INTERNATIONAL SERIES OF MONOGRAPHS ON COMPUTER SCIENCE • 11

Computation and Reasoning

A Type Theory for Computer Science

ZHAOHUI LUO

OXFORD SCIENCE PUBLICATIONS

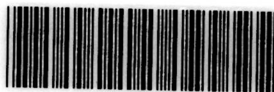
TP301
L964

9463287

Computation and Reasoning

A Type Theory for Computer Science

ZHAOHUI LUO
University of Edinburgh


E9463287

CLARENDON PRESS • OXFORD
1994

Oxford University Press, Walton Street, Oxford OX2 6DP
Oxford New York Toronto
Delhi Bombay Calcutta Madras Karachi
Kuala Lumpur Singapore Hong Kong Tokyo
Nairobi Dar es Salaam Cape Town
Melbourne Auckland Madrid
and associated companies in
Berlin Ibadan

Oxford is a trade mark of Oxford University Press

Published in the United States
by Oxford University Press Inc., New York

© Zhaohui Luo, 1994

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission in writing of Oxford University Press. Within the UK, exceptions are allowed in respect of any fair dealing for the purpose of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act, 1988, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms and in other countries should be sent to the Rights Department, Oxford University Press, at the address above.

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's prior consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser.

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

Library of Congress Cataloging in Publication Data

ISBN 0 19 853835 9

Typeset by the Author in LaTeX
Printed in Great Britain by
Bookcraft (Bath) Ltd.
Midsomer Norton, Avon

INTERNATIONAL SERIES OF
MONOGRAPHS ON COMPUTER SCIENCE

General Editors

DOV M. GABBAY   JOHN E. HOPCROFT
GORDON D. PLOTKIN   JACOB T. SCHWARTZ
DANA S. SCOTT   JEAN VUILLEMIN
ZVI GALIL

THE INTERNATIONAL SERIES OF MONOGRAPHS ON COMPUTER SCIENCE

1. *The design and analysis of coalesced hashing* Jeffrey S. Vitter and Wen-chin Chen
2. *Initial computability, algebraic specifications, and partial algebras* Horst Reichel
3. *Art gallery theorems and algorithms* Joseph O'Rourke
4. *The combinatorics of network reliability* Charles J. Colbourn
5. *Discrete relaxation techniques* T. Henderson
6. *Computable set theory* D. Cantone, A. Ferro, and E. Omodeo
7. *Programming in Martin-Löf's type theory: an introduction* Bengt Nordström, Kent Petersson, and Jan M. Smith
8. *Nonlinear optimization: complexity issues* Stephen A. Vavasis
9. *Derivation and validation of software metrics* Martin Shepperd and Darrel Ince
10. *Automated deduction in multiple-valued logics* Reiner Hähnle
11. *Computation and reasoning: a type theory for computer science* Zhaohui Luo

此为试读，需要完整PDF请访问：www.ertongbook.com

To Dejuan

Preface

This monograph studies a type theory and its applications to computer science. When its publication was considered three years ago, it was meant to be a simple and straightforward modification of my PhD thesis on the Extended Calculus of Constructions. However, it turns out to be a much more extensive task than I expected. Reviewing many of the ideas dealt with in the original thesis, I have found them calling for more extensive treatment and explanations, which have led me to more research in order to obtain a better understanding. The result is a further development of the type theory; many chapters are rewritten and several new chapters are added. It is my hope that the basic ideas in the development of the type theory are explained more clearly and its potential in applications is better demonstrated.

I would like to thank many people who have helped me in the course of this work. First of all, I would like to express my gratitude to Rod Burstall, my PhD supervisor, who has provided not only many interesting ideas and helpful advice, but encouragement at the right time and friendship along the way. I am greatly indebted to Thierry Coquand, Susumu Hayashi, Eugenio Moggi, and Randy Pollack who have either directly or indirectly contributed their ideas and insights in various ways to the development of the type theory considered in this monograph. Thanks also go to, among others, Peter Aczel, Henk Barendregt, Stefano Berardi, Matt Fairtlough, Healfdene Goguen, Per Martin-Löf, Gordon Plotkin, Don Sannella, Thomas Streicher, Paul Taylor, and the members in the Lego club in Edinburgh for the helpful discussions we have had, and to my former teachers in China who have taught me the basic knowledge in computer science and logic.

Edinburgh Z.L.
December 1993

Contents

1 Introduction 1
1.1 Motivations in computer science 2
1.2 Basic concepts in type theory 3
1.2.1 Objects and types 3
1.2.2 Propositions as types 5
1.2.3 Meaning and use 6
1.3 The conceptual universe of types 8
1.3.1 Paradoxical type structures 9
1.3.2 Martin-Löf's type theory 10
1.3.3 Impredicative type theories 11
1.3.4 Data types vs. logical propositions 12
1.4 Towards a unifying theory of dependent types 15

2 The Extended Calculus of Constructions 21
2.1 The language of ECC 21
2.1.1 Terms and computation 21
2.1.2 Judgements and inference rules 24
2.2 Informal explanations 26
2.2.1 Computation and computational equality 26
2.2.2 Judgements and context validity 28
2.2.3 Types and their meaning explanation 29
2.2.4 Non-propositional types 29
2.2.5 Predicative universes and the reflection principle 32
2.2.6 Propositions and the impredicative universe 34
2.2.7 Type equality and the cumulativity relation 36
2.3 Further remarks and discussion 38
2.3.1 Data types vs. logical propositions 38
2.3.2 Σ-types and existential types 41
2.3.3 Equalities: intensionality vs. extensionality 43
2.3.4 Decidability issues 46
2.3.5 On the use of the type theory 46

3 Basic meta-theoretic properties 49
3.1 Church-Rosser theorem and cumulativity 49
3.2 Derivable judgements and derivability 54
3.3 Principal types 61

4 Strong normalisation 65
- 4.1 The Girard–Tait reducibility method 66
  - 4.1.1 The reducibility method and the notion of predicativity 66
  - 4.1.2 Environments 69
  - 4.1.3 Saturated sets and candidates of reducibility 71
- 4.2 Quasi-normalisation 74
  - 4.2.1 Levels of types 74
  - 4.2.2 Quasi-normalisation and degrees of types 76
  - 4.2.3 The complexity measure of types 79
  - 4.2.4 An inductive proof of the quasi-normalisation theorem 81
- 4.3 Strong normalisation 88
  - 4.3.1 Possible denotations of objects 88
  - 4.3.2 Assignments and valuations 90
  - 4.3.3 The interpretation 92
  - 4.3.4 Soundness of the interpretation 96
  - 4.3.5 The strong normalisation theorem 100

5 The internal logic and decidability 103
- 5.1 The internal higher-order logic 103
  - 5.1.1 The internal logic and its consistency 103
  - 5.1.2 Understanding the logical operators 106
  - 5.1.3 The Leibniz equality and equality reflection 109
- 5.2 Decidability of the type theory 112
  - 5.2.1 Decidability of conversion and cumulativity 112
  - 5.2.2 Decidability of type inference and type checking 112

6 A set-theoretic model 117
- 6.1 Understanding ECC in the ω-**Set** framework 118
- 6.2 Valid contexts and objects 120
- 6.3 Predicative universes and non-propositional types 121
- 6.4 The impredicative universe and propositions 123
- 6.5 Remarks 125

7 Computational and logical theories 127
- 7.1 Computational theories and inductive data types 128
  - 7.1.1 The type of natural numbers 128
  - 7.1.2 The type of lists of natural numbers 131
- 7.2 Abstract theories and abstract reasoning 132
  - 7.2.1 A notion of abstract theory 133
  - 7.2.2 Abstract reasoning 136
  - 7.2.3 Theory morphisms and proof inheritance 136
- 7.3 Discussion 137

8 Specification and development of programs 139
- 8.1 A brief summary 140

8.2 Specifications and data refinement 141
8.2.1 Program specifications and their realisations 141
8.2.2 Specifications of abstract data types 144
8.2.3 Data refinement and implementation 146
8.3 Modular design and structured specification 149
8.3.1 Decomposition and sharing 150
8.3.2 Constructors and selectors 153
8.3.3 Constructor/selector implementation 156
8.4 Parameterised specification 156
8.4.1 Parameterised specifications 157
8.4.2 Implementation of parameterised specifications 159
8.5 Discussion 162

9 Towards a unifying theory of dependent types 165
9.1 A logical framework with inductive schemata 166
9.1.1 Martin-Löf's logical framework 166
9.1.2 Specifying type theories in LF 170
9.1.3 Inductive schemata 173
9.2 The formulation of UTT 175
9.2.1 SOL: the internal logical mechanism 175
9.2.2 Inductive data types 177
9.2.3 Predicative universes 182
9.2.4 A summary 185
9.3 Discussion 186
9.3.1 The internal logic and pure logical truths 186
9.3.2 Further separation of propositions and data types 195
9.3.3 Intensionality and η-equality rules 198
9.3.4 Understanding of the type theory 202
9.3.5 Inductive families of types 204
9.3.6 Subtyping and other implementation issues 208
9.4 Final remarks 209

Bibliography 211

Notation and Symbols 221

Index 225

1
Introduction

The study of type theory may offer an *adequate* computational and logical language for computer science. There are several compelling reasons supporting such a claim of adequacy. First, type theory offers a coherent treatment of two related but different fundamental notions in computer science: computation and logical inference. This makes it possible for one to program and to understand and reason about programs in a single formalism. Second, type theory can provide nice abstraction mechanisms which support conceptually clear (e.g. modular and structured) development of programs, specifications, and proofs. This makes it a promising candidate for a uniform language for programming, specification, and reasoning in the large as well as in the small. Finally, although type theory provides powerful and sophisticated tools, it is simple. Its simplicity is to be understood in two aspects: one is that it allows a direct operational understanding of the meanings of the constructions in its language, which gives a solid basis of its use in applications; the other is that it is manageable in the sense that there is a good way to implement it on the computer.

In this monograph, we develop a theory of dependent types, study its properties, and illustrate its uses in computer science. Besides its contribution to the studies of type theory, logic and computer languages in general, it is hoped that such an investigation will help to explain the theme and points summarised in the last paragraph, make explicit the advantages and possible limitations of using a type-theoretic language in pragmatic applications, and, therefore, stimulate further development in this area.

In this introduction, after a brief explanation of some viewpoints and motivations of studying type theory for computer science, the basic concepts in type theory and their relationship with logical inference and programming are explained intuitively, and a general inquiry into the underlying structures of various type theories is made to discuss several issues in the study of type theory, leading to an explication of the ideas based on which our type theory is developed. The type structure of the type theory is briefly described, followed by an overview of this monograph.

此为试读，需要完整PDF请访问：www.ertongbook.com

1.1 Motivations in computer science

Type theories have mainly been developed as foundational and logical languages by logicians who are interested in the foundation of mathematics. Why is type theory useful for computer science? Here, rather than discussing the possible applications of type theories in computer science, we briefly explain our particular motivations for studying and developing type theory as a promising computational and logical language.

The study of languages in computer science, most of which are necessarily formal in order to be implementable on the computer, is closely related to that of formal systems, including type theories, which may provide valuable insights and useful ideas in understanding and designing computer languages. In this respect, the study of logical systems is particularly important since logic plays an indispensable role in specifying and reasoning about programs. Besides such a general contribution, type theory has several features that are particularly interesting in computer science.

Computer languages have developed with their own distinguishing characteristics, and computer scientists have rather different concerns as compared with those in the traditional study of logical systems. For example, the notion of *computation* or computational behaviour of programs is the most basic and the most important in understanding a programming language. That is why the operational semantics which directly characterises the computational behaviour of programs has been essential for the programmer and prevailing in understanding and using a programming language, although other semantics such as denotational semantics may also help a great deal. Unlike the other logical or mathematical languages, type theory is essentially a computational language (in particular, a functional programming language) where computation is taken as the basic notion and its operational semantics is simple and allows a clear understanding of the language and its use. On the basis of that, type theory may be used as a uniform language for programming, specification, and reasoning.

Another important aspect in the study of computer languages is that computer scientists design and invent languages to be *used* in real applications to, for example, large software development. This leads to the particular interests of computer scientists in studying such issues as modularisation and abstraction, which are among the central concerns in the design of programming and specification languages. These concerns are rather different as compared with those of meta-mathematicians in studying formal systems for the foundation of mathematics where what seems to prevail is the theoretical *possibility* of formalisation and foundational understanding of mathematics. In the research of programming methodology,

which is closely related to but has different emphasis from software engineering, computer scientists have been looking for theoretical foundations on the basis of which a science of programming or program development may be developed. The research on proof development systems or proof engineering has attracted more and more interest in computer science now that the need to verify various proof obligations in rigorous program development has been recognised. In these researches, people are interested not only in the fundamental understanding of the basic notions such as computation and proof, but in the methodological issues such as modularity which may lead to the design of good computer languages. Since types are useful tools in organising concepts, a type-theoretic language with a rich type structure can provide nice abstraction mechanisms for modular development of programs, specifications, and proofs.

Simplicity of languages is another issue considered by computer scientists as well as logicians. As logicians, computer scientists seek for simplicity for the fundamental understanding of the computer languages, but not just that. Furthermore, computer scientists are interested in the quality of software and the effectiveness of its production, where the simplicity of the languages in use (e.g. a simple semantics that makes it easier to learn and use a language) is important. Simplicity for a computer scientist also includes the implementability of the language. As we have remarked, type theory has a simple operational semantics, and it is a more manageable language as compared with classical set theory since its good proof-theoretic properties such as decidability provide a good basis for the computer implementation.

Looking for simplicity, the work by a theoretical computer scientist is in a certain sense 'foundational'. For instance, to study computer languages, he is usually working with a core language with essential constructions to explain the issues and problems of interest, which may be further developed into a full-scale language (e.g. with syntax sugar and other features) to be used in practice. However, doing so, he has a wider concern about the use of the language in practical applications since his study also involves the methodological issues mentioned above. Type theory may offer a computational and logical language for computer science, which is foundational in the above sense and adequate as claimed in the first paragraph of this chapter.

1.2 Basic concepts in type theory

1.2.1 Objects and types

Computer scientists and mathematicians consider various *constructions* or *objects*: computational objects like programs and specifications, and math-

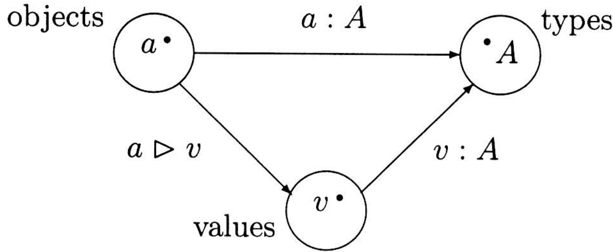


Fig. 1.1. Understanding the judgement $a : A$.

ematical objects like proofs and theorems. The world of objects under consideration may naturally be classified into *types*, to each of which some objects belong while the others do not. For example, the natural numbers constitute a type and the functions from type A to type B constitute the type of functions from A to B. We can therefore think of two *conceptual universes*, one of objects and the other of types (see Fig. 1.1). The relationship between (the representations of) objects and types is captured in type theory by the judgements of the form

$$a : A$$

that asserts that *object a is of type A*. Among the objects of a type, some are called *canonical objects*, which are the values of objects of the type under computation. For example, a canonical natural number is either zero or the successor of a (canonical) natural number. A canonical object of the type of functions is in the form of a λ-expression. In the presence of a (defined) operation of addition, the natural number $1+1$ is not canonical and computes with 2 as its value. A canonical object cannot be further computed and has itself as value.

The notion of *computation* is another basic concept in type theory, which generates an equivalence relation, the *computational equality* between the basic expressions in the language of type theory. For instance, applying a function $\lambda x{:}A.b[x]$ to an object a of type A yields $b[a]$. It is an important property of computation that *every object has a unique value under computation and the objects which are computationally equal have the same value.* As we shall see, this guarantees the harmony between the different uses of the entities in the type theory.

The distinction between canonical objects from the others, together with the above property of computation, is important in understanding

the language of type theory.[1] It allows us to understand the meanings of judgements, objects, and types in a direct and simple way (see Fig. 1.1). In particular, an object a being of type A, as asserted by judgement $a : A$, means that a computes into a canonical object of type A. That is to say, $a : A$ can be correctly asserted if the value of a is of type A. A type may then be understood as consisting of its canonical objects, relative to the understanding of computation. For instance, the type of natural numbers is understood as consisting of the canonical natural numbers, while the other natural numbers like $1 + 1$ are understood as programs that produce canonical natural numbers as their values.

On the basis of such a treatment of objects and types, a type theory with a rich type structure is a typed functional programming language, where programs are objects and functions (or functional programs) are first-class citizens. The notions of computation and canonical object give a basis for an operational semantics which allows a direct understanding of the programs in the language.

1.2.2 Propositions as types

Types may be viewed as partial specifications of their objects (programs). To describe and reason about the objects in type theory, one also needs logical formulas or propositions and incorporates logical inference. The key idea that makes this possible is that of *propositions-as-types*, discovered by Curry [CF58] and Howard [How80].

The basic idea is that any proposition P corresponds to a type $\mathbf{Prf}(P)$, the type of its proofs, and a proof of P corresponds to an object of type $\mathbf{Prf}(P)$. Furthermore, one is able to assert a proposition to be true if and only if one has a proof of the proposition, that is, an object of the type of its proofs. The origin of this idea goes back to the intuitionistic philosophy and, in particular, comes from Heyting and Kolmogorov's intuitionistic interpretation of logical operators (see [Hey71, Kol32]). For example, according to Heyting's interpretation, one has a proof of $P \supset Q$ (P implies Q) if and only if one has a construction which, whenever given a construction that proves P, yields a construction that proves Q. To cast such an informal semantics into a formal setting, a construction that proves $P \supset Q$ may be represented as a function that maps the proofs of P to proofs of Q, and the type of the proofs of $P \supset Q$, $\mathbf{Prf}(P \supset Q)$, may then be represented by the type of functions from $\mathbf{Prf}(P)$ to $\mathbf{Prf}(Q)$.

Being considered in this way, the truth of a proposition is understood by the inhabitance of the type of proofs of the proposition. The notion

[1] The notion of canonical object is a key idea in the verificationistic meaning theory (cf., the forceful arguments by Dummett, Prawitz and Martin-Löf [Dum75, Dum91] [Pra73, Pra74][ML84, ML85]).

of canonical objects for type $\mathbf{Prf}(P)$ gives a notion of *canonical or direct proofs* of proposition P, while the non-canonical objects of type $\mathbf{Prf}(P)$ may be called *indirect proofs*. The understanding of propositions and their proofs (and hence their truth and falsity) is then given, as we have discussed briefly in the last section. To be explicit, an object being a proof of a proposition P means that it computes into a canonical proof of P, and a proposition is true if and only if there is a canonical proof of it.

The above type-theoretic conception of logical inference is also based on the important idea, as emphasised by Martin-Löf, that there is a fundamental distinction between propositions, which are formulas describing properties and facts, and judgements, which are assertions of whether formulas are true. On the basis of this distinction, a type theory with sufficient logical type structures has an internal logic and presents a logical language rather different from that of set theory or that of logic programming. It is not a logical theory in the traditional sense in that judgements and the notion of computation are not sentences in some base logic such as first-order logic, and many mathematical concepts such as the set of natural numbers are introduced as types rather than axiomatic logical theories. However, this does not mean that type theory is 'logic-free'; its internal logic can provide powerful tools to describe and reason about the entities in the type theory.

1.2.3 Meaning and use

As explained above, the language of a type theory consists of objects and types which are related by the assertive sentences called judgements of the form $a : A$ whose meaning can essentially be understood as that a computes into a value (canonical object) of type A. Such a meaning explanation is verificationistic since verification is taken as the central concept in the operational meaning theory.

It should be made clear that taking verification as the central concept in meaning explanations does *not* mean that to understand a judgement, one only has to know *when it can be correctly asserted*, although that is an important aspect of its use. Another equally important aspect of the use of a judgement is about *what consequences it has to accept that a judgement is correct*. An understanding of a judgement is not complete unless one has grasped both of these two complementary aspects of its use.[2]

In type theory, as in other logical systems in natural deduction style, the above two aspects of use are essentially reflected by the introduction rules

[2]It is still questionable whether it is possible to have a satisfactory theory of meaning on the basis of which one can obtain or derive a complete understanding of the other aspect of use from the understanding of the aspect chosen to be the central concept for meaning. See [Dum91] for a very interesting explication.

and the elimination and computation rules, respectively. For example, in a language with dependent types, a type N of natural numbers may have introduction rules informally described as follows:

1. 0 is of type N.
2. If n is of type N, so is $succ(n)$.

The introduction rules determine what the canonical natural numbers are, and hence govern whether a judgement $a : N$ can be correctly asserted. The elimination and computation rules for N allows one to define functional operations with domain type N by primitive recursion in the form

$$F(0) = c, \qquad F(n+1) = f(n, F(n)).$$

The elimination rule introduces a recursion operator Rec_N such that

- for any family of types $C[x]$ indexed by natural numbers, any object of type $C[0]$, and any functional operation f that returns an object $f(m,c)$ of type $C[succ(m)]$ for any objects m of type N and c of type $C[m]$, $Rec_N(c,f)$ is a functional operation which for any natural number n returns an object of type $C[n]$.

The meaning of the recursion operator is given by the computation rules:

1. $Rec_N(c,f)(0)$ computes to c.
2. $Rec_N(c,f)(succ(n))$ computes to $f(n, Rec_N(c,f)(n))$.

Therefore, the elimination and computation rules determine what one can do once a judgement $n : N$ is correctly asserted, that is, how one may use such a fact to assert other judgements such as $Rec_N(c,f)(n) : C[n]$.

The understanding of objects and types can also be analysed into two aspects and may be derived from the above explanation of the uses of judgements. To understand a type, one must know what objects it may have or how it may be inhabited, and how its objects may be used or how its inhabitance implies that of other types. To understand an object, one needs to know its type(s) and how it may be used to define other objects. When a judgement is of the form $p : \mathbf{Prf}(P)$ asserting that p is a proof of proposition P, the above analysis of use gives rise to an analysis of the use of logical propositions and their proofs. The two aspects of use concern mainly about *when a logical proposition can be proved* and *what the logical consequences are when a proposition is accepted to be true*; these two aspects are reflected by the introduction rules and the elimination rules of the logical operators concerned, respectively.

此为试读,需要完整PDF请访问: www.ertongbook.com