$(\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\})$$

with parameters σ, τ, where D^σ is a non-empty set, corresponding to arbitrary type symbol σ, and $\varepsilon_{\sigma\tau}$ is the mapping

$$\varepsilon_{\sigma\tau} : D^{(\sigma,\tau)} \times D^\sigma \to D^\tau, \quad \varepsilon_{\sigma\tau} : [x, z] \mapsto x(z)$$

for a function $x \in D^{(\sigma\tau)}$ and argument $z \in D^\sigma$, which is valid for all the type symbols σ, τ.

The pre-structure is *extensional*, if for the elements $x, y \in D^{(\sigma,\tau)}$ on arbitrary element $z \in D^\sigma$ from the equality $\varepsilon_{\sigma\tau}[x, z] = \varepsilon_{\sigma\tau}[y, z]$ it follows, that $x = y$:

$$\frac{x, y \in D^{(\sigma,\tau)} \qquad \forall z \in D^\sigma . \varepsilon_{\sigma\tau}[x, z] = \varepsilon_{\sigma\tau}[y, z]}{x = y}.$$

Assignment

An *assignment* in the system $(\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\})$ is the function ρ with a domain constituted of the set of all the variables and such, that

$$\rho(x_i^\sigma) \in D^\sigma.$$

A set of all the assignments is denoted by Asg.

If x, y are the variables, then $\rho(\cdot)|_a^x$ is determined by:

$$\rho(y)|_a^x \equiv \begin{cases} \rho(x)|_a^x = a, & \text{if } y \equiv x, \\ \rho(y), & \text{if } y \not\equiv x. \end{cases}$$

Compare with the definition of assignment of a substitution in 23.1.1 on p. 259.

Structure

A *structure* is the family

$$(\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\}, \|\cdot\|\cdot),$$

with parameters σ, τ, where $(\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\})$ is a pre-structure, and $\|\cdot\|\cdot$ is the evaluation mapping

$$\|\cdot\|\cdot : Tm \times Asg \to \bigcup_\sigma D^\sigma,$$

which is defined elementwise by induction on term construction:

i) $\|x_i^\sigma\|\rho = \rho(x_i^\sigma)$;

ii) $\|(s\,t)\|\rho = \varepsilon_{\sigma\tau}[\|s\|\rho, \|t\|\rho]$, where term s is assigned the type (σ, τ), and t is assigned the type σ;

iii) for any element $a \in D^\sigma$ the following equality is valid:

$$\varepsilon_{\sigma\tau}[\|(\lambda x.s)\|\rho, a] = \|s\|\rho|_a^x,$$

where term s is assigned the type τ, and the variable x is assigned the type σ.

For arbitrary structure $(\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\}, \|\cdot\|\cdot)$ write:

$$(\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\}) \models s = t[\rho]$$

in that case, if

$$\|s\|\rho = \|t\|\rho.$$

Example 23.1. Consider a special case of the structure, denoted by T_B:

$$T_B = (\{B^\sigma\}, \{\varepsilon_{\sigma\tau}\}),$$

that is defined on B. For this structure:

$$B^o = B, \quad B^{\sigma\to\tau} = B^\sigma \to B^\tau, \quad \varepsilon_{\sigma\tau}[x, y] = x(y).$$

This is a sample of the structure of *complete type* on B. Indeed, for the type $(\sigma, \tau) \equiv (\sigma \to \tau)$ corresponding domain (a set) of possible elements of this type is exactly $B^{\sigma\to\tau}$. Its every element is the mapping of type $\sigma \to \tau$. On the other hand, $B^\sigma \to B^\tau$ has as a domain the set of all the mappings from B^σ to B^τ. In the structure of complete type it is regarded that these sets are to be likewise.

Substitution

If s is a term, x is a variable, and t is the term of the same type as x, the notation

$$\text{`}[t/x]s\text{'}$$

is read as

> 'substitution of term t in place of every free occurrence of x in s'.

A substitution is defined by induction on complexity of s as follows: for *atomic* s:

i) $[t/x]x = t$;

ii) $[t/x]y = y$ for the variables $y \not\equiv x$;

for *non-atomic* s:

iii) $[t/x](r\ s) = ([t/x]r)([t/x](r\ s))$;

iv) $[t/x](\lambda x.s) = (\lambda x.s)$;

v) $[t/x](\lambda y.s) = (\lambda y.[t/x]s)$ for the variables $y \not\equiv x$.

Substitution is the function $Subst$ from all the variables to terms such, that $Subst(x)$ has the same type as x. Similarly, take $s(Subst)$ as a notation for the simultaneous substitution of every free occurrence of any variable y in s, i.e. $Subst(y)$.

Consistency

Given a structure $M = (\{D^\sigma\}, \{\varepsilon_{\sigma\tau}\}, \|\cdot\|\cdot)$ prove a theorem of consistency.

Lemma 23.1. In case the bound variables of the term s and free variables of the term t are distinct, i.e.

$$BV(s) \cap FV(t) = \varnothing,$$

then the following equality is used to substitute the term t in place of the variable x in the term s:

$$\|[t/x]s\|\rho = \|s\|\rho|^x_{\|t\|\rho}.$$

Proof. For the fixed t by induction on complexity of s. □

Lemma 23.2. If the variable y is distinct both from free and bound in s variables:

$$y \notin FV(s) \cup BV(s),$$

then the following equality takes place:

$$\|[y/x]s\|\rho|^y_d = \|s\|\rho|^x_d.$$

Proof. It follows from Lemma 23.1. □

Lemma 23.3. If the variable y is distinct both from free and bound in s variables:

$$y \notin FV(s) \cup BV(s),$$

then the equality holds:

$$M \models (\lambda x.s) = (\lambda y.[y/x]s).$$

Proof. From Lemma 23.2 it follows that

$$\begin{aligned} \varepsilon[\|(\lambda x.s)\|\rho, d] &= \|s\|\rho|^x_d \\ &= \|[y/x]s\|\rho|^y_d \\ &= \varepsilon[\|(\lambda y.[y/x]s)\|\rho, d]. \end{aligned}$$

□

Lemma 23.4. In case the bound variables of the term s and free variables of term t are distinct, i.e.

$$BV(s) \cap FV(t) = \varnothing,$$

then the following equality holds to substitute the term t in place of the variable x in the term s:

$$\|(\lambda x.s)t\|\rho = \|[t/x]s\|\rho.$$

Proof. From Lemma 23.1 it follows that

$$\begin{aligned} \|(\lambda x.s)t\|\rho &= \varepsilon[\|(\lambda x.s)\|\rho, \|t\|\rho] \\ &= \|s\|\rho|^x_{\|t\|\rho} \\ &= \|[t/x]s\|\rho. \end{aligned}$$

□

Lemma 23.5. If the variable x is distinct from any of the free in term s variables, i.e.

$$x \notin FV(s),$$

then the equality holds:

$$\|(\lambda x.sx)\|\rho = \|s\|\rho.$$

Proof.

$$\begin{aligned} \varepsilon[\|(\lambda x.sx)\|\rho, d] &= \|(sx)\|\rho|^x_d \\ &= \varepsilon[\|s\|\rho|^x_d, \|x\|\rho|^x_d] \\ &= \varepsilon[\|s\|\rho, d]. \end{aligned}$$

□

Lemma 23.6. If $M \models s = t$, then $M \models (\lambda x.s) = (\lambda x.t)$.

Proof. Fix ρ. Then:

$$\begin{aligned} \varepsilon[\|(\lambda x.s)\|\rho, d] &= \|s\|\rho|^x_d \\ &= \|t\|\rho|^x_d \\ &= \varepsilon[\|(\lambda x.t)\|\rho, d]. \end{aligned}$$

□

Theorem 23.1 (soundness). If

$$\vdash s = t,$$

then in every structure

$$M \models s = t.$$

Proof. By induction on proof of equality $s = t$, using Lemmas 23.1–23.6. □

23.1.3 Partial objects

The formal means of logic ought to be used to build data object model. But in case of using the standard means they are correct only for non-empty domains. A practice of deployment *open information systems* leads to considering the subdomains defined by unsolvable predicates.

Partial elements

In case of the subdomains defined by unsolvable predicates, which is applicable to World Wide Web, the question

is there the objects x *as* an element of this subdomain,

could be unsolvable. As it appears, if we consider more than one sort of variables, then logical means should be conformed to operating on the domains, relatively which it is impossible to know, neither if are they "inhabited", nor if do its elements completely exist.

> A reality arose, when in reasoning with the objects it is not always possible to assume their existence.

In this case a well formed language constructions can mean nothing.

Predicate of existence

To the aims of considering the terms which could have no value — the meaningless, — a special *predicate of existence* E is introduced. For a term t the construction

$$\mathsf{E}t$$

is read as 't exists'. It is assumed that free variables of the sorts of t range the domain of *potential*, or *possible* elements, that are selected out of the *implicit* outer domain.

As known, any domain D of the partial elements can be represented as a subdomain of some domain $\widetilde{A}$ of *total* elements, and these

last are considered as an objectification of possible elements of the set D. Accepting this point of view means the following:

the predicate of physical existence E enables a selection of *actual* in D elements.

The usual assumption, used in logic is that the bound variables range only over the actual elements. This means that bound variables are considered as *restricted* by the predicate E.

Similarity between partial elements

The establishing of any domain assumes that the sentences concerning the *similarity* of its objects within this domain can be constructed. As the partial elements are used, then the equality between them is considered in the following sense:

$$(t = s) \supset (\mathsf{E}t \,\&\, \mathsf{E}s),$$

i.e. the established equality of partial elements implies their existence.

Equality between partial elements

A deployment of representation of the *equality* between elements begins with that all the elements outside the predicate E are assumed as non-existent are considered as equal is their non-existence. More formally, the biconditional

$$t \equiv s \Leftrightarrow (\mathsf{E}t \supset t = s) \,\&\, (\mathsf{E}s \supset t = s)$$

is accepted. Thus, the equal elements become just the same.

Extensionality

A demand of correctness of the predicate definition assumes that they are extensional not only relatively identity, but also relatively equality.

The other name of this principle is the *substitutivity of equal* ones:

$$[t/x]\phi \,\&\, t = s \supset [s/x]\phi. \qquad (ext)$$

Essentially, the relation of equality $\equiv$ and predicate of existence E are selected as the initial ones, and equality become derived and is defined by the following biconditional

$$t = s \Leftrightarrow t \equiv s \,\&\, \mathsf{E}t \,\&\, \mathsf{E}s.$$

Sorts

The logical means of building data object model are given by a *many-sorted* theory with higher types. It means that higher order structures are allowed to build. To achieve this aim any finite sequence of sorts

$$(T_1, T_2, \ldots, T_n)$$

is assigned the sort

$$[T_1, T_2, \ldots, T_n],$$

which is assumed as *power sort* of all the n-ary relations over the given sequence of domains. As turns out, the rest of type constructions can be expressed by the power sorts.

Descriptions

In building the logical means, the *descriptions* – the constructions 'such ...that ...', – are used among the terms. Such constructions are suitable, and their usage does not lead to the difficulties, because there is a predicate of existence. In this case there is a set of terms with the rich expressive power, which can be used in correspondence with the aims to reach, as was shown in (D. Scott, [114]) and (M. Fourman, [8]). Thus introduced term

$$\mathbf{I}x\Phi$$

canonically is read as

'such x that Φ is valid'.

Possible worlds

The *possible worlds* are understood as various sets of individuals with the additional structure or without it. The terms 'element' and 'individual' will be used changeably. The distinct sets of individuals are identified by the *indexed expressions*. As known, an arbitrary system of structures can be indexed. This is done by a choice as the indices the elements of the appropriate set, and this choice can be done by the various ways.

Take some fixed set I of these indices, and also will consider the set of *possible* individuals D and *virtual* individuals V. Next, mark by the indices the system of *actual* individuals A_i, assuming for any index $i \in I$, possibly, distinct $A_i \subseteq D$. It is evident, that the following inclusion

$$A_i \subseteq D \subseteq V$$

is valid, because the virtual individuals are introduced just for increasing the regularity. Note, that between A_i and i there is no assumed one-to-one correspondence, because the elements of the set I can have a structure, which is not completely reflected by the distinctions of different sets A_i each of other.

23.1.4 Data object models

In building the data object models from a formal point of view instead of term 'assignment', used on p. 258 and on p. 262, we will use the term 'reference', that is relative to a representation of the possible world or indexing system.

Concepts, individuals and states

Among the *objects* the concepts, individuals and states will be distinguished. These entities are used as the units — the main building blocks using which a *data object model*, or DOM is built. As awaited, the additional means to formalize this model are needed, and

they are shown in Figure 23.1. An interpretation of correspondences shown in this figure, leads to a formulation of the special approach which can be called the *conceptualization principle*.

Conceptualization principle

The interrelation of concepts, individuals and states allows to formulate the main *principle of conceptualization*as follows:

$$\|(\text{individual concept})\| = \text{function} : \text{references} \to \text{individuals},$$

i.e. a *value* of the individual concept can be considered as the function *from* a set of references *into* a set of individuals.

It means that to describe concepts the formulae are used which identify the functions from references to individuals. In other words, a concept is considered as the process in a mathematical sense. Using the conceptualization principle above, establish a ground to the scheme of investigating the problem domain aimed to choosing and fixing of data objects:

$$\begin{aligned} \|(individual\ concept)\| &\in individual^{\,reference} \\ individual^{\,reference} &\succ\!\!- \quad individual \\ individual &\succ\!\!- \quad state^{\,reference} \\ state^{\,reference} &\succ\!\!- \quad state \end{aligned}$$

In the scheme above the following notations are used:

- symbol '$\succ\!\!-$' indicates the transition from invariant to the family;
- notation $individual^{\,reference}$ means the set of all the mappings from references to individuals, i.e. exponential;
- notation $state^{\,reference}$ means the set of all the mappings from references to states.

The main feature of this scheme is in making a transition from the individual concepts (in a language) to the individual concepts in a domain. Such a transition is performed by the evaluating function $\| \cdot \| \cdot$. Next, the concept in PD[1] is teated as a process, using which the individuals are chosen and fixed. An individual is also considered as the process allowing to indicate the states. The more neutral is the terminology of a kind 'state-metastate', which can be used in place of 'individual-concept' or instead 'state-individual'.

Both the principle of conceptualization and the scheme of investigating the problem domain can be formulated not only at a qualitative level, but as the formal diagram, shown in Figure 23.1. In accordance with this figure, the individual concept is described by the formula Φ, that is additionally equipped with the description operator $\mathbf{I}$. Remind, that the notation

$$\mathbf{I}x\Phi(x)$$

is considered as the descriptive sentence of a kind:

'the (unique) x, such that (it has the property) $\Phi(x)$'.

Describing the objects

It turned out that from an intuitive point of view the description operator, of course, serves the goals of describing one or another object. At first, for such a describing, the evaluation is performed which allows the language construction to be corresponded to the object from a domain. To obtain, by this way, the image of of a language construction, take into account the reference, that is formally indicated by an index. In fixing the individuals of DOM the most important is to conform the condition for $i \in I$:

$$\|\mathbf{I}x\Phi(x)\|i \;=\; d \Leftrightarrow \{d\} = \{d \in D|\; \|\Phi(\overline{d})\|i = 1\},$$

[1] PD – **P**roblem **d**omain.

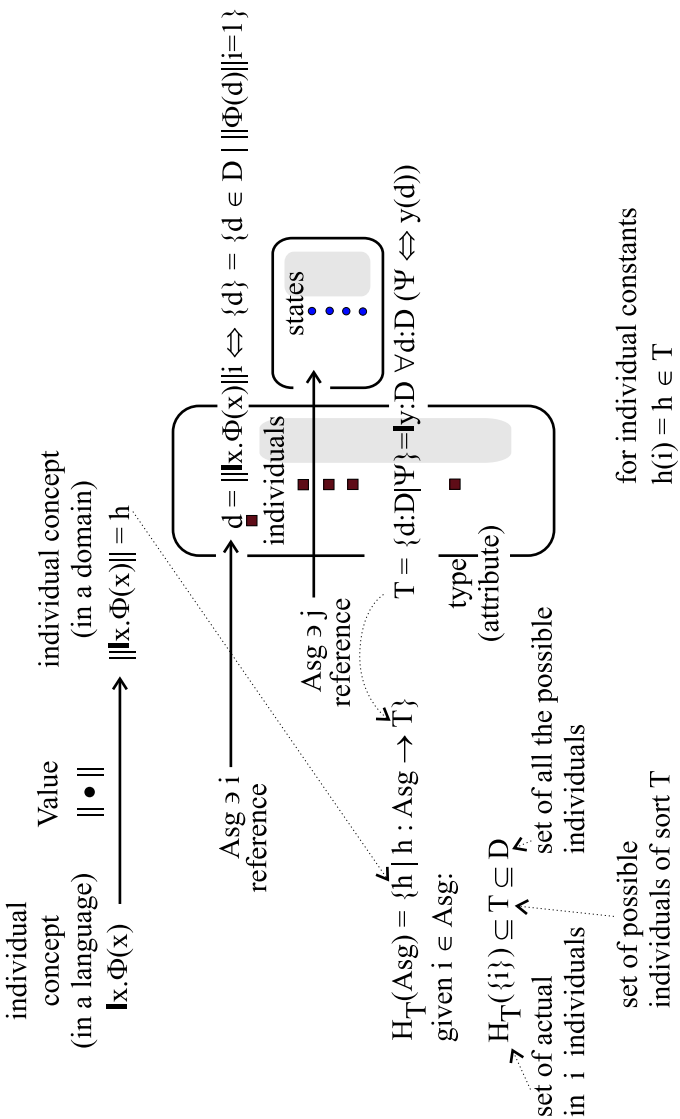

Figure 23.1: Data object model

which is the main characteristic of formalization, where the symbol '1' is used as a truth value *true*. The variants of this principle are well known in mathematics, but they are in a considerable degree less used in the research area of such a subject as data objects. Indeed, all that is said here – this is a singularity of the individualizing the object d by the formula Φ. The other aspect of characteristic of the formalism is the following reason. The descriptions in use have sufficient "selectivity" to choose the individual from a problem domain. According this reason, the characteristic of DOM formalization, besides the fixing individual in a domain, is aimed to establish a connection with the systems of symbolic computations. Development and using of the systems of symbolic computations leads to applying the notion of a function in the sense of definition. Such a research is based on a process of transition from an arguments to the value, when this process is encoded by the definitions. The definitions usually are determined by the sentences of a researcher language. Next they are applied to the arguments, also determined by the sentences of the researcher language. In case of computer systems of symbolic computations, the definitions are understood like the programs, which, in turn, are applied to the programs. Since the objects under research are both the functions and their arguments, then an untyped system arises, that allows the self-applicability of a function, what is considered impossible for the "usual" mathematical functions. As a main untyped system, the λ-calculus is usually used, which is based on the notion of bound variable, or the combinatory logic, which avoids the notion of a variable.

23.2 The main task

Give a rather general formulation of the task that could be reduced to a creation of the *object* representation – the *data object* or *metadata object*.

Task 23.1. Build the object of a given form, using its determining properties Φ.

Task formulation. Establish the definition of an object, using the *principle of comprehension*:

$$C \quad = \quad \mathbf{I}y : [D]\forall x : D(y(x) \leftrightarrow \Phi) = \{x : D \mid \Phi\}, \qquad (C)$$

where $[D]$ denotes a *power set* (a set of all the subsets) of D.

Solution. A solution of the main task is based on the diagram shown in Figure 23.1. A particular application depends on a lot of factors and first of all – on the aims of using the object. Among these objects are the *elementary types* and *indexed concepts*. Besides that, a form of resulting object can be changed depending on the computational model in use. In the following subsections the different variants of the solution of main task are given, and this is a task to construe the data object model.

23.2.1 Elementary types

Consider the constructing of the definitions of the elementary types.

$H_T(I)$–1. *Type.* Usually type is considered as the subset of a set, identified by a sort symbol. Thus, for sort D, the type T is determined by the description

$$T = \mathbf{I}y : [D]\forall x : D(y(x) \leftrightarrow \Phi) = \{x : D \mid \Phi\},$$

for which the inclusion $T \subseteq D \in [D]$ holds.

$H_T(I)$–2. *Relation.* The data object called a relation, is considered as a subset of cartesian product of the domains, identified by the sort symbols. Hence, for sorts A, B the relation R is defined by the description

$$\begin{aligned} R \quad &= \quad \mathbf{I}z : [A, B]\forall x : A\forall y : B(z[x, y] \leftrightarrow \Psi) \\ &= \quad \{[x : A, y : B] \mid \Psi\}, \end{aligned}$$

for which the inclusion $R \subseteq A \times B$ holds. Here, to avoid complications, the definition of two placed relation R is considered.

$H_T(I)$–3. *Value of function.* In developing the data bases a significant attention is paid to the class of relations called the functional relations. To reach this aim the definition

$$R'(t) \;=\; \mathbf{I}y : B.R([t, y])$$

is introduced where t is a term of sort A. In this case the membership $R'(t) \in B$ could be written.

$H_T(I)$–4. *Functional abstraction.* This object is distinguished by an especially often usage in applicative programming systems to indicate the definition of a function. For the variable u of sort A and the term s of sort B the functional abstraction is defined by the description

$$\begin{array}{rcl} \lambda u : A.s & = & \mathbf{I}w : [A, B]\forall u : A\forall v : B(w[u, v] \leftrightarrow v = s) \\ & = & \{[u, v] \mid v = s\}, \end{array}$$

for which the inclusion $\lambda u.s \subseteq A \times B$ holds.

As it turned out, the means of descriptions have a sufficient power – they allow to express the derived operator of abstraction. Note, that in combinatory logic the abstraction operator is also expressed by the combinators. In this sense a certain similarity between the means of combinatory logic and the means of the descriptions can be observed.

23.2.2 Typed variable objects

Give a generalization of the means of typed data objects for the case of definition of the sets that vary with the outer conditions. In this case the *variable concepts* are introduced which are the invariants of the corresponding object set for given conditions.

$H_T(I)$–1. The conditions are taken into account as an index that characterizes the selected reference. The most typical cases of the definitions of the variable concepts are to be the unary type concept and binary relation concept.

$H_T(I)$–2. *Variable type concept.* It arises in considering the pairs reference-individual and corresponds to the description

$$\begin{aligned} C &= C(I) = \mathbf{I}z : [I,T]\forall i : I\forall hi : T(z[i,hi] \leftrightarrow \Phi) \\ &= \{[i,hi] \mid \Phi\} \subseteq \{h|h : I \to T\} \\ &= H_T(I). \end{aligned}$$

In the case above $H_T(I)$ is considered as a set of all the individuals for the references from I and the type T. The definition allows a derivation of one particular case that plays a central role in building the computational models using the λ-calculus. In case we take I as T, and individuals are assumed to be constants, then C maps such an individual h into the singleton $\{h\}$:

$$C : h \mapsto h$$

and, of course, $C(h) \in \{h\}$. This immediately implies the equality

$$C = \mathbf{1}_C : C \to C,$$

i.e. C is the identity mapping $\mathbf{1}_C$ with the property

$$C = C \circ C.$$

$H_T(I)$–3. *Binary relation concept.* As this is the most general case of dependency between the objects, then it is not sufficient to restrict a consideration by the description

$$\begin{aligned} \phi &= \phi(I) \\ &= \mathbf{I}z : [I,(T,\mathcal{T})].\forall i\forall ui : T\forall vi : \mathcal{T}(z[i,[ui,vi]] \leftrightarrow \Phi) \\ &= \{[i,[ui,vi]] \mid \Phi\} \\ &\subseteq \{< u,v > |< u,v >: I \to T \times \mathcal{T}\} \\ &= H_{T\times\mathcal{T}}(I). \end{aligned}$$

The important feature is that u is an element of $H_T(I)$, and v is an element of $H_{\mathcal{T}}(I)$, i.e. $u \in H_T(I)$, $v \in H_{\mathcal{T}}(I)$. Moreover, in defining the description for ϕ the following equalities hold:

$$\begin{array}{rcl}
\phi & = & \phi(I) \\
& = & \mathbf{I}z : [(I,T),(I,\mathcal{T})].\forall i \forall ui \forall vi(z[[i,ui],[i,vi]] \leftrightarrow \Phi) \\
& = & \{[[i,ui],[i,vi]] \mid \Phi\} \\
& \subseteq & \{[u,v] \mid \Phi\} \\
& = & H_T(I) \times H_{\mathcal{T}}(I).
\end{array}$$

Both the expressions for ϕ turned out isomorphic and can be put into the basis of a type system.

23.2.3 Computational models

Consider the typed computational models of data objects. The main aim of type systems of DO[2] is to apply it for simulating of execution either the constructions of DOML[3], or the program that, possibly, uses the constructions of DOML. Consider the way of interpretation the constructions of pure DOML or the constructions of DOML that, possibly, included into the host program, or, visa versa, contain the program (the functions) of programming system. This way is based on the defining and using the families of applicative pre-structures of the kind

$$(\{H_B\}, \{\varepsilon_{BC}\})$$

for arbitrary types B, C of type system of DO, where H is the domain of objects of type B, and ε_{BC} is the applicator of evaluating the functions on the argument from H_B.

$H_T(I)$−1. *Construction of the model.* The steps of building the computational data object model (CDOM) are as follows:

[2] DO is the abbreviation for **D**ata **O**bjects.

[3] DOML is the abbreviation for **D**ata **O**bject **M**anipulation **L**anguage.

• introduce the functional space as explicit objects in a representing category;

• given the objects A B, build the explicit object $(A \to B)$;

• equip it, for using, with the applicator mapping $\varepsilon_{BC} : (B \to C) \times B \to C$ that given $f : B \to C$ and $x : B$, returns $f(x) : C$, i.e. $f, x \mapsto f(x)$;

• in evaluating $h(x, y)$, the mapping operator on two arguments $h : A \times B \to C$, the first of them x is fixed, and $h(x, y)$ is considered as a function on y;

• for connecting h with its values, a special abstractor function Λ_{ABC} is introduced and typed by

$$(\Lambda_{ABC}h) : A \to (B \to C);$$

• given the operator h and argument x, the abstractor takes a form of $(\Lambda_{ABC}h)(x) : B \to C$ that allows to use applicator in evaluating on the second argument.

A significant feature of this CDOM is an ability to use it separately for concepts, individuals and states. At the same time, CDOM can be applied to the object on the whole.

$H_T(I)$–2. *Abstraction by two variables.* In abstracting on two variables, all the features of the approach are revealed that carried over the case of more variables. To shorten the notation of the description, the symbols of sorts are omitted:

$$\lambda[x, y].t \;=\; \lambda z \mathbf{I} w \forall x \forall y (z \;=\; [x, y] \,\&\, t \;=\; w).$$

In turn, w is treated as a relation

$$w \;=\; \mathbf{I} d \forall z \forall \tau [d[z, \tau] \leftrightarrow \Psi]$$

and $\tau \in w'(z)$. Since the term t depending on z takes the values from $w'(z)$, them assume that $t \in w'(z)$. Corresponding ordered pairs $[z, t]$ are put into basis of the definition of the object

$$\lambda z.t \ = \ \lambda[x, y].t,$$

that gives the needed description.

$H_T(I)$–3. *Functional space*. This object serves the aims of representing of the set of functions from type $\mathbf{A}$ into type $\mathbf{B}$ that is written as the description

$$\mathbf{A} \to \mathbf{B} \ = \ \{z : [\mathbf{A}, \mathbf{B}] \mid \forall x \in \mathbf{A} \exists y \in \mathbf{B}.z[x, y]\}$$

or, in more weak form, for sorts A, B and $A \to B$. In the last case the defining formula has been changed to take into account the relation of sorts A, B with the types $\mathbf{A}$, $\mathbf{B}$.

$H_T(I)$–4. *Applicator*. Using two descriptions above, write the description for an evaluating morphism:

$$\varepsilon_{AB} \ = \ \lambda[z, x]\mathbf{I}y.z[x, y],$$

that, given the function $z \in \mathbf{A} \to \mathbf{B}$ and argument $x \in \mathbf{A}$ returns the value $y \in \mathbf{B}$, i.e.

$$\varepsilon_{AB} : [z, x] \mapsto z(x),$$

or, more generally, $\varepsilon_{AB} : [z, x] \mapsto y$.

23.2.4 Indexed objects

Building of CMDOM[4] assumes the choosing and fixing the main building blocks of extensible programming environment. Every of

[4] CMDOM is an abbreviation for **C**omputational **M**eta**D**ata **O**bject **M**odel.

this blocks is the dynamic metadata object in the sense that the referencing or evolving the events is captured. The steps below, aimed to capture dynamics, assume an introducing, by means of the DODL[5], a spectrum of MDO[6] that combines the abilities of ACS and type systems. The most important of these MDO are the indexed concept and indexed relation.

$H_T(I)$−1. *Indexed concept.* Evaluation of the expressions, built as λ-abstractions, is based on the property of extensibility. This property reveals in adding to the environment and tying up the individuals, conforming the body of λ-abstraction. Considering the application $(\lambda.\Phi)\overline{h}$, where Φ is the body of λ-abstraction, h is the individual (individual constant) from a domain, obtain the following equality:

$$\|(\lambda.\Phi)\overline{h}\|i = \|\Phi\|[i, hi]$$

for the reference i. By this way a tying up the individual h with the body of λ-abstraction Φ is reduced to building of the concept, corresponding $\|\Phi\|$ and checking the membership $h \in \|\Phi\|$. Checking up of the membership is done by a selection from the relational Data Manipulation Language. The checking is performed by the following procedure:

1) the reference i in correspondence with the data base is established;

2) the type $\mathbf{T}$ of the individual h, selected from the data base is established;

3) check up if the individual h does conform to the restriction Φ;

4) select all such individuals $h' = hi$ that generate the extension of the concept $C'(i)$ for which the following inclusion holds:

$$C'(i)\ ISA\ \mathbf{T};$$

[5]DODL means **D**ata **O**bject **D**efinition **L**anguage.
[6]MDO means **M**eta**D**ata **O**bjects.

5) all the selected pairs $[i, hi]$ identify the individuals h in the world i of a set I and are accepted as the extension of the variable concept $C(I)$;

6) the natural inclusion $C(I) \subseteq H_T(I)$ for the variable domain $H_T(I)$, defined by

$$H_T(I) \;=\; \{h \mid h : I \to \mathbf{T}\}$$

is preserved.

$H_T(I)$–2. *Dynamic of concept.* The most effective application of the families of indexed concepts is the ability to consider their changes depending on the variations in a domain. Let in a domains the events evolve along the law f where $f : B \to I$, i.e. the transition from the world I to the world B is performed. In this case the applying of the property of extensibility in evaluation of the λ-abstractions has the specific features. For application $(\lambda.\Phi)\overline{h}$ we obtain:

$$\|(\lambda.\Phi)\overline{h}\|i \;=\; \|\Phi\|_f[b, (h \circ f)b]$$

for the reference $i = fb$. Thus, the tying up the individual h with the body of λ-abstraction Φ is not reduced to checking the membership of individual h to the value $\|\Phi\|$, but to checking the membership of the image of h under evolving the events along the law f in the world b for shifted evaluation $\|\Phi\|_f$. Hence, checking of the membership is performed by the modified procedure:

1) the new reference B is established as an alternative to the old reference under evolving the events along the law f;

2) the type $\mathbf{T}$ of an image of the individual h, selected out of the image of data base under the transformation f, is established;

3) check if the image of the individual h does conform the restriction Φ in the world b;

4) select all such individuals $h' = (h \circ f)$ that generate an extension of the concept $C'_f(b)$; the inclusion $C'_f(b)$ ISA $\mathbf{T}$ holds;

5) all the selected pairs $[b, hb]$ identify the individuals h in the world b from the set B and are accepted as an extension of the variable concept $C_f(B)$;

6) the natural inclusion $C_f(B) \subseteq H_T(B)$ for the domain $H_T(B)$, defined by

$$H_T(B) = \{h \mid h : B \to \mathbf{T}\},$$

is preserved.

$H_T(I)$–3. *Static of concept.* The boundary case of the law of evolving the events is the identity (unitary) mapping. Then the notation

$$f = \mathbf{1}_I : I \to I$$

means that the law f of evolving the events does not lead to the changes in a domain. In evaluation of the λ-abstraction obtain:

$$\|(\lambda.\Phi)\overline{h}\|i = \|\Phi\|_{\mathbf{1}_I}[i, hi].$$

Thus, the tying up of the individual h with the body of λ-abstraction of Φ does not need the modified procedure, but is completely similar to the procedure for indexed concept. A studying of concept static leads to an important corollary. Since there are no changes neither in a domain (nor in data base D), then the references I and D can be seen as identical, assuming $I = D$. Then

$$C_{\mathbf{1}_I}(I) : h \to h,$$

i.e. the individual h is transformed to itself and

$$C_{\mathbf{1}_I}(I) = C_{1_D}(D) = 1_D.$$

On the other hand, obtain $C_{\mathbf{1}_I}(I) = C(I)$ and $C_I : C_I \to C_I$, from where the following rule is derived:

indexed concept is similar to identity mapping.

More detailed, for any concepts A, B and the mapping $f : A \to B$ the following equalities

$$A = A \circ A; \quad f = B \circ f \circ A,$$

which are derivable directly from the equalities and the rule above, hold.

$H_T(I)$–4. *Functorial characteristic of concept.* An analysis of dynamic and static of the concepts allows to separate the control action on the concepts, one one hand, the the system of the concepts, on the other hand. It means, that a control performs a switching of the system of concepts depending on the law of evolving the events. The representation of a system of variable concepts leads to the formulation of the *functorial characteristic of the concepts*. A significant is the way of taking into account the law f of evolving the events – C is similar to contravariant functor:

1) the law $f : B \to I$ is assumed as a transition from the world I to the world B, i.e. from the knowledge level I to the knowledge level B etc.;

2) obtain: $f = \mathbf{1}_I \circ f \circ \mathbf{1}_B$ and $C_{\mathbf{1}B} = C(B)$, $C_{\mathbf{1}I} = C(I)$;

3) for the concept $C_f = C_f(B)$ the following inclusion holds:

$$C(f) : C(I) \to C(B);$$
$$C_f \subseteq C(B),$$

because the type checking $\mathbf{T}$ for the individual h and its image under transformation f preserves a natural inclusion for the variable domains:

$$\begin{aligned} C(I) &\subseteq H_T(I) = \{h \mid h : I \to \mathbf{T}\}; \\ C(B) &\subseteq H_T(B) = \{h \mid h : B \to \mathbf{T}\}; \\ C_f &= \{h \circ f \mid h \circ f : B \to \mathbf{T}\} \subseteq C(B). \end{aligned}$$

$H_T(I)$–5. *Indexed relation.* In the above an indexing of the one placed concepts was considered. In this case all the important features, connected with taking into account the evolving of events, were already revealed. However, in case of interaction with the data base the definition and maintenance of multi placed concepts is involved. To simplify the notations consider the indexing of two placed concept, corresponding to the binary relation. Its complete characteristic follows from the consideration of application $(\lambda\lambda.\Phi)\overline{uv}$ for the formula Φ and individuals $\overline{u}$ and $\overline{v}$. It is obtained:

$$\begin{aligned}\|(\lambda\lambda.\Phi)\overline{uv}\|i &= (\|(\lambda\lambda.\Phi)\|i)(\|\overline{u}\|i)(\|\overline{v}\|i)\\ &= \Lambda\|\lambda.\Phi\|i(ui)(vi)\\ &= (\|\lambda.\Phi\|[i,ui])(vi)\\ &= \Lambda\|\Phi\|[i,ui](vi)\\ &= \|\Phi\|[[i,ui],vi]\end{aligned}$$

for the reference i. The other way of reasoning is also possible:

$$\begin{aligned}\|(\lambda.\Phi)[\overline{u},\overline{v}]\|i &= \Lambda\|\Phi\|i(\|[\overline{u},\overline{v}]\|i)\\ &= \Lambda\|\Phi\|i[ui,vi]\\ &= \|\Phi\|[i,[ui,vi]],\end{aligned}$$

in applying which it is assumed that Φ is a two placed operator (uncurried). The tying up the individual $[ui,vi]$ with a body of λ-abstraction Φ takes down to building the concept, corresponding $\|\Phi\|$, and to checking up the membership $[ui,vi] \in \|\Phi\|$. In this case the individual is represented by an ordered pair, which every element is defined over the corresponding one placed concept:

$$\begin{aligned}\|(\lambda.\Psi_1)\overline{u}\|i &= \|\Psi_1\|[i,ui] = \mathbf{U}(\{i\});\\ \|(\lambda.\Psi_2)\overline{v}\|i &= \|\Psi_2\|[i,vi] = \mathbf{V}(\{i\}).\end{aligned}$$

Checking the membership of individual $wi = [ui,vi]$ to the concept $\phi = \|\Phi\|$ is performed by the following procedure:

1) the reference i is established in correspondence with the data base;

2) the types $\mathbf{T}$ and $\mathcal{T}$ of the individuals ui and vi, selected from the data base (and paired) are established;

3) check if the individual wi conforms the restriction Φ;

4) select all such individuals wi that generate an extension of the concept $\phi'(i)$; the following inclusion

$$\phi'(i)\ ISA\ (\mathbf{T} \times \mathcal{T})$$

holds;

5) all the selected pairs $[i, wi]$ identify the individuals w in the world i of the set I and are accepted as the extension of the variable concept $\phi\ =\ \phi(I)$;

6) the natural inclusion $\phi(I)\ \subseteq\ H_{\mathbf{T}\times\mathcal{T}}(I)$ for the variable domain $H_{\mathbf{T}\times\mathcal{T}}(I)$, defined by

$$H_{\mathbf{T}\times\mathcal{T}}(I)\ =\ \{w \mid w : I \to (\mathbf{T} \times \mathcal{T})\}$$

is preserved.

$H_T(I)$–6. *Dynamic of relations.* A reflection of the changes in the domain reveals in the associated changes with the data base. Accepting $f : B \to I$ as a law of evolving the events, apply the property of extensibility in evaluating the λ-abstractions. For the application $(\lambda.\Phi)[\overline{u}, \overline{v}]$ obtain:

$$\|(\lambda.\Phi)[\overline{u}, \overline{v}]\|i\ =\ \|\Phi\|_f[b, (< u, v > \circ f)b].$$

This expression can be written in slightly modified form. The essence of this modifying is the following:

$$\begin{array}{rcl} \|(\lambda.\Psi_1)\overline{u}\|(fb) & = & \|\Psi_1\|_f[b, (u \circ f)b], \\ \|(\lambda.\Psi_2)\overline{v}\|(fb) & = & \|\Psi_2\|_f[b, (v \circ f)b], \end{array}$$
$$u \circ f \in \mathbf{U}_f,\ v \circ f \in \mathbf{V}_f.$$

Then

$$\phi_f \subseteq [\mathbf{U}_f, \mathbf{V}_f]$$

and

$$< u, v > \circ f \in \phi_f$$

for $\phi_f \subseteq (B \times \mathbf{T}) \times (B \times \mathcal{T})$. Therefore, tying up the individual w with the body of λ-abstraction Φ takes down to checking the membership of the image of w under evolving the events along the law f in the world b for shifted evaluation $\|\Phi\|_f$. The details of checking are straightforward and are performed according the following procedure:

1) as an alternative to the reference i the new reference b for the law of evolving the events f is established;
2) the type $\mathbf{T} \times \mathcal{T}$ of the image of individual w, selected from the data base image under the transformation f is established;
3) check if the image of individual w does conform the restriction Φ in the world b;
4) select all such individuals $< u, v > \circ f$ which generate an extension of the concept $\phi'_f(b)$; the inclusion

   $$\phi'_f(b)\ ISA\ \mathbf{T} \times \mathcal{T}$$

   holds;
5) all the selected pairs

   $$[b, (< u, v > \circ f)b]$$

   identify the individuals

   $$< u, v > \circ f$$

   in the world b of the set B, and are accepted as an extension of the variable concept $\phi_f(B) = \phi_f$;

6) the natural inclusions

$$\phi_f(B) \subseteq \mathbf{U}_f(B) \times \mathbf{V}_f(B) \subseteq H_T(B) \times H_{\mathcal{T}}(B)$$

for the variable domain $H_T(B) \times H_{\mathcal{T}}(B)$, defined by

$$H_T(B) \times H_{\mathcal{T}}(B) \;=\; \{\overline{h} \mid \overline{h} : (B \to \mathbf{T}) \times (B \to \mathcal{T})\}$$

are preserved.

23.3 Interpretation of evaluating environment

The evaluation environment, represented by the objects, in practice, usually is equipped with a control structure – the system of *scripts*, – that determines its behavior. The examples of implementations of such structures are well known. From an intuitive point of view, a script is the system of instructions that determines the transitions from one states – the *old states* of computation, – to other ones – the *new states*. This is rather special treating of evaluations, however enabling some suitabilities. Its implementing needs to involve the *parameterized objects*. As a main model of object is taken the representation of a *variable set* – the variable concept, – relatively new in computer science representation. The volume of an intuitive knowledge on what is the object, is replaced by an idealized entity, called *functor-as-object*. As known, the functors allow to operate with the higher order languages when an object is assumed as the mapping *from* objects *into* objects.

The getting acquainted a notion store and associated methods of reasoning as above, not saying of the methods of the implementation, is considered as an ability to work at the "cutting edge" of the studies in an area of programming.

Bibliography

[I] Russian editions:

[1] Aksenov K.E., Balovnev O.T., Wolfengagen V.E., Voskresenskaya O.V., Gannochka A.V., Chuprikov M.Yu. *A guide to practical works "Artificial Intelligence Systems"*. – M.: MEPhI, 1985. – 92 p. (in Russian)

[2] Barendregt H.P. *The lambda calculus. Its syntax and semantics*. – North-Holland Publishing Company: Amsterdam, N.Y., Oxford, 1981.
(Russian edition: Moscow: Mir, 1985)

[3] Belnap N.D., Steel T.B. *The logic of questions and answers*. – Yale University Press: New Haven, C.T., 1976.
(Russian edition: Moscow: Progress, 1981)

[4] Bulkin M.A., Gabovich Yu.R., Panteleev A.G. *Methodical recommendations to programming and exploitaition of the interpretor for algorithmic language Lisp of OS ES*. – Kiev: NIIASS, 1981. – 91 p. (in Russian)

[5] Burge W. *Recursive programming techniques*. – Addison-Wesley: Reading, MA, 1978.
(Russian edition: Moscow: Mashinostroyeniye, 1983)

[6] Curry H.B. *Foundations of mathematical logic.* – McGraw-Hill Book Company, Inc.: N.Y., San Francisco, Toronto, London, 1963.
(Russian edition: Moscow: Mir, 1969)

[7] Engeler E. *Metamathematik der Elementarmathematik.* – Springer-Verlag: N.Y., 1983. – 132 p.
(Russian edition: Moscow: Mir, 1986)

[8] Fourman M.P. *Logic of topoi.* – In: *Handbook of mathematical logic.* – ed. Barwise J., North-Holland Publishing Company: Amsterdam, N.Y., Oxford, 1977.
(Russian edition: Moscow: Nauka, 1983)

[9] Goldblatt R. *Topoi: The categorial analysis of logic.* – North-Holland Publishing Company: Amsterdam, New York, Oxford, 1979.
(Russian edition: Moscow: Mir, 1983)

[10] Henderson P. *Functional programming: application and implementation.* – Prentice-Hall: Englewood Cliffs, N.J., 1980.
(Russian edition: Moscow: Mir, 1983)

[11] Hopcroft J.E., Motwani R., Ullman J.D. *Introduction to automata theory, languages, and computation.* – 2nd ed., Addison-Wesley Publishing Company: Boston, San Francisco, N.Y., 2001.
(Russian edition: Moscow: Izdatelsky dom "Williams", 2002)

[12] Ilyukhin A.A., Ismailova L.Yu., Shargatova Z.I. *Expert systems on the relational basis.* – M.: MEPhI, 1990. (in Russian)

[13] Johnstone P.T. *Topos theory.* – Academic Press, 1977.
(Russian edition: Moscow: Nauka, 1986)

[14] Kleene S.C. *Introduction to metamathematics*. – D. Van Nostrand Company, Inc.: Princeton, N.J., 1952.
(Russian edition: Moscow: IL, 1957)

[15] Kuzichev A.S. *Some properties of the Schönfinkel-Curry's combinators*. – Combinatorniy analiz, vipusk. 1., izdat. MGU, 1971. (in Russian)

[16] Kuzichev A.S. *Deductive combinatory construction of the theory of functionalities*. – DAN SSSR, 1973, Vol. 209, N° 3. (in Russian)

[17] Kuzichev A.S. *Consistent extensions of pure combinatory logic*. – Vestnik Mosk. Univers., matem., mechan., N° 3, p. 76-81, 1973. (in Russian)

[18] Kuzichev A.S. *On the subject and methods of combinatory logic*.– Istoriya i metodologiya estesvennikh nauk, M.: MGU, vipusk 14, 1973, p. 131-141. (in Russian)

[19] Kuzichev A.S. *The system of lambda-conversion with deductive operator of the formal implication*. – DAN SSSR, 212, N° 6, 1973, p. 1290-1292. (in Russian)

[20] Kuzichev A.S. *The deductive operators of combinatory logic*.– Vestnik Mosk. Univers., matem., mechan., N° 3, p. 13-21, 1974. (in Russian)

[21] Kuzichev A.S. *On expressive power of the deductive systems of lambda-conversion and combinatory logic*.– Vestnik Mosk. Univers., matem., mechan., N° 6, p. 19-26, 1974. (in Russian)

[22] Kuzichev A.S. *Principle of combinatory completeness in mathematical logic*. – Istorya i methodologiya estestvennikh nauk, sbornik MGU, vipusk 16, 1974. – p. 106-127. (in Russian)

[23] Kuzichev A.S. *Combinatory complete systems with the operators* Ξ, *F, Q,* Π, $\exists$, *P,* $\neg$, &, $\vee$, $=$. — Vestnik Mosk. Univers., matem., mechan., N° 6, 1976. (in Russian)

[24] Kuzichev A.S. *Operation of substitution in the systems with unrestricted principle of combinatory completeness.* — Vestnik Mosk. Univers., matem., mechan., N° 5, 1976. (in Russian)

[25] Kuzin L.T. *Foundations of Cybernetics, Vol. 2.* — M.: Energiya, 1979, 15-9. (in Russian)

[26] Kuzin L.T. *Foundations of Cybernetics. Vol. 1. Mathematical foundations of Cybernetics: Primary for the institutes.* — 2nd ed., revised and completed. — M.: Energoatomizdat, 1994. — 576 p. (in Russian)

[27] Maltsev A.I. *Algorithms and recursive functions.*— M.: Nauka, 1965. (in Russian)

[28] Markov A.A. *Impossibility of some algorithms in the theory of associative systems.*— DAN SSSR, 1947, Vol. 55, N° 7; Vol. 58, N° 3. (in Russian)

[29] Markov A.A. *On logic of the constructive mathematics.*— Vestnik Mosk. Univers., matem., mechan., N° 2, 7-29, 1970. (in Russian)

[30] Markov A.A. *On logic of the constructive mathematics.*— M.: Znaniye, 1972. (in Russian)

[31] Mendelson E. *Introduction to mathematical logic.* — Princeton: N.J., N.Y., Toronto, London, 1964. — 300 p. (Russian edition: Moscow: Nauka, 1971)

[32] Panteleev A.G. *On interpretor from the Lisp language for ES EVM.* — Programmirovaniye, 1980, N° 3, p. 86—87 (in Russian)

[33] Pratt T.W., Zelkowitz M.V. *Programming languages. Design and implementation.* – 4th ed., Prentice Hall PTR: N.J., 2001.
(Russian edition: St.-Pb.: Piter, 2002)

[34] Schoenfield J.R. *Mathematical logic.* – Addison-Wesley: Reading, MA, 1967.
(Russian edition: Moscow: Nauka, 1975)

[35] Sebesta R. *Concepts of programming languages.* – 4th ed., Benjamin Cummings: Redwood City, CA, 1998.
(Russian edition: Moscow: Izdatelsky dom "Williams", 2001)

[36] Shabunin L.V. *On consistency of some calculi of combinatory logic.*– Vestnik Mosk. Univers., matem., mechan., 1971, N° 6. (in Russian)

[37] Smirnov V.A. (ed.) *Semantics of modal and intensiaonal logics.* – Moscow: Progress, 1981. (In Russian)

[38] Smirnov V.A., Karpenko A.S., Sidorenko E.A. (eds.) *Modal and intensional logics and their applications to the problems of a methodology of science.*– M.: Nauka, 1984.– 368 p. (in Russian)

[39] Stogniy A.A., Wolfengagen V.E., Kushnirov V.A., Sarkisyan V.I., Araksyan V.V., Shitikov A.V. *Development of the integrated data bases.*– Kiev: Technika, 1987. (in Russian)

[40] Takeuti G. *Proof theory.* – North-Holland Publishing Company: Amsterdam, London, 1975.
(Russian edition: Moscow: Mir, 1978)

[41] Wolfengagen V.E. *Computational model of the relational calculus, oriented for knowledge representation .* – M.: preprint MEPhI, 004-84, 1984. (in Russian)

[42] Wolfengagen V.E., Yatsuk V.Ya. *Computational model of relational algebra.* – Programmirovaniye, N° 5, M.: AN SSSR, 1985. – p. 64-76. (in Russian)

[43] Wolfengagen V.E., Sagoyan K.A. *Methodical recommendations to practical classes in the course "Discrete mathematics". The special chapters of discrete mathematics.* – M.: MEPhI, 1987. – 56 p. (in Russian)

[44] Wolfengagen V.E., Aksenov K.E., Ismailova L.Yu., Volshanik T.V. *A guide to laboratory works in course "Discrete mathmetics. Aplicative programming and supporting technolgy for relational systems".* – M.: MEPhI, 1988 . – 56 p. (in Russian)

[45] Wolfengagen V.E., Yatsuk V.Ya. *Applicative computational systems and conceptual method of knowledge systems design.* – MO SSSR, 1987. (in Russian)

[46] Wolfengagen V.E., Chepurnova I.V., Gavrilov A.V., *Methodical recommendations to practical classes in course "Discrete mathematics". Special chapters of discrete mathematics.* – M.: MEPhI, 1990. – 104 p. (in Russian)

[47] Wolfengagen V.E., Goltseva L.V. *Applicative computations based on combinators and λ-calculus.*– (The leader of project "Applicative computational systems" Dr. L.Yu. Ismailova.) – M.: MEPhI, 1992. – 41 p. (in Russian)
The basics of applicative computational systems are covered by the elementary means that provides the students and postgraduate students with short and sound guide that can be used for the 'first reading'. During several years this guide in various versions was used in the practical works and laboratory works in the corresponding partitions of computers science. The topics of using the combinators and λ-calculus in implementing the applicative computations are

covered. The needed minimal theoretical background is included, and the main attention is paid to solving the exercises that explicate the main computational ideas, notions and definitions. To make the learning easier, the guide is equipped with the interactive program which can be used to support the introductory practical works. When using this Teaching Program, it should be taken that the solving the problems assumes the additional transformations of the epressions aimed to make the optimizations, eliminate the variables, and simplify the target executable expression. The applicative computations are delivered as a set of such methods and means.
For the students and postgraduate students of all the specialities. It can be used for the initial self studying of the subject.

[48] Wolfengagen V.E. *Theory of computations*. — M.: MEPhI, 1993. — 96 p. (in Russian)

[49] Wolfengagen V.E. *Categorical abstract machine*.— M.: MEPhI, 1993. — 96 p.; — 2nd edition — M.: "Center JurInfoR", 2002. — 96 p. (in Russian)
The main attention is paid to analyse in depth the evaluation of the programming language constructions. The topics of compiling the code, its optimizing and execution are covered using the environment of the categorical abstract machine. The series of examples of increasing complexity are used.

[50] Wolfengagen V.E. *Programming languages design and a theory of computations*. — M.: MEPhI, 1993. — 189 p. (in Russian)

[51] Wolfengagen V.E. *Programming languages constructions. Methods of description*.— M.: "Center JurInfoR", 2001. — 276 p. (in Russian)
This book covers the basics concerning the development, implementation and application of the constructions both of imperative and

functional programming languages. An attention is paid to the denotational semantic allowing to reveal completely all the advantages of the object-oriented approach, that, after all, gives an ability to construe the resulting computational model of pure functional type.
The detailed solutions for the exercises are included that are carefully commented to make it easier to learn the implementations of the constructions of various programming languages.
This book can be used as a guide to the subject. It can be useful for the students, postgraduate students and the professionals in computer scince, information technologies and programming.

[52] Wolfengagen V.E. *Logic. Synopsis of the lectures: Reasoning techniques*. — M.: "Center JurInfoR", 2001. — 137 p. (in Russian)
This book covers the ways of rewriting the factual text into the symbolic language allowing the using of classic logical means. The ways and methods to use and validate the argumentation are covered. The numerous examples help to study the means of logical reasoning, inference and proof. The ways of using the comments in an inference are indicated that can assist to establish the truth or false of the demonstrated reasons.
For the students and postgraduate students of the humanities. It can be used for the initial study of the subject and also for self studying.

[53] Yanovskaya S.A. *Foundations of mathematics and mathematical logic.*— Mathematics in the USSR for thirty years. 1917-1947.— M.-L.: Gostechizdat, 1948. (in Russian)

[54] Zakharyaschev M.V., Yanov Yu.I. (eds.) *Mathematical logic in programming.*— M.: Mir, 1991. — 408 p. (in Russian)

[II] English editions:

[55] *Nested relations and complex objects in databases.*— Lecture Notes in Computer Science, 361, 1989.

[56] Amadio R.M., Curien P.-L. *Domains and lambda-calculi.*— Cambridge University Press, 1998. — 484 p.
This book describes the mathematical aspects of the semantics of programming languages. The main goals are to provide formal tools to assess the meaning of programming constructs in both a language-independent and a machine-independent way, and to prove properties about progrms, such as whether they terminate, or whether their result is a solution of the problem they are supposed to solve.
The dual aim is pursued: to provide the computer scientists to do some mathematics and to motivate of interested mathematicians in unfamiliar application areas from computer science.

[57] Appel A. *Compiling with continuations.*— Cambridge University Press, 1992.

[58] Avron A., Honsel F., Mason I., and Pollak R. *Using typed lambda-calculus to implement formal systems on a machine.*— Journal of Automated Reasoning, 1995.

[59] Backus J.W. *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.*— Comm. ACM, 1978, v.21, N° 8, p. 614-641.

[60] Backus J.W. *The algebra of functional programs: functional level reasoning, linear equations and extended definitions.*— Int. Col. on formalization of programming concepts, LNCS, v. 107, 1981, pp. 1-43.

[61] Backus J.W., Williams J.H., Wimmers E.L. *FL language manual (preliminary version).*— IBM research report No RJ 5339(54809), 1987.

[62] Banerji R.B. (ed.) *Formal techniques in artificial intelligence: a sourcebook.*— Studies in computer science and artificial intelligence, 6, North-Holland, 1990.

[63] Beery G., Levy J-J. *Minimal and optimal computations of recursive programs.*— J. Assoc. Comp. Machinery, Vol.26, No 1, 1979.

[64] Belnap N.D. (Jr.) *A useful four-valued logic.* Modern uses of multiple-valued logic.— Epstein G.,Dunn J.M. (eds.) Proceedings of the 1975 International Symposium of multiple-valued logic, Reidel, 1976.

[65] Belnap N.D. (Jr.) *How a computer should think.*— Contemporary aspects of philosophy, Proceedings of the Oxford International Symposium, 1976.

[66] Bird R.S. *An introduction to the theory of lists.*— Logic programming and calculi of descrete design (ed. Broy M.), Springer-Verlag, 1986, pp. 5-42.

[67] Böhm C. (ed.) *Lambda calculus and computer science theory.*— Proceedings of the Symposium held in Rome. March 25-29, LNCS, vol.37, Berlin: Springer 1975.

[68] Bonsanque M. *Topological dualities in semantics.*— PhD thesis, Vrije Universiteit Amsterdam, 1996.

[69] Bunder M.V.W. *Set Theory based on Combinatory Logic.*— Doctoral thesis, University of Amsterdam, 1969.

[70] Bunder M.V.W. *Propositional and predicate calculuses based on combinatory logic.*— Notre Dame Journal of Formal Logic, Vol. XV, 1974, pp. 25-34.

[71] Bunder M.V.W. *The naturalness of illative combinatory logic as a basis for mathematics.*— To H.B.Curry: Essays

on combinatory logic, lambda calculus and formalism.— Seldin J.P., Hindley J.R. (eds.), Academic Press, 1980, pp. 55-64.

[72] Bucciarelli A. *Logical reconstruction of bi-domains.*— In: Proc. Typed Lambda Calculi and Applications, Springer Lecture Notes in Comp. Sci., 1210, 1997.

[73] Church A. *The calculi of lambda-conversion.*— Princeton. 1941, ed. 2, 1951.

[74] Coppo M., Dezani M., Longo G. *Applicative information systems.*— LNCS, 159, 1983, pp. 35-64.

[75] Cousineau G., Curien P.-L., Mauny M. *The categorical abstract machine.*— LNCS, 201, Functional programming languages computer architecture.— 1985, pp. 50-64.

[76] Curien P.-L. *Categorical combinatory logic.*— LNCS, 194, 1985, pp. 139-151.

[77] Curien P.-L. *Typed categorical combinatory logic.*— LNCS, 194, 1985, pp. 130-139.

[78] Curry H.B., Feys R. *Combinatory logic.*— Vol. 1. Amsterdam: North-Holland, 1958.

[79] Curry H.B., Hindley J.R., Seldin J.P. *Combinatory logic.*— Vol. II. Amsterdam, 1972.

[80] Curry H.B. *Some philosophical aspects of combinatory logic.*— Barwise J., Keisler H.J., Kunen K. (eds.) The Kleene Symposium.— North-Holland Publ. Co, 1980, p. 85-101.

[81] Danforth S., Tomlison C. *Type theories and object oriented programming.*— ACM Computing Surveys, 1988, v.20, No 1, pp. 29-72.

[82] Darlington J., Henderson P., Turner D.A. (eds.) *Functional programming and its applications.*— Cambridge Univ. Press, Cambridge, 1982.

[83] de Bruijn N.G. *Lambda-calculus notations with nameless dummies: a tool for automatic formula manipulation.*— Indag. Math. 1972, N 34, pp. 381-392.

[84] Eisenbach S. (ed.) *Functional programming: languages, tools and architectures.*— Chichester: Horwood, 1987.

[85] Fasel J.H., Keller R.M. (eds.) *Graph reduction.*— LNCS, 279, 1986.

[86] Fenstad J.E. et al. *Situations, language and logic.* — Dordrecht: D. Reidel Publ. Comp., 1987.

[87] Fitting M. *First-order logic and automated theorem proving.*— Springer-Verlag, 1990.

[88] Frandsen G.S., Sturtivant C. *What is an efficient implementation of the lambda-calculus?*— LNCS, 523, 1991, pp. 289-312.

[89] Friedman H. *Equality between functionals.*— Proceedings of the Symposium on Logic. Boston, 1972—1973.— Lecture Notes in Mathematics, 453, 1975, pp. 22-37.

[90] Gardenfors P. *Induction, Conceptual spaces and AI.*— Proceedings of the workshop on inductive reasoning, Riso National Lab, Rpskilde, 1987.

[91] Henson M.S. *Elements of functional languages.*— Exford: Blackwell, 1987.

[92] Hindley J.R. *The principial type-scheme of an object in combinatory logic.*— Trans. Amer. Math. Soc., 1969, vol. 146.

[93] Hindley J., Lercher H., Seldin J. *Introduction to combinatory logic.*— Cambridge University Press, 1972.
A general introduction which goes fairly deep; primarily mathematical.

[94] Howard W. *The folmulas-as-type notion of construction.*— Seldin J.P., Hindley J.R. (eds.), To H.B.Curry: Essays on combinatory logic, lambda-calculus and formalism.— Amsterdam: Academic Press, 1980.

[95] Hughes R.J.M. *Super combinators: a new implementation method for applicative languages.*— Proceedings of the 1982 ACM symposium on LISP and functional programming, pp. 1-10.

[96] Hughes R.J.M. *The design and implementation of programming languages.*— PhD Thesis, University of Oxford, 1984.

[97] Hunt L.S. *A Hope to FLIC translator with strictness analysis.*— MSc dissertation, Department of Computing, Imperial College, University of London, 1986.

[98] Kelly P.H.J. *Functional languages for loosely-coupled multiprocessors.*— PhD Thesis, Imperial College, University of London, 1987.

[99] Lambek J., Scott P.J. *Introduction to higher order categorical logic.* — Cambridge Studies in Advanced Mathematics 7, Cambridge University Press, 1986, 1988, 1989, 1994. — 293 p.

[100] Landin P. *The next 700 programming languages.*— Communications of the ACM, 3, 1966.

[101] McCarthy J. *A basis for a mathematical theory of computation.*— Computer programming and formal systems (eds.:

Braffort and Hirshberg), Amsterdam: North-Holland, 1963, pp. 33-69.

[102] Michaelson G. *An introduction to functional programming through lambda-calculus*. — Addison Wesley Publ.Co, 1989, 320 p.

[103] Milner R., Parrow J., Walker D. *A calculus of mobile process*. — Parts 1-2. — Information and Computation, 100(1), 1992, pp. 1-77

[104] Mycroft A. *Abstract interpretation and optimizing transformations for applicative programmes*.— PhD Thesis, Department of Computer Science, University of Edinburgh, 1981.

[105] Peyton Jones S.L. *The implementation of functional programming languages*.— Prentice Hall Int., 1987.

[106] Pitts A. *Relational properties of domains*. — Information and Computation, 127, 1996, pp. 66-90.

[107] Rosser J.B. *A mathematical logic without variables*.— Ann. of Math., vol. 36, No 2, 1935. pp. 127-150.

[108] Schönfinkel M. *Uber die Baustein der mathematischen Logik*.— Math. Annalen, vol. 92, 1924, pp. 305-316.

[109] Schroeder-Heister P. *Extentions of logic programming*.— LNAI, 475, 1991.

[110] Scott D.S. *Advice on modal logic*.— Philosophical problems in logic. Some recent developments.— Lambert K. (ed.), Dordrecht; Holland: Reidel, 1970.

[111] Scott D.S. *Outline of a mathematical theory of computation*.— Proceedings of the 4-th Annual Princeton conference on information sciences and systems, 1970.

[112] Scott D.S. *The lattice of flow diagrams.*— Lecture Notes in Mathematics, 188, Symposium on Semantics of Algorithmic Languages.— Berlin, Heidelberg, New York: Springer-Verlag, 1971, pp. 311-372.

[113] Scott D.S. *Some phylosophycal issues concerning theories of combinators.* — Lambda Calculus and Computer Science Theory. — Böhm C. (ed.), Proceedings of the Symposium held in Rome, March 25-26, 1975, Lecture Notes in Computer Science, Vol. 37, Springer: Berlin, 1975. — pp. 346-370.

[114] Scott D.S. *Identity and existence in intuitionistic logic.*— In: Applications of Sheaves. Berlin: Springer, 1979, pp. 660-696.

[115] Scott D.S. *Lambda calculus: some models, some philosophy.*— The Kleene Symposium. Barwise, J., et al.(eds.), Studies in Logic 101, North-Holland, 1980, pp. 381-421.

[116] Scott D.S. *Relating theories of the lambda calculus.*— Hindley J., Seldin J. (eds.) To H.B.Curry: Essays on combinatory logic, lambda calculus and formalism.— N.Y.& L.: Academic Press, 1980, pp. 403-450.

[117] Scott D.S. *Lectures on a mathematical theory of computation.*— Oxford University Computing Laboratory Technical Monograph PRG-19, 1981. — 148 p.

[118] Scott D.S. *Domains for denotational semantics.*— LNCS, 140, 1982, pp. 577-613.

[119] Stoy J.E. *Denotational semantics: The Scott-Strachey approach to programming language theory.*— M.I.T. Press, Cambridge, Mass., 1977.— xxx+414 p.

[120] Stoye W.R. *The implementation of functional languages using custom hardware.*— PhD Thesis, University of Cambridge, 1985.

[121] Szabo M.E. *Algebra of proofs.*— Studies in Logic foundations of mathematics, v. 88. North-Holland Publ. Co, 1978.— 297 p.

[122] Talcott C. *Rum: An intensional theory of function and control abstractions.*— Foundations of logic and functional programming, LNCS, 306, 1986, pp. 3-44.

[123] Tello E.R. *Object-Oriented Programming for Windows / Covers Windows 3.x.*— Wiley and Sons, Inc., 1991.

[124] Turner D.A. *A New Implementation Technique for Applicative Languages.*— Software Practice and Experience.— No 9, 1979, pp. 31-49.

[125] Turner D.A. *Aspects of the implementation of programming languages.*— PhD Thesis, University of Oxford, 1981.

[126] Turner R. *A theory of properties.*— J. Symbolic logic, v. 52, 1987, pp. 455-472.

[127] Wodsworth C.P. *Semantics and pragmatics of the lambda calculus.*— PhD Thesis, University of Oxford, 1981.

[128] Wolfengagen V.E. *Frame theory and computations.*— Computers and artificial intelligence. V.3, No 1, 1984, pp. 1-31.

[129] Wolfengagen V.E. *Building the access pointers to a computational environment.* — In: electronic Workshops in Computing, Berlin Heidelberg New York: Springer-Verlag, 1998. pp. 1-13
http://ewic.springer.co.uk/adbis97/

[130] Wolfengagen V.E. *Event driven objects*. — In: Proceedings of the 1st International Workshop on Computer Science and Information Technologies, Moscow, Russia, 1999, Vol. 1. pp. 88-96

[131] Wolfengagen V.E. *Functional notation for indexed concepts*. — In: Proceedings of The 9th International Workshop on Functional and Logic Programming WFLP'2000, Benicassim, Spain, September 28-30, 2000
http://www.dsic.upv.es/~wflp2000/

[132] Zhang *The largest cartesian closed category of stable domains*. — Theoretical Computer Science, 166, 1995, pp. 203-219.

[133] http://www.rbjones.com
This Web-resource FACTASIA is aimed "to develop a vision for our future and to provide recourses for building the vision and the future", and also "to contribute to the values which shape our future and to the technology which helps us build it". The partition "Logic" includes *combinatory logic* and *λ-calculus*, bibliography.
See http://www.rbjones.com/rbjpub/logic/cl/

[134] http://ling.ucsd.edu/~barker/Lambda/ski.html
This is an easy on-line introduction in combinatory logic.

[135] http://www.cwi.nl/~tromp/cl/cl.html
The guide, bibliography and Java-applet are represented which allow to interpret the objects — expressions of an applicative language, — that are to be build by the rules of combinatory logic.

[136] http://www.brics.dk
BRICS — Basic Research in Computer Science.
This is a Research Center and International Postgraduate School. In the chapter "BRICS Publications" the "Lecture Series" is presented

that covers the courses in 9 main areas: *discrete mathmatics*, *semantic of evaluations*, *logic in computer science*, *computation complexity*, *constructing and analysis of algorithms*, *programming languages*, *verification*, *distributed computations*, *criptology* and *data security*.

[137] http://www.afm.sbu.ac.uk
Formal Methods.
This document contains the references to available Web-resourses on formal methods, that are useful for mathematical description and analysis of the properties of computer systems. The attention is paid to such formal methods that are useful to decrease the number of errors in the systems, especially at the early stages of development. These methods are complementary to such a method of error detection as testing.

[138] http://liinwww.ira.uka.de/bibliography/
The Collection of Computer Science Bibliographies.
This collection is accumulated from the different sources and covers many aspects of computer science. The bibliography is renewing monthly, thus containing the fresh versions. More than 1.2 mln. references to the journal papers, conference proceedings and technical reports are partitioned in approximately 1400 bibliographies. More than 150000 references contain URLs to electronic copies of the papers available on-line.

[139] http://www.ncstrl.org
NCSTRL — Networked Computer Science Technical Reference Library.
This library is created in a framework of the Open Archives Initiative (http://www.openarchives.org).

[140] http://xxx.lanl.gov/archive/cs/intro.html
CoRR — The Computing Research Repository.
Contains the papers from 1993 in the computer science area, that are

classified in two ways: by subject and using the ACM Classification Scheme in computing of 1998. The ACM Classification Scheme is stable and covers all the partitions of computer science. The subjects are not pairwise distinct and do not cover all the topics. Nevertheless they reflect in computer science the areas of an active research.

[141] http://web.cl.cam.ac.uk/DeptInfo/CST/node13.html
Foundations of Computer Science.
The lecture notes of University of Cambridge aimed to cover the basics in programming.

[142] http://web.comlab.ox.ac.uk/oucl/courses/topics01-02/lc
Lambda Calculus.
The lecture notes of Oxford University that cover the formal theory, rewriting systems, combinatory logic, Turing's completeness, and type systems. This is a short introduction to many branches of computer science, revealing the connections with lambda-calculus.

Index

$Curry$, 147
$unCurry$, 148
'+1'
  $\sigma = \lambda xyz.xy(yz)$, 89
  $\widehat{\sigma} \equiv [xyz](y(xyz))$, 82

Axiom
  (FI), 96
  (FK), 96
  (FS), 96

Basis
  I, B, C, S, 120
  I, K, S, 114

Cartesian product, 150
Category
  $\mathcal{C}(\mathcal{L})$, 150
  cartesian closed, 161
Characteristic
  combinatory, 46
Code
  categorical, 245
  optimized, 245
Combinator
  B, 37, 51
  B^2, 37, 56
  B^3, 37, 57
  C, 37, 53
  $C^{[2]}$, 37, 58
  $C^{[3]}$, 37, 59
  $C_{[2]}$, 37, 58
  $C_{[3]}$, 37, 60
  F, 38
  $I = \lambda x.x$, 157
  I, 35
  $K = \lambda xy.x$, 157
  P, 38
  W, 37, 54
  Y, 37, 61, 68–71, 200
  Y_0, 38, 70, 71
  Y_1, 38, 70, 71
  $\&$, 38
  Φ, 37, 61
  Ψ, 37, 55
  Ξ, 38
  $\exists$, 38
  $\neg$, 38
  $\vee$, 38
  addition $\mathbb{A}$, 85
  cancellator, K, 50
  compositor, B, 50
  connector, S, 50
  cut subtraction $\mathbb{L}$, 84
  duplicator, W, 50
  identity, I, 50
  minimum $\overline{\min}$, 84

multiplication $\mathbb{M}$, 86
pairing $\mathcal{D}$, 83
permutator, C, 50
predecessor π, 83
successor $\widehat{\sigma}$, 82
symmetric difference $\mathbb{R}$, 85
Completeness
combinatory, 75
Composition
$f \circ g$, 150
Concept, 271
individual, 271
of predicate, 155
Constructor
Append, 164
Car, 164
Cdr, 164
List, 164
Nil, 164
Null, 164
else, 128
fi, 211
hd, 201
if, 128, 201
if-FALSE, 214
if-TRUE, 214
in, 221
let, 221
then, 128
tl, 201
where, 221
Couple
$< f, g >$, 254
$< x, y >$, 140
Currying
Curry, 147

de Bruijn's encoding
$\underline{0}$, 226
$\underline{1}$, 226
Description
I, 76
Domain
$H_T(I)$, 275

Equality
(ac), 229
(ass), 229
$(dpair)$, 229
(fst), 229
$(quote)$, 229
(snd), 229
$< Id, g >=< g >$, 246
$< M, N > w = (Mw, Nw)$, 235
$< f, g > = \lambda t.[f(t), g(t)]$, 140
$[x, y] = \lambda r.rxy$, 140
$\lambda xy.xy = \lambda x.x$, 77
$a \circ b = \lambda x.a(bx)$, 149
$id\ a = a$, 150

Function
$Curry_{ABC}$, 145
Λ_{ABC}, 140
$\Lambda_{ABC}\ h$, 140
$\oplus$, 222
$\varepsilon \circ < k \circ p, q >: A \times B \to C$, 140
ε_{BC}, 140
el, 201, 211
fac, 202, 214, 219
$h : A \times B \to C$, 145
plus, 221
unCurry, 148
predicate, 155

Individual, 271
Instruction
car, 238
cdr, 238
cons, 238
cur, 238
push, 238
quote, 238
skip, 246
swap, 238

Karoubi's shell
$\mathcal{L}$, 149

Language
Lisp, 164

Number
combinatory Z_0, 82
Numeral
$\bar{n} = (SB)^n(KI)$, 87
$\bar{n} = \lambda xy.(x^n)y$, 87
$\overline{0}$, 88
$\overline{1}$, 88

Object
Append, 43
Car, 43
Cdr, 43, 89
Fst, 240, 248
List, 43
Nil, 43, 90
Null, 43, 89
Snd, 240, 248
Y, 129
Λ, 41, 230, 240
$\mathcal{S}$, 240
$\mathcal{S}$, 229
ε, 41, 230, 240
$\varepsilon_{BC} : (B \to C) \times B \to C$, 151
append, 41, 128, 135
car, 127, 134, 135
cdr, 128, 134, 135
concat, 41, 128, 135
false, 134
$h : (A \times B) \to C$, $\Lambda_{ABC}h : A \to (B \to C)$, 151
identity, 135
length, 41, 89, 128, 135
list, 134
*list*1, 39, 133
map, 41, 128, 135
null, 129, 134
postfix, 135
product, 41, 128, 135
reverse, 135
sum, 41, 128, 135
sumsquares, 135
times, 127, 202
true, 134
arithmetical
Z_0, 82
$\mathbb{A}$, 85
$\mathbb{L}$, 84
$\mathbb{M}$, 86
$\mathbb{R}$, 85
$\overline{\min}$, 84
π, 83
$\widehat{\sigma}$, 82
modes of combining, 18

Pair
$[f, g]$, 254
$[x, y] = \lambda r.rxy$, 157
$[x, y]$, 140

Pairing
$[x, y] = \lambda r.rxy$, 145
Parameter
actual, 48
formal, 48
substitutional, 48
Postulate
alpha, α, 35
eta, η, 77
mu, μ, 35
nu, ν, 35
sigma, σ, 35
xi, ξ, 35
Predicate
variable, 156
Principle
$(Beta)$, 245
$[\oplus]$, 248
of comprehension, 155, 275
Projection
$p : A \times B \to A$, 140
$q : A \times B \to B$, 140

Recursion
stack, 65
Reference, 271
Rule
(F), 96, 97
(λ), 97
characteristic, 46

Semantic
denotational, 250
State, 271
Supercombinator, 249
alpha, 202, 213
beta, 202, 213
gamma, 202, 213

Term
$\lambda x.P$, 40
$\lambda x.PQ$, 40
Terms
$\lambda V.E$, 200
Theory
metatheory, 11
subtheory, 11
Type
$\#(B)$, 99
$\#(B^2)$, 104
$\#(B^3)$, 105
$\#(C)$, 111
$\#(C^{[2]})$, 105
$\#(C^{[3]})$, 106
$\#(C_{[2]})$, 107
$\#(C_{[3]})$, 108
$\#(D)$, 111
$\#(SB)$, 100
$\#(W)$, 103
$\#(X)$, 95
$\#(Y)$, 109
$\#(Z^n)$, 102
$\#(Z_0)$, 101
$\#(Z_1)$, 101
$\#(\Phi)$, 109

Glossary

Algebra

Often algebra means a system, which does not use the bound variable at all, i.e. all the variables in use are free.

Algorithm (informally)

Algorithm is a deterministic procedure that can be applied to any element of some class of symbolic *inputs* and that for every such input results, after all, in corresponding symbolic *output*.

The essential features of the algorithm:

*1) algorithm is given by a set of instructions of finite size;

*2) there is a computing device that is able to proceed with instructions and perform the computations;

*3) there is an ability to choose, store and repeat the steps of computations;

*4) let P be a set instructions in accordance with *1), and L be a computing device of *2). Then L interacts with P so, that for any given input the computation is performed discretely by steps, without analog devices and methods;

*5) L interacts with P so that the computations move forward in a deterministic way, without access to stochastic methods or devices, e.g. dice.

Alphabet

Alphabet is a set of objects, called *symbols* or letters, which has a property of unlimited reproducing (in written form).

Applicative computational systems

Usually, the *applicative computational system*, or ACS, includes the systems of object calculi based on combinatory logic and lambda-calculus. The only that is essentially developing in these systems, is a representation of an *object*. Combinatory logic contains the only metaoperator – *application*, or, in other terms, *action* of one object on another. The lambda-calculus contains two metaoperators – *application* and functional *abstraction* that allows to bind one variable in the one object.

The objects, generated in these systems, have a behavior of the functional entities with the following features:

the number of argument places, or arity of an object is not fixed in advance, but reveals step by step, in interactions with other objects;

in building the compound object one of the generic objects – the function, – is applied to the other one – to argument, – while in other contexts they can change their roles, i.e. the function and arguments are considered on equal rights;

the self-applicability of functions is allowed, i.e. an object can be applied to itself.

Axiomatic set theory

The characteristics of such a theory are the following (see, e.g., [6]): (1) propositional functions are considered extensionally (the functions having the same truth values on the same arguments are identical); (2) propositional functions more that of one argument are reduced to the propositional functions of one argument, i.e. to classes; (3) there is the class which elements

are called the sets; a class can be the element of other class if and only if it is a set; (4) the sets are characterized genetically, according to their building, so that too big classes, e.g. class of all the sets, could not be the sets.

Calculus

The calculus is a system with bound variables. In particular, λ-calculus uses bound variables, and the only operator binding a variable, i.e. transforming a variable into formal parameter, is the operator of functional abstraction λ.

Category

A *category* E contains the *objects* X, Y, ...and *arrows* f, g, ... Any arrow f is attached to the object X, called the *domain* and the object Y, called *codomain*. This is written as $f : X \to Y$. In addition the restrictions on using a composition are imposed – taking into account the unitary (identity) map. The arrows are viewed as the representations of mappings. Moreover, if g is an arbitrary arrow $g : Y \to Z$ with the domain Y that is the same as the codomain of f, then there is an arrow $g \circ f : X \to Z$, called a *composition* of g with f. For any object Y there is an arrow $1 = 1_Y : Y \to Y$, called the identity arrow for Y. The axioms of identity and associativity are assumed to be valid for all the arrows $h : Z \to W$:
$1_Y \circ f = f$, $g \circ 1_Y = g$, $h \circ (g \circ f) = (h \circ g) \circ f : X \to W$.

Class

A notion of class is assumed to be intuitively clear. The classes of objects are usually considered as some objects. The *proper* classes (e.g., a class of numbers, houses, humans etc.) – are such the classes that are not the members of themselves (e.g., a class of all the notions).

Class (— conceptual)

In a broad sense of the word this is a set of admissible elements

of this class.

Class (— inductive)

The inductive class is a conceptual class generated from the definite generic elements by the selected modes of combining. More rigorously, the class $\mathcal{K}$ is inductive, if:

(1) class $\mathcal{K}$ includes the basis;

(2) class $\mathcal{K}$ is closed relatively the modes of combining;

(3) class $\mathcal{K}$ is a subclass of any class conforming the conditions (1) and (2).

The notion of inductive class is used in two cases:

(1) the elements are the *objects*, and modes of combining are the *operations*;

(2) the elements are the *propositions*, and the combinations are the *connectives*.

Combinator

The combinator is an object that relatively an evaluation reveals the property of being a constant. From a point of view of the λ-calculus the combinator is a closed term.

Combinatory logic

In a narrow sense this is a branch of mathematical logic that studied the combinators and their properties. In the combinatory logic the functional abstraction can be expressed in terms of usual operations, i.e. *without using the formal variables* (parameters).

Construction

The process of deriving the object X belonging to the inductive class $\mathcal{K}$ (see ***Class inductive***) by the iterative use of the combining modes is considered as the *construction* X relatively $\mathcal{K}$.

Definition

The definition traditionally is regarded as an agreement concerning the usage of a language. Then a new symbol or combination of symbols, called *definable*, is introduced with the permission to substitute it in place of some other combination of symbols, called the *defining* one which value is already known from the data and previously introduced definitions.

Definition (— of function)

The functions are introduced by the explicit definitions. Thus, starting with a variable x that denotes arbitrary object of type A, the expression $b[x]$ denoting the object of type $B(x)$ is constructed. Next, the function f of type $(\forall x \in A)B(x)$ with the scheme $f(x) \stackrel{def}{=} b[x]$, where brackets indicate an occurrence of the variable x into the expression $b[x]$, is defined. If $B(x)$ for any object x of type A defines one and the same type B, then in place of '$(\forall x \in A)B(x)$' the abbreviation $A \to B$ is used. The last notation is accepted as the type of functions from A to B.

Definition (— recursive)

The recursive definition of a function is such a definition in that the values of function for given arguments are directly determined by the values of the same function for "simpler" arguments or by the values of "simpler" functions. The notion of 'simpler' is augmented by a choice of formalization – the simplest, as a rule, are all the functions-constants. Such a method of formalization is suitable, because the recursive definitions can be considered as an algorithm (see ***Algorithm***).

Definitional equality

The binary relation of a definitional equality is denoted by '$\stackrel{def}{=}$'. Its left part is *definable*, and its right part is *defining*. This is an equality relation (reflective, symmetric, and transitive).

Description

In building the logical means, among the terms are often used *the descriptions* – the constructions 'such ..., that ...'. The description corresponds to the term $\mathbf{I}x\Phi$ that is canonically read as 'such and the only x, that Φ (for which Φ is true)'.

Evaluation

By the evaluation means such a mapping, when intensional objects are corresponded to the formal objects, and one and the same intensional object can be mapped in one or more different formal objects.

Function

See ***Definition of function***.

Function (— of higher order)

The function is called a 'higher order' function, if its arguments can in turn be the functions or it results in a function.

Function (—, computable by an algorithm)

This is the mapping determined by a procedure, or by *algorithm*.

Function (— primitive recursive)

The class of primitive recursive functions is the least class $\mathcal{C}$ of the total functions such that:

i) all the *constant functions* $\lambda x_1 \dots x_k.m$ are in $\mathcal{C}$, $1 \leq k, 0 \leq m$;

ii) the *successor* function $\lambda x.x + 1$ is in $\mathcal{C}$;

iii) all the *choice functions* $\lambda x_1 \dots x_k.x_i$ are in $\mathcal{C}$, $1 \leq i \leq k$;

iv) if f is a function of k variables of $\mathcal{C}$ and g_1, g_2, ..., g_k are the functions of m variables of $\mathcal{C}$, then the function

$$\lambda x_1 \dots x_m.f(g_1(x_1, \dots, x_m), \dots, g_k(x_1, \dots, x_m))$$

is in $\mathcal{C}$, $1 \leq k, m$;

v) if h is a function of $k+1$ variables of $\mathcal{C}$, and g is a function of $k-1$ variables of $\mathcal{C}$, then the only function f of k variables that conforms the conditions

$$\begin{aligned} f(0, x_2, \ldots, x_k) &= g(x_2, \ldots, x_k), \\ f(y+1, x_2, \ldots, x_k) &= h(y, f(y, x_2, \ldots, x_k), x_2, \ldots, x_k), \end{aligned}$$

is in $\mathcal{C}$, $1 \leq k$.

Note that the 'function of zero variables of $\mathcal{C}$' means a fixed natural number.

Functor (— grammatical)

The functor is regarded as the means to join the *phrases* to generate other phrases.

Infix

Infixes are the binary functors ("connectives", operators) that are written between the arguments.

Interpretation (— of a theory)

By the interpretation of a theory relatively intensional domain (problem domain) we mean many-to-one mapping between the elementary propositions of a theory and definite intensional propositions regarding this intensional domain.

Interpretation (— of a term)

An *evaluation*, or interpretation of the term M in the structure $\mathcal{M}$ is a mapping:

$$\| \cdot \|\cdot : \text{terms} \times \text{references} \rightarrow \text{elements of } \mathcal{M}$$

(see also ***Evaluation***).

Interpretation (— adequate)

An adequate, relatively complete interpretation maps every intensional proposition (interpretant of the intensional domain) into the theorem of a theory.

λ-term (lambda-term)

λ-term, or λ-expression is an object derived by induction on complexity with possible using of the operators of application and abstraction.

Language

The language in a broad sense of the word is determined by the introducing of *agreements*: (1) the *alphabet* is fixed; (2) the rules of constructing the certain combinations of the letters of alphabet are called the *expressions*, or words.

Language (— of researcher, or $\mathcal{U}$-language)

$\mathcal{U}$-language is characterized by the following properties:

(1) *singularity* for every particular context;

(2) presence of the means to *formalize a terminology*;

(3) variability in a sense that it is a *process* relatively joining the new symbols or terms, and the usage of old terms is not necessary unchangeable;

(4) $\mathcal{U}$-language by need is inexact, but using it any reasonable degree of precision can be reached.

Logic

"Logic is an analysis and criticism of thought" (see Johnson W.E. Logic, part I, London, 1921; part II, London, 1922; part III, London, 1924.). When in studying logic the mathematical methods are used then the mathematical systems, that are by the definite ways related to the logic, are constructed. These systems are the subject of self standing studies and are considered as a branch of mathematics. These systems constitute the *mathematical logic*. The mathematical logic deals with a task of explaining the nature of mathematical rigor because the mathematics is a deductive discipline, and the notion of a

rigorous proof is the central notion of all its partitions. Moreover, mathematical logic includes the studies of foundations of mathematics.

Logic (— mathematical)

The mathematical logic describes a new direction in mathematics (see ***Mathematics modern***), paying attention to the language in use, the modes of defining the abstract objects, and the laws of logic that are used in reasoning about these objects. Such a study have been attempted in logic to understand the nature of mathematical experience, enrich the mathematic with the most important results, and also to find out the applications to other branches of mathematics.

Logic (— modern mathematical)

The modern mathematical logic follows from the works of Leibnitz on the universal calculus that can include all the brainwork and, in particular, all the mathematics.

Mathematics (— modern)

The modern mathematics can be described as a science of the *abstract objects* such as the real numbers, functions, algebraic systems etc.

Name

The name calls some actual or imaginable object.

Numeral

Combinatory logic or λ-calculus allows to establish such combinators or, respectively, terms that are similar to numbers. These representations of numbers are called the *numerals*. Numerals as the combinators conform to all the laws of combinatory logic. Moreover, the combinators that represent the arithmetic operations, e.g., addition of numbers, can be constructed.

Ob-system

The formal objects of ob-systems constitute an inductive class. The elements of this class are called the *obs*, or objects. The generic objects of inductive class are the *atoms*, and the modes of combining are the *operations*. The ob-systems are used to find out the significant, invariant assemblies of the objects.

Object

Object is a mathematical entity that is used in a theory. The object is a mathematical *representation* of the physical object of a problem domain ("outer world").

Object (— arithmetical)

Arithmetical objects are the combinatory representations of the numbers – they are numerals and also the corresponding combinatory representations of the operations over numbers (see ***Numeral***).

Objects (system of —)

See ***System of objects***.

Object-oriented programming

The object-oriented programming (OOP) is such a way of programming that enables a modularity of programs due to the partitioning a memory into the extents containing the data and procedures. The extents can be used as the samples which by request can be copied.

Phrase (— grammatical)

The phrases are the *names*, *sentences* and *functors*.

Possible worlds

Possible worlds are considered as various set of individuals with the additional structure or without it.

Postulates

The inference rules and axioms are called by a term 'postulates'.

Prefix

The prefix is a functor (operator, "connective") that is written before the arguments.

Process (— effective)

Assume that there are definite transformations that can be actually performed over the certain elements. Assume that there also is a prescription defining the sequence of transformations that should be applied to some element one after other. It is said that the prescription defines an *effective process* to reach the certain aim relatively the element, if, when given this elements, the prescription singularly determines such a sequence of transformations, that the aim is reached in a finite number of steps.

Product

The product is a set of tuples (n-tuples). Depending on the number of elements in a tuple, the product is attached with the arity.

Projection

The projection is a subset of corresponding cartesian product.

Property

The property is a propositional function defined over a(n) (arbitrary) type A.

Proposition

A proposition is determined by such a way relatively which its proof can be considered.

Relational system

This is the system with a single basic predicate that is the binary relation.

Representation

The representation (of a system) is any way of considering the particular objects (of a domain) as the formal objects. The intensional (concrete) objects preserve a structure of the formal objects.

Sentence

The sentence expresses an assertion.

Set (— countable)

Any set which elements can be enumerated, i.e. arranged as a general list where some element is placed in the first position, some element is placed in the second position etc., so that any elements of this set early or late occurs in this list, is called *countable*.

Shell (Karoubi's —)

Karoubi's shell is a particular case of a category.

Suffix

Suffix is a functor that is written after the arguments.

System (— of objects)

In a *system of objects* the formal objects constitute a homogeneous inductive class (see ***Class inductive***). The elements of this inductive class are regarded as the *objects*, its generic objects are *atomic objects*, and the modes of combining are the *primitive generic operations*.

Thesis (Church's —)

It is impossible to prove the hypothesis that some standard formalization gives the satisfactory analogs of a non-formal notion of *algorithm* (see ***Algorithm***) and *algorithmic function* (see ***Function, computable by an algorithm***). Many of the

mathematicians accept the hypothesis of that the standard formalizations give a "reasonable rebuilding" of unavoidable inexact non-formal notions, and the hypothesis is called *Church's thesis*.

Theory

The theory is a way of choosing the subclass of truth propositions of the propositions belonging to the class of all the propositions $\mathcal{A}$.

Theory (— deductive)

The theory $\mathcal{T}$ is a deductive one, if $\mathcal{T}$ is an inductive class of the (elementary) propositions (see ***Class inductive***).

Theory (— of models)

The model theory is a branch of mathematical logic, studying the connections between the formal language and its interpretations, or *models*. The main studying objects are the sentences ϕ and algebraic systems $\mathcal{M}$ of the language L.

Theory (— consistent)

The consistent theory is defined as such a theory that does not contain a class $\mathcal{A}$ of all the propositions.

Theory (— complete)

The complete theory is such a deductive theory $\mathcal{T}$ that a joining to its axioms the elementary proposition which is not an elementary theorem with saving its rules unchangeable, makes it inconsistent (contradictory).

Theory (— complete by Post)

$\mathcal{T}$ is a complete theory, if every proposition of the class $\mathcal{A}$ of propositions is the consequence relatively $\mathcal{T}$ of any proposition X that is not in $\mathcal{T}$.

Theory (— of recursion)

The recursion theory studies the class of recursive, or effectively computable functions and its applications in mathematics. In a broad sense, the theory of recursion is considered as studying of the general processes of defining with a recursion not only over natural numbers, but over all the types of mathematical structures.

Theory (— of types)

In a basis of this theory is put a principle of the hierarchy. This means that the logical notions – propositions, individuals, propositional functions, – are arranged as the hierarchy of types. It is essential, that an arbitrary function, as its arguments, has only the notions that are its predecessors in this hierarchy.

Type doctrine

The type doctrine is due to B. Russell according to whom any type can be assumed as a range of truth of the propositional function. Moreover, it is assumed that every function has a type (its domain). In the type doctrine the *principle of replacing a type (of proposition) by definitionally equal type (proposition)* holds.

Unsolvability

The mathematical sense of some result concerning unsolvability means that some particular set is not recursive.

Value

The values constitute a conceptual class that contains the intensional objects which are assigned to the formal objects by the *evaluation*.

Variable

The variable is a "variable, or changeable object", that can be replaced (using a substitution).

Variable (— bound)

This is an object that participates in the operation having one or more formal parameters. Binding the variable have a sense relatively such an operation.

Variable (— indefinite)

This is an (atomic) object that (in ob-system) is not restricted.

Variable (— substitutive)

This is such an object in place of that the substitutions by the explicitly formulated substitution rule are allowed.

Practical work

Sources

Initially the practical work in applicative computations was included in a general practical work in the course "The systems of artificial intelligence" delivered in MEPhI (L.T. Kuzin, [25]). This practical work was prepared by the group of authors in the topics of deductive inference, relational algebra and relational calculus, conceptual knowledge representation and frames, Lisp programming (K.E. Aksenov, O.T. Balovnev, V.E. Wolfengagen, O.V. Voskresenskaya, A.V. Gannochka, M.Yu. Chuprikov, [1]; A.G. Panteleev, [32], M.A. Bulkin, Yu.R. Gabovich, A.G. Panteleev, [4]). Its essential part was the relational DBMS Lisp/R, that implemented one of the ways to develop the initial computational ideas. This method was based on the embedded computational systems and covered in (O.V. Voskresenskaya, [1]), which is included in the list of dissertations on p. 334.

Applicative computations

The variant of course delivered in MEPhI on the basis of this book (e.g., as "Special topics of discrete mathematics" or "Foundations of computer science") is equipped with the practical works that are in (V.E. Wolfengagen, L.V. Goltseva, [47]). Further development can be found in (L.V. Goltseva, [7]) and (A.V. Gavrilov, [6]), included in the list of dissertations on p. 334. The software for practical work is

implemented for IBM PC and is distributed in a machine form.

Structure of practical work

The practical work covers the main notions and notations used in applicative computational systems and allows to learn it completely: from the language and notational agreements to the metatheorems on the corresponding formal systems. It gives the basic knowledge on usage the applicative computational systems (ACS) as a foundation of the functional languages, functional programming systems and purely object languages.

The central notion in use is a term considered as the *representation* of an object. The term syntactically corresponds to the program in a functional language.

At this reason the *first* chapter of the guide to practical work deals with the technique of correct syntactical transformations – arranging the parentheses, – for the terms of different kinds.

The *second* chapter is intended to learn the reductions in ACS. An execution of the functional program results in the value obtained at the end of computations. In ACS the normal form of a term corresponds to the result of performing the reduction. According to the Church-Rosser theorem, the normal form is not depending on the order of the reduction steps that allows to build the various strategies of computations in the functional programming systems. A simulation of the λ-abstraction in combinatory logic can be considered as an example of interpretation the logical systems by means of combinatory logic.

The pair of combinators K and S constitute the basis for arbitrary λ-term, while the combinators I, B, C, S are the basis only for such terms that have no free variables. Using of these two bases for disassembling the terms from two chapters of the practical work gives a sufficient completeness of covering the topics because there are represented both the arbitrary λ-terms and combinatory terms

that are the λ-terms without free variables.

The *last* chapter of the practical work is the most creative because it covers the building of your own combinatory systems. It is shown how, joining the additional combinators to the basis combinators, increase the expressive power of the implemented environment.

As a whole, every of the chapters of practical works gives the learner a skill in the certain unit of questions and can be used as a separate laboratory work. In case the practical works are used as a computerized book then the chapters can be compiled and arranged in accordance with the learner's experience or such a way that is needed for an instructor.

Independent recourses

http://www.rbjones.com A purpose of Web-resource FACTASIA is "to develop a vision for our future and to provide recourses for building the vision and the future", and also "to contribute to the values which shape our future and to the technology which helps us build it". The partition "Logic" includes *combinatory logic* and λ-*calculus*, bibliography. See http://www.rbjones.com/rbjpub/logic/cl/

http://www.cwi.nl/~tromp/cl/cl.html The guide, bibliography and Java-applet are represented which allow to interpret the objects — expressions of an applicative language, — that are to be build by the rules of combinatory logic.

http://ling.ucsd.edu/~barker/Lambda/ski.html A simple on-line introduction to combinatory logic is represented.

http://tunes.org/~iepos/oldpage/lambda.html
Introduction to the lambda-calculus and combinatory logic.

http://foldoc.doc.ic.ac.uk This is FOLDOC – Free On-Line Dictionary of Computing.

http://dmoz.org/Science/Math/ Contains references and bibliography. The chapter '/Logic_and_Foundations' contains '/Computational_Logic', that has a reference to '/Combinatory_Logic_and_Lambda_Calculus/'.

http://www.cl.cam.ac.uk/Research/TSG
The directions of academic and research activity of the world known University of Cambridge Computer Laboratory.

http://web.comlab.ox.ac.uk/oucl The Oxford University Computing Laboratory (OUCL) which is the world known department on computer science. The URL http://web.comlab.ox.ac.uk/oucl/strachey contains the references on a termly series of Distinguished Lectures named after Christopher Strachey (1916-1975), the first Professor of Computation at Oxford University. He was the first leader of Programming Research Group (PRG), founded in 1965 – and was succeeded by Sir Tony Hoare in 1977, – and with Dana Scott he founded the field of *denotational semantics*, providing a firm mathematical foundation for programming languages.

Dissertations

[1] Voskresenskaya O.V., *Methods of development the relational database management system*, Candidate of Technical Sciences Thesis, 05.13.06 – Computer aided management systems, Moscow Ehgineering Physical Institute, Dissertatinal Council K.053.03.04 MEPhI, Moscow, 1985.

[2] Alexandrova I.A., *Development of information support and software for the systems of organizational type on the basis of conceptual models*, Candidate of Technical Sciences Thesis, 05.13.06 – Computer aided management systems, Moscow Ehgineering Physical Institute, Dissertatinal Council K.053.03.04 MEPhI, Moscow, 1986.

[3] Ismailova L.Yu., *Development of software for relational data processing in expert systems*, Candidate of Technical Sciences Thesis, 05.13.11 – Mathematical means and software for computing machines, complexes, systems and networks, Moscow Ehgineering Physical Institute, Dissertatinal Council K.053.03.04 MEPhI, Moscow, 1989.

[4] Volkov I.A., *Exploration and development of methods for analysis and trustworthy support of information about R&D in the area of medicine*, Candidate of Technical Sciences Thesis, 05.13.06 – Computer aided management systems; 14.00.33 – Social hygiene and public health, Moscow Ehgineering Physical Institute, Dissertatinal Council D-053.03.04 MEPhI, Moscow, 1990.

[5] Wolfengagen V.E., *Conceptual method of data bank design*, Doctor of Technical Sciences Thesis, 05.13.11 – Mathematical means

and software for computing machines, complexes, systems and networks, Moscow Ehgineering Physical Institute, Dissertatinal Council D-053.03.04 MEPhI, Moscow, 1990.

{ *Summary*. This Doctoral Thesis describes the conceptual method of data bank design which is determined as an approach to develop, apply and manage the databases and metadata bases. This approach covers the accommodations to the variable problem domain and its representation, e.g., by increasing/decreasing the level of details being described. The topics concerning the management of databases and metadata bases deal with an integration of data objects, metadata objects and programs. A unified computational environment preserves the extensibility of data object model. The representations of data and metadata are specified by the variety of data and metadata objects which are embedded into a computational framework.

The design procedure is based on a choice of multilevel conceptualization to capture the semantic features of data/metadata interconnections. The variety of data and metadata objects can be expanded without violation of the computational environment properties.

All the topics give a conceptual framework for thinking about computations with the objects. Several areas of theoretical computer science are covered, including type free and typed λ-calculus and combinatory logic with applications, evaluation of expressions, computations in a category. The topics, covered in this Thesis accumulated much experience in teaching these subjects in postgraduate computer science courses. }

[6] Gavrilov A.V., *Tunable programming system for the categorical computaions*, Candidate of Technical Sciences Thesis, 05.13.11 — Mathematical means and software for computing machines, complexes, systems and networks, Moscow Ehgineering Physical Institute, Dissertatinal Council D-053.03.04 MEPhI, Moscow, 1995.

[7] Goltseva L.V., *Applicative computational system with the intensional relations*, Candidate of Technical Sciences Thesis, 05.13.11 — Mathematical means and software for computing machines, complexes, systems and networks, Moscow Ehgineering Physical Institute, Dissertatinal Council D-053.03.04 MEPhI, Moscow, 1995.

[8] Zykov S.V., *Exploring and implementation of the integrated corporative information system for solving the tasks of personnel management*, Candidate of Technical Sciences Thesis, 05.13.11 — Mathematical means and software for computing machines, complexes and

computer networks, Moscow Ehgineering Physical Institute, Dissertatinal Council D-053.03.04 MEPhI, Moscow, 2000.

[9] Zabrodin A.L., *Exploring and implementation of the software for data management in computer aided systems of operative control of communication*, Candidate of Technical Sciences Thesis, 05.13.11 – Mathematical means and software for computing machines, complexes and computer networks, Moscow Ehgineering Physical Institute, Dissertatinal Council D-053.03.04 MEPhI, Moscow, 2000.

[10] Gorelov B.B. *Exploring and implementation of distributed information system for finacial data management*, Candidate of Technical Sciences Thesis, 05.13.11 – Mathematical means and software for computing machines, complexes and computer networks, Moscow Ehgineering Physical Institute, Dissertatinal Council D-212.130.03 MEPhI, Moscow, 2003.

About the Author

Viacheslav Wolfengagen received his Candidate of Technical Science degree in 1977 and the Doctor of Technical Science degree in 1990 from Moscow Engineering Physics Institute. He is a full professor of theoretical computer science and discrete mathematics at the Cybernetics Department of MEPhI and a professor of programming languages at the Cryptology and Discrete Mathematics Department of MEPhI. Since 1994 he has been with the Institute for Contemporary Education "JurInfoR-MSU" in Moscow where he is currently a head of the Department of Advanced Computer Studies and Information Technologies.

He chaired the 1999-2003 International Workshops in Computer Science and Information Technologies (CSIT). He is author of the books *Logic: Techniques of Reasoning* (2001, Center "JurInfoR"), *Constructions in Programming Languages: Methods of Description* (2001, Center "JurInfoR"), *Categorical Abstract Machine: Introduction to Computations* (2002, Center "JurInfoR"), and *Combinatory Logic in Programming: Computations with Objects through Examples and Exercises* (2003, MEPhI – Center "JurInfoR").

His research interests include data models, database design, software development databases, object and object-oriented programming and design, computation theory, programming languages, applicative computational systems. He was a manager of research and development projects *Logical Applicative Modeling Base of DAta* LAMBDA (version 3, project 93-01-00943 granted by RFBR), *Categorical Object-Oriented Abstract Machine* COOAM (project 96-01-01923 granted by RFBR), *Metadata Objects for Proxy-Based Computational Environment* (project 99-01-01229 granted by RFBR).

Viacheslav **Wolfengagen**

Combinatory logic in programming

Computations with objects through examples and exercises

Scientific editor: *L. Yu. Ismailova*
Typesetting: *Author*
Proof-reading: *L. M. Zinchenko*
Cover design: *O. V. Mortina*

Signed to publishing 04.01.2003. Offset print.
Offset paper. Format 6084/16. Pr. sh. 21,25. Cond. pr. sh. 19,8.
Copies 1500 Order N° .

"Center JurInfoR" Ltd.
103006, Moscow, Vorotnikovsky per., 7, phone (095) 956-25-12,
http://www.jurinfor.ru, e-mail: info@jurinfor.ru
License for publishing
ID N° 03088 issued by October 23, 2000

The production conforms the conditions of the
Public Health Department of the Russian Federation.
Sanitary-epidemiologic Certificate
N° 77.99.02.953.D.003230.06.01 issued by June 9, 2001

Tax incentive – All-Russia Classifier of Production
-005-93, volume 2; 953000 – books, booklets

Printed in a full accordance with
the quality of given transparencies
in the Typography "Nauka"
121099, Moscow, Shubinsky per., 6

ISBN 5-89158-101-9

9 785891 581012