Lambda Calculus with Types

This handbook with exercises reveals in formalisms, hitherto mainly used for hardware and software design and verification, unexpected mathematical beauty.

The lambda calculus forms a prototype universal programming language, which in its untyped version is related to Lisp, and was treated in the first author's classic The Lambda Calculus (1984). The formalism has since been extended with types and used in functional programming (Haskell, Clean) and proof assistants (Coq, Isabelle, HOL), used in designing and verifying IT products and mathematical proofs.

In this book, the authors focus on three classes of typing for lambda terms: simple types, recursive types and intersection types. It is in these three formalisms of terms and types that the unexpected mathematical beauty is revealed. The treatment is authoritative and comprehensive, complemented by an exhaustive bibliography, and numerous exercises are provided to deepen the readers' understanding and increase their confidence using types.

HENK BARENDREGT holds the chair on the Foundations of Mathematics and Computer Science at Radboud University, Nijmegen, The Netherlands.

WIL DEKKERS is an Associate Professor in the Institute of Information and Computing Sciences at Radboud University, Nijmegen, The Netherlands.

RICHARD STATMAN is a Professor of Mathematics at Carnegie Mellon University, Pittsburgh, USA.

PERSPECTIVES IN LOGIC

The *Perspectives in Logic* series publishes substantial, high-quality books whose central theme lies in any area or aspect of logic. Books that present new material not now available in book form are particularly welcome. The series ranges from introductory texts suitable for beginning graduate courses to specialized monographs at the frontiers of research. Each book offers an illuminating perspective for its intended audience.

The series has its origins in the old *Perspectives in Mathematical Logic* series edited by the Ω-Group for "Mathematische Logik" of the Heidelberger Akademie der Wissenschaften, whose beginnings date back to the 1960s. The Association for Symbolic Logic has assumed editorial responsibility for the series and changed its name to reflect its interest in books that span the full range of disciplines in which logic plays an important role.

PERSPECTIVES IN LOGIC

Lambda Calculus with Types

HENK BARENDREGT
Radboud University, Nijmegen

WIL DEKKERS
Radboud University Nijmegen

RICHARD STATMAN
Carnegie Mellon University

With contributions from
FABIO ALESSI, MARC BEZEM, FELICE CARDONE, MARIO COPPO,
MARIANGIOLA DEZANI-CIANCAGLINI, GILLES DOWEK, SILVIA GHILEZAN,
FURIO HONSELL, MICHAEL MOORTGAT, PAULA SEVERI, PAWEŁ URZYCZYN.

ASSOCIATION FOR SYMBOLIC LOGIC

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Mexico City

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521766142

Association for Symbolic Logic
Richard Shore, Publisher
Department of Mathematics, Cornell University, Ithaca, NY 14853
http://www.aslonline.org

First published 2013

Printed and bound in the United Kingdom by the CPI Group Ltd, Croydon CR0 4YY

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

ISBN 978-0-521-766-142 Hardback

Contents

Preface *page* ix
Contributors xii
Our Founders xiii
Introduction xv

PART I SIMPLE TYPES $\boldsymbol{\lambda}^{\mathbb{A}}_{\rightarrow}$ 1

1 The Simply Typed Lambda Calculus 5
1.1 The systems $\boldsymbol{\lambda}^{\mathbb{A}}_{\rightarrow}$ 5
1.2 First properties and comparisons 19
1.3 Normal inhabitants 32
1.4 Representing data types 38
1.5 Exercises 50

2 Properties 55
2.1 Normalization 55
2.2 Proofs of strong normalization 64
2.3 Checking and finding types 68
2.4 Checking inhabitation 77
2.5 Exercises 86

3 Tools 94
3.1 Semantics of $\boldsymbol{\lambda}_{\rightarrow}$ 94
3.2 Lambda theories and term models 107
3.3 Syntactic and semantic logical relations 114
3.4 Type reducibility 136
3.5 The five canonical term-models 156
3.6 Exercises 181

4 Definability, unification and matching
Written with the collaboration of Gilles Dowek 192
4.1 Undecidability of lambda-definability 192
4.2 Undecidability of unification 207
4.3 Decidability of matching of rank 3 213
4.4 Decidability of the maximal theory 230
4.5 Exercises 240

5 Extensions
Written in part with the collaboration of Marc Bezem 243
5.1 Lambda delta 243
5.2 Surjective pairing 255
5.3 Gödel's system $\mathcal{T}$: higher-order primitive recursion 282
5.4 Spector's system $\mathcal{B}$: bar recursion 303
5.5 Platek's system $\mathcal{Y}$: fixed point recursion 312
5.6 Exercises 315

6 Applications 323
6.1 Functional programming 323
6.2 Logic and proof-checking 341
6.3 Proof theory
Written with the collaboration of Silvia Ghilezan 351
6.4 Grammars, terms and types
Written with the collaboration of Michael Moortgat 363

PART II RECURSIVE TYPES $\lambda_{=}^{\mathcal{A}}$
Written with the collaboration of Felice Cardone and Mario Coppo 377

7 The Systems $\lambda_{=}^{\mathcal{A}}$ 379
7.1 Type algebras and type assignment 379
7.2 More on type algebras 390
7.3 Recursive types via simultaneous recursion 397
7.4 Recursive types via μ-abstraction 407
7.5 Recursive types as trees 424
7.6 Special views on trees 437
7.7 Exercises 442

8 Properties of Recursive Types 451
8.1 Simultaneous recursions vs μ-types 451
8.2 Properties of μ-types 455
8.3 Properties of types defined by a simultaneous recursion 475
8.4 Exercises 490

9 Properties of Terms with Types 494
9.1 First properties of $\boldsymbol{\lambda}^{\mathcal{A}}_{=}$ 494
9.2 Finding and inhabiting types 497
9.3 Strong normalization 507
9.4 Exercises 518

10 Models 520
10.1 Interpretations of type assignments in $\boldsymbol{\lambda}^{\mathcal{A}}_{=}$ 520
10.2 Interpreting Π_{μ} and Π^{*}_{μ} 525
10.3 Type interpretations in systems with explicit typing 541
10.4 Exercises 547

11 Applications 554
11.1 Subtyping 554
11.2 The principal type structures 567
11.3 Recursive types in programming languages 570
11.4 Further reading 573
11.5 Exercises 576

PART III INTERSECTION TYPES $\boldsymbol{\lambda}^{\mathcal{S}}_{\cap}$
Written with the collaboration of Fabio Alessi, Mariangiola Dezani-Ciancaglini, Furio Honsell and Paula Severi 577

12 An Example System 579
12.1 The type assignment system $\boldsymbol{\lambda}_{\cap}{}^{\mathrm{BCD}}$ 580
12.2 The filter model $\mathcal{F}^{\mathrm{BCD}}$ 586
12.3 Completeness of type assignment 589

13 Type Assignment Systems 591
13.1 Type theories 594
13.2 Type assignment 606
13.3 Type structures 610
13.4 Filters 614
13.5 Exercises 617

14 Basic Properties of Intersection Type Assignment 619
14.1 Inversion lemmas 622
14.2 Subject reduction and expansion 627
14.3 Exercises 634

15 Type and Lambda Structures 640
15.1 Meet semi-lattices and algebraic lattices 643

15.2 Natural type structures and lambda structures 656
15.3 Type and zip structures 662
15.4 Zip and lambda structures 667
15.5 Exercises 676

16 Filter Models 680
16.1 Lambda models 683
16.2 Filter models 689
16.3 $\mathcal{D}_\infty$ models as filter models 701
16.4 Other filter models 716
16.5 Exercises 724

17 Advanced Properties and Applications 728
17.1 Realizability interpretation of types 730
17.2 Characterizing syntactic properties 735
17.3 Approximation theorems 742
17.4 Applications of the approximation theorem 757
17.5 Undecidability of inhabitation
Written with the collaboration of Paweł Urzyczyn 762
17.6 Exercises 786

References 791

Indices 814
Index of terms 815
Index of citations 823
Index of symbols 828

Preface

This book is about lambda terms typed using *simple*, *recursive* and *intersection* types. In some sense it is a sequel to Barendregt (1984). That book is about untyped lambda calculus. Types give the untyped terms more structure: function applications are allowed only in some cases. In this way one can single out untyped terms having special properties. But there is more to it. The extra structure makes the theory of typed terms quite different from the untyped ones.

The emphasis of the book is on syntax. Models are introduced only insofar as they give useful information about terms and types or if the theory can be applied to them.

The writing of this book has been different from the one on untyped lambda calculus. First of all, since many researchers are working on typed lambda calculus, we were aiming at a moving target. Moreover there has been a wealth of material to work with. For these reasons the book was written by several authors. Several long-term open problems have been solved during the period the book was written, notably the undecidability of lambda definability in finite models, the undecidability of second-order typability, the decidability of the unique maximal theory extending $\boldsymbol{\beta\eta}$-conversion and the fact that the collection of closed terms of not every simple type is finitely generated, and the decidability of matching at arbitrary types of order higher than 4. The book has not been written as an encyclopedic monograph: many topics are only partially treated; for example, reducibility among types is analyzed only for simple types built up from only one atom.

One of the recurring distinctions made in the book is the difference between the implicit typing due to Curry versus the explicit typing due to Church. In the latter case the terms are an enhanced version of the untyped terms, whereas in the Curry theory to some of the untyped terms a collection

of types is being assigned. The book is mainly about Curry typing, although some chapters treat the equivalent Church variant.

The applications of the theory are within the theory itself, or in the theory of programming languages, or in proof theory, including the technology of fully formalized proofs used for mechanical verification, or in linguistics. Often the applications are given in an exercise with hints.

We hope that the book will attract readers and inspire them to pursue the topic.

Acknowledgments

Many thanks are due to many people and institutions. The first author obtained substantial support in the form of a generous personal research grant by the Board of Directors of Radboud University, the Spinoza Prize by The Netherlands Organisation for Scientific Research (NWO), and the Distinguished Lorentz Fellowship by the Lorentz Institute at Leiden University and The Netherlands Institue of Advanced Studies (NIAS) at Wassenaar. Not all of these means were used to produce this book, but they have been important. The Mathematical Forschungsinstitut at Oberwolfach, Germany, provided generous hospitality through their 'Research in Pairs' program. The Residential Centre at Bertinoro of the University of Bologna hosted us in their stunning castle. The principal regular sites where the work was done are the Institute for Computing and Information Sciences of Radboud University at Nijmegen, The Netherlands, the Department of Mathematics of Carnegie–Mellon University at Pittsburgh, USA, the Departments of Informatics at the Universities of Torino and Udine, both Italy.

The three main authors wrote the larger part of Part I and thoroughly edited Part II, drafted by Mario Coppo and Felice Cardone, and Part III, drafted by Mariangiola Dezani-Ciancaglini, Fabio Alessi, Furio Honsell, and Paula Severi. Various chapters and sections were drafted by other authors as follows: Chapter 4 by Gilles Dowek, Sections 5.3–5.5 by Marc Bezem, Section 6.4 by Michael Moortgat, and Section 17.5 by Pawel Urzyczyn, while Section 6.3 was co-authored by Silvia Ghilezan. This 'thorough editing' consisted of rewriting the material to bring it all into one style, but in many cases also in adding results and making corrections. It was agreed upon beforehand with all co-authors that this would happen.

Since 1974 Jan Willem Klop has been a close colleague and friend and

we have been engaged with him in many inspiring discussions on λ-calculus and types.

Several people helped during the later phases of writing the book. The reviewer Roger Hindley gave invaluable advice. Vincent Padovani carefully read Section 4.3. Other help came from Jörg Endrullis, Clemens Grabmeyer, Tanmay Inamdar, Thierry Joly, Jan Willem Klop, Pieter Koopman, Dexter Kozen, Giulio Manzonetto, James McKinna, Vincent van Oostrom, Andrew Polonsky, Rinus Plasmeijer, Arnoud van Rooij, Jan Rutten, Sylvain Salvati, Christian Urban, Bas Westerbaan, and Bram Westerbaan.

Use has been made of the following macro packages: 'prooftree' of Paul Taylor, 'xypic' of Kristoffer Rose, 'robustindex' of Wilbert van der Kallen, and several lay-out commands of Erik Barendsen.

At the end producing this book turned out a time-consuming enterprise. But that seems to be the way: while the production of the content of Barendregt (1984) was expected to take two months, it took fifty; for this book our initial estimate was four years, while it turned out to be twenty.

Our partners were usually patiently understanding when we spent yet another period of writing and rewriting. We cordially thank them for their continuous and continuing support and love.

Nijmegen and Pittsburgh April 23, 2013

Henk Barendregt[1,2]
Wil Dekkers[1]
Rick Statman[2]

[1] Faculty of Science
Radboud University, Nijmegen, The Netherlands
[2] Departments of Mathematics and Computer Science
Carnegie Mellon University, Pittsburgh, USA

Contributors

Fabio Alessi, *Department of Mathematics and Computer Science, Udine University*
Marc Bezem, *Department of Informatics, Bergen University*
Felice Cardone, *Department of Informatics, Torino University*
Mario Coppo, *Department of Informatics, Torino University*
Mariangiola Dezani-Ciancaglini, *Department of Informatics, Torino University*
Gilles Dowek, *Department of Informatics, École Polytechnique;* and *INRIA*
Silvia Ghilezan, *Center for Mathematics & Statistics, University of Novi Sad*
Furio Honsell, *Department of Mathematics and Computer Science, Udine University*
Michael Moortgat, *Department of Modern Languages, Utrecht University*
Paula Severi, *Department of Computer Science, University of Leicester*
Paweł Urzyczyn, *Institute of Informatics, Warsaw University*

Our Founders

The founders of the topic of this book are Alonzo Church (1903–1995), who invented the lambda calculus (Church (1932), Church (1933)), and Haskell Curry (1900–1982), who invented 'notions of functionality' (Curry (1934)) that later got transformed into types for the hitherto untyped lambda terms. As a tribute to Church and Curry below are shown pictures of them at an early stage of their careers. Church and Curry were honored jointly for their timeless invention by the Association for Computing Machinery in 1982.

Alonzo Church (1903–1995), Studying mathematics at Princeton University (1922 or 1924). Courtesy of Alonzo Church and Mrs. Addison-Church.

Haskell B. Curry (1900–1982), BA in mathematics at Harvard (1920). Courtesy of Town & Gown, Penn State.

Introduction

The rise of lambda calculus

Lambda calculus is a formalism introduced by Church in 1932 that was intended to be used as a foundation for mathematics, including its computational aspects. Supported by his students Kleene and Rosser – who showed that the prototype system was inconsistent – Church distilled a consistent computational part and ventured in 1936 the Thesis that exactly the intuitively computable functions could be captured by it. He also presented a function that could not be captured by the λ-calculus. In that same year Turing introduced another formalism, describing what are now called Turing Machines, and formulated the related Thesis that exactly the mechanically computable functions are able to be captured by these machines. Turing also showed in the same paper that the question of whether a given statement could be proved (from a given set of axioms) using the rules of any reasonable system of logic is not computable in this mechanical way. Finally Turing showed that the formalism of λ-calculus and Turing Machines define the same class of functions.

Together Church's Thesis, concerning computability by *homo sapiens*, and Turing's Thesis, concerning computability by mechanical devices, using formalisms that are equally powerful and that have their computational limitations, made a deep impact on the 20th century philosophy of the power and limitations of the human mind. So far, cognitive neuropsychology has not been able to refute the combined Church–Turing Thesis. On the contrary, that discipline also shows the limitation of human capacities. On the other hand, the analyses of Church and Turing indicate an element of reflection (universality) in both Lambda Calculus and Turing Machines, that according to their combined thesis is also present in humans.

Turing Machine computations are relatively easy to implement on elec-

tronic devices, as started to happen early in the 1940s. The above-mentioned universality was employed by von Neumann[1] enabling the construction not only of ad hoc computers but even a universal one, capable of performing different tasks depending on a program. This resulted in what is now called *imperative programming*, with the C language presently the most widely used for programming in this paradigm. As with Turing Machines a computation consists of repeated modifications of some data stored in memory. The essential difference between a modern computer and a Turing Machine is that the former has random access memory[2].

Functional Programming

The computational model of Lambda Calculus, on the other hand, has given rise to *functional programming*. The input M becomes part of an expression FM to be evaluated, where F represents the intended function to be computed on M. This expression is reduced (rewritten) according to some rules (indicating the possible computation steps) and some strategy (indicating precisely which steps should be taken).

To show the elegance of functional programming, here is a short functional program generating primes using Eratosthenes sieve (Miranda program by D. Turner):

```
primes = sieve [2..]
         where
         sieve (p:x) = p : sieve [n | n<-x ; n mod p > 0]
primes_upto n = [p | p<- primes ; p<n]
```

while a similar program expressed in an imperative language looks like (Java program from <rosettacode.org>)

```
public class Sieve{
  public static LinkedList<Integer> sieve(int n){
    LinkedList<Integer> primes = new LinkedList<Integer>();
    BitSet nonPrimes = new BitSet(n+1);

    for (int p = 2; p <= n; p = nonPrimes.nextClearBit(p+1)){
        for (int i = p * p; i <= n; i += p)
            nonPrimes.set(i);
```

[1] It was von Neumann who visited Cambridge UK in 1935 and invited Turing to Princeton during 1936–1937, so he probably knew Turing's work.

[2] Also, the memory on a TM is infinite: Turing wanted to be technology-independent, but was restricting a computation with a given input to one using finite memory and time.

```
                primes.add(p);
        }
        return primes;
    }
}
```

Of course the algorithm is extremely simple, one of the first ever invented. However, the gain for more complex algorithms remains, as functional programs do scale up.

The power of functional programming languages derives from several facts.

(1) All expressions of a functional programming language have a constant meaning (i.e. independent of a hidden state). This is called 'referential transparency' and makes it easier to reason about functional programs and to make versions for parallel computing, important for quality and efficiency.
(2) Functions may be arguments of other functions, usually called 'functionals' in mathematics and higher-order functions in programming. There are functions acting on functionals, etc; in this way one obtains functions of arbitrary order. Both in mathematics and in programming, higher-order functions are natural and powerful phenomena. In functional programming this enables the flexible composition of algorithms.
(3) Algorithms can be expressed in a clear goal-directed mathematical way, using various forms of recursion and flexible data structures. The bookkeeping needed for the storage of these values is handled by the language compiler instead of the user of the functional language[3].

Types

The formalism as defined by Church is untyped. The early functional languages, of which Lisp (McCarthy et al. (1962)) and Scheme (Abelson et al. (1991)) are best known, are also untyped: arbitrary expressions may be applied to each other. Types first appeared in *Principia Mathematica*, Whitehead and Russell (1910-1913). In Curry (1934) types are introduced and assigned to expressions in 'combinatory logic', a formalism closely related to lambda calculus. In Curry and Feys (1958) this type assignment mechanism was adapted to λ-terms, while in Church (1940) λ-terms were ornamented

[3] In modern functional languages there is a palette of techniques (such as overloading, type classes and generic programming) to make algorithms less dependent of specific data types and hence more reusable. If desired the user of the functional language can help the compiler to achieve a better allocation of values.

by fixed types. This resulted in the closely related systems $\boldsymbol{\lambda}^{\mathrm{Cu}}_{\rightarrow}$ and $\boldsymbol{\lambda}^{\mathrm{Ch}}_{\rightarrow}$ treated in Part I.

Types are being used in many, if not most, programming languages. These are of the form

$$\mathtt{bool}, \mathtt{nat}, \mathtt{real}, \ldots$$

and occur in compounds like

$$\mathtt{nat} \to \mathtt{bool}, \mathtt{array}(\mathtt{real}), \ldots$$

Using the formalism of types in programming, many errors can be prevented if terms are required to be typable: arguments and functions should match. For example M of type A can be an argument only of a function of type $A \to B$. Types act in a way similar to the use of dimensional analysis in physics. Physical constants and data obtain a 'dimension'. Pressure p, for example, has a dimension expressed as

$$M/L^2$$

giving the constant R in Boyle's law,

$$\frac{pV}{T} = R,$$

that has a dimension which prevents one from writing an equation like $E = TR^2$. By contrast Einstein's famous equation

$$E = mc^2$$

is already meaningful from the viewpoint of dimensional analysis.

In most programming languages the formation of function space types is usually not allowed to be iterated as in

$(\mathtt{real} \to \mathtt{real}) \to (\mathtt{real} \to \mathtt{real})$ for indefinite integrals $\int f(x)dx$;

$(\mathtt{real} \to \mathtt{real}) \times \mathtt{real} \times \mathtt{real} \to \mathtt{real}$ for definite integrals $\int_a^b f(x)dx$;

$([0,1] \to \mathtt{real}) \to (([0,1] \to \mathtt{real}) \to \mathtt{real}) \to (([0,1] \to \mathtt{real}) \to \mathtt{real})$,

where the latter is the type of a map occuring in functional analysis, see Lax (2002). Here we have written "$[0,1] \to \mathtt{real}$" for what should be more accurately the set $C[0,1]$ of continuous functions on $[0,1]$.

Because there is the Hindley–Milner algorithm (see Theorem 2.3.14 in Chapter 2) that decides whether an untyped term does have a type and computes the most general type, types have found their way to functional programming languages. The first such language to incorporate the types of the simply typed λ-calculus is ML (Milner et al. (1997)). An important

aspect of typed expressions is that if a term M is correctly typed by type A, then also during the computation of M the type remains the same (see Theorem 1.2.6, the 'subject reduction theorem'). This is expressed as a feature in functional programming: one only needs to check types during compile time.

In functional programming languages, however, types come of age and are allowed in their full potential by giving a precise notation for the type of data, functions, functionals, higher-order functionals, ... up to arbitrary degree of complexity. Interestingly, the use of higher-order types given in the mathematical examples is modest compared to higher-order types occurring in a natural way in programming situations:

$$\begin{array}{l}[(a \to ([([b],c)] \to [([b],c)]) \to [([b],c)] \to [b] \to [([b],c)]) \to \\ ([([b],c)] \to [([b],c)]) \to [([b],c)] \to [b] \to [([b],c)]] \to \\ a \to (d \to ([([b],c)] \to [([b],c)]) \to [([b],c)] \to [b] \to [([b],c)]) \\ \to ([([b],c)] \to [([b],c)]) \to [([b],c)] \to [b] \to [([b],c)]] \to \\ d \to ([([b],c)] \to [([b],c)]) \to [([b],c)] \to [b] \to [([b],c)]] \to \\ ([([b],c)] \to [([b],c)]) \to [([b],c)] \to [b] \to [([b],c)].\end{array}$$

This type (it does not actually occur in this form in the program, but is notated using memorable names for the concepts being used) is used in a functional program for efficient parser generators, see Koopman and Plasmeijer (1999). The type $[a]$ denotes that of lists of type a and (a, b) denotes the 'product' $a \times b$. Product types can be simulated by simple types, while for list types one can use the recursive types developed in Part 2 of this book. Although in the pure typed λ-calculus only a rather restricted class of terms and types is represented, relatively simple extensions of this formalism have universal computational power. Since the 1970s the following programming languages have appeared: ML (not yet purely functional); Miranda (Thompson (1995), `<www.cs.kent.ac.uk/people/staff/dat/miranda/>`) the first purely functional typed programming language, well-designed, but slowly interpreted; Clean (van Eekelen and Plasmeijer (1993), Plasmeijer and van Eekelen (2002), `<wiki.clean.cs.ru.nl/Clean>`); and Haskell (Hutton (2007), Peyton Jones (2003), `<www.haskell.org>`). Both Clean and Haskell are state of the art pure functional languages with fast compiler generating fast code). They show that functional programming based on λ-calculus can be efficient and apt for industrial software. Functional programming languages are also being used for the design (Sheeran (2005)) and testing (Koopman and Plasmeijer (2006)) of hardware. In each case it is the compact mathematical expressivity of the functional languages that makes them fit for the description of complex functionality.

Semantics of natural languages

Typed λ-calculus has also been employed in the semantics of natural languages (Montague (1973), van Benthem (1995)). An early indication of this possibility can already be found in Curry and Feys (1958), Section 8S2.

Certifying proofs

In addition to its use in design, the λ-calculus has also been used for verification, not just for the correctness of IT products, but also of mathematical proofs. The underlying idea is the following. Ever since Aristotle's formulation of the axiomatic method and Frege's formulation of predicate logic, one could write down mathematical proofs in full detail. Frege, who captured reasoning by his introduction of the predicate logic, started to formalize mathematics, but unfortunately began from an axiom system that turned out to be inconsistent, as shown by the Russell paradox. In *Principia Mathematica* Whitehead and Russell used types to prevent the paradox. They had the same formalization goal in mind and developed some elementary arithmetic. Based on their work, Gödel was able to state and prove his fundamental incompleteness result. In spite of the intention behind *Principia Mathematica*, proofs in the underlying formal system were not fully formalized. Substitution was left as an informal operation and in fact the way *Principia Mathematica* treated free and bound variables was implicit and incomplete. Here begins the role of the λ-calculus. As a formal system dealing with manipulating formulas, distinguishing carefully between free and bound variables and their interaction, it was the missing link towards a full formalization. Now, if an axiomatic mathematical theory is fully formalized, a computer can verify the correctness of the definitions and proofs. The reliability of computer-verified theories relies on the fact that logic has only about a dozen rules and their implementation poses relatively few problems. This idea was pioneered in the late 1960s by N. G. de Bruijn in the proof-checking language and system Automath (Nederpelt et al. (1994), `<www.win.tue.nl/automath>`).

The methodology has given rise to proof-assistants. These are computer programs that help the human user to develop mathematical theories. The initiative comes from the human who formulates notions, axioms, definitions, proofs and computational tasks. The computer verifies the well-definedness of the notions, the correctness of the proofs, and performs the computational tasks. In this way arbitrary mathematical notions can represented and ma-

nipulated on a computer. Many of the mathematical assistants are based on extensions of typed λ-calculus. See Section 6.2 for more information.

What this book is and is not about

None of the fascinating applications, mentioned above, of lambda calculus with types are treated in this book. We will study the formalism for its mathematical beauty. In particular this monograph focuses on mathematical properties of three classes of typing for lambda terms.

Simple types, constructed freely from type atoms, cause strong normalization, subject reduction, decidability of typability and inhabitation, undecidability of lambda-definability. There turn out to be five canonical term models based on closed terms. Powerful extension with respectively a discriminator, surjective pairing, operators for primitive recursion, bar recursion, and a fixed point operator are being studied. Some of these extensions remain constructive, others are utterly non-constructive, and some will be at the boundary of these two methods.

Recursive types allow functions to fit as input for themselves, losing strong normalization (restored by allowing only positive recursive types). Typability remains decidable. Unexpectedly, α-conversion, dealing with the hygienic treatment of free and bound variables among recursive types, has interesting mathematical properties.

Intersection types allow functions to take arguments of different types simultaneously. Under certain mild conditions this leads to subject conversion, turning the filters of types of a given term into a lambda model. Classical lattice models can be described as intersection type theories. Typability and inhabitation now becomes undecidable, the latter being equivalent to undecidability of lambda-definability for models of simple types.

A flavor of some of the applications of typed lambda calculus is given: functional programming (Section 6.1), proof-checking (Section 6.2), and formal semantics of natural languages (Section 6.4).

What this book could have been about

This book could have been also about dependent types, higher-order types and inductive types, all used in some of the mathematical assistants. Originally we had planned a second volume to do so. But given the effort needed to write this one, we will probably not do so. Higher-order types are treated in a mathematically oriented style in Girard et al. (1989), and Sørensen and Urzyczyn (2006). Research monographs on dependent and inductive types

are lacking. This is an invitation to the community of the next generation of researchers!

Some notational conventions

A *partial function* from a set X to a set Y is a collection of ordered pairs $f \subseteq X \times Y$ such that $\forall x \in X, y, y' \in Y.[\langle x, y\rangle \in f \ \& \ \langle x, y'\rangle \in f \ \Rightarrow \ y = y']$.

The set of partial functions from a set X to a set Y is denoted by $X \mathrel{\text{↛}} Y$. If $f \in (X \mathrel{\text{↛}} Y)$ and $x \in X$, then $f(x)$ is *defined*, notation $f(x)\!\downarrow$ or $x \in \mathrm{dom}(f)$, if for some y one has $\langle x, y\rangle \in f$. In that case one writes $f(x) = y$. On the other hand $f(x)$ is *undefined*, notation $f(x)\!\uparrow$, means that for no $y \in Y$ one has $\langle x, y\rangle \in f$. An expression E in which partial functions are involved, may be defined or not. If two such expressions are compared, then, following Kleene (1952), we write $E_1 \simeq E_2$ for

$$\text{if } E_1\!\downarrow, \text{ then } E_2\!\downarrow \text{ and } E_1 = E_2, \text{ and vice versa.}$$

The set of natural numbers is denoted by ω. The notation $\triangleq$ is used for "equality by definition". Similarly '$\overset{\triangle}{\Longleftrightarrow}$'. is used for the definition of a concept. By contrast ::= stands for the more specific introduction of a syntactic category defined by the Backus–Naur form. The notation $\equiv$ stands for syntactic equality (for example to remind the reader that the left hand side was defined previously as the right hand side In a definition we do not write 'M is *closed* iff $\mathrm{FV}(M) = \emptyset$' but '$M$ is *closed* if $\mathrm{FV}(M) = \emptyset$'. The end of a proof is indicated by '■'.