SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE 1987

P R O C E E D I N G S

Symposium on
LOGIC IN COMPUTER SCIENCE

Ithaca, New York

June 22-25, 1987

Sponsored by the Computer Society of the IEEE
Technical Committee on
Mathematical Foundations of Computing

In cooperation with:
ACM SIGACT
Association for Symbolic Logic
European Association for Theoretical Computer Science

Support provided by:
Odyssey Research Associates
Cornell Mathematical Sciences Institute

Local arrangements supported by:
Cornell University

Computer Society Order Number 793
Library of Congress Number 87-45360
IEEE Catalog Number 87CH2464-6
ISBN 0-8186-0793-9
SAN 264-620X

THE COMPUTER SOCIETY OF THE IEEE

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

COMPUTER SOCIETY PRESS

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, Computer Society Press of the IEEE, or The Institute of Electrical and Electronics Engineers, Inc.

Published by Computer Society Press of the IEEE
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

Computer Society Order Number 793
Library of Congress Number [illegible]60
IEEE Catalog Number 87CH2[illegible]
ISBN 0-8186-0793-9 (paper[illegible]
ISBN 0-8186-4793-0 (microfich[illegible]
ISBN 0-8186-8793-2 (case)
SAN 264-620X

Order from: Computer Society of the IEEE
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

Computer Society of the IEEE
Avenue de la Tanche, 2
B-1160 Brussels
BELGIUM

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

FOREW0RD

The purpose of this second annual conference on Logic in Computer Science (LICS) is to bring together a wide range of issues in computer science broadly relating to logic, including algebraic and topological approaches. The LICS conferences evolved from the Logics of Programs workshops and have a substantially broader scope. The call for papers for this second annual LICS symposium included the following topics of interest: abstract data types, computer theorem proving, verification, concurrency, type theory and constructive mathematics, constructive proofs as programs, data base theory, foundations of logic programming, program logics and semantics, knowledge and belief, software specifications, logic-based programming languages, logic in complexity theory.

Organizing Committee:

K. Jon Barwise
Woodrow W. Bledsoe
Ashok K. Chandra (Chair)
Edsger W. Dijkstra
Erwin Engeler
Joseph A. Goguen
Dexter C. Kozen
Zohar Manna
Albert R. Meyer
Rohit Parikh
Gordon D. Plotkin
Dana S. Scott

CONFERENCE PROGRAM

The thirty-four papers in these Proceedings were selected by the Program Committee on January 19, 1987, from 126 extended abstracts submitted in response to the call for papers. A large number of worthy abstracts had to be rejected because of size limitations. The program committee wishes to thank all who submitted papers for consideration.

Neither the extended abstracts submitted to the Program Committee nor the final papers in these Proceedings went through a formal refereeing process. Selections were based on several criteria, including quality and originality, but also including presentability, appropriateness, and completeness. Many of the papers are preliminary reports of on-going research, and it is expected that many authors will publish more polished and complete versions in scientific journals.

Program Committee:

Stephen Brookes
Luca Cardelli
Robert Constable
Melvin Fitting
Joseph A. Goguen
David Gries (Chair)
Yuri Gurevich
David Harel
Jean-Pierre Jouannaud
Richard E. Ladner
Vladimir Lifschitz
Giuseppe Longo
Anil Nerode
Gordon Plotkin
Amir Pnueli
Philip Scott

CONFERENCE ORGANIZATION

Conference Chair

Ashok K. Chandra
IBM T.J. Watson Research Center

Program Chair

David Gries
Cornell University

Local Arrangements Chair

Dexter Kozen
Cornell University

Publicity Chair

David W. Bray
Clarkson University

Table of Contents

Tuesday, June 23, 1987

Invited Speaker

(Chair: P. Scott, *University of Ottawa*)

Session 4

(Chair: S. Brookes, *Carnegie-Mellon University*)

Wednesday, June 24, 1987

Invited Speaker

(Chair: J. Goguen, *SRI International*)

Session 5

(Chair: M. Fitting, *Herbert H. Lehman College*)

Session 6

(Chair: G. Plotkin, *University of Edinburgh*)

Session 7

(Chair: A Nerode, *Cornell University*)

Thursday, June 25, 1987

Session 8

(Chair: G. Longa, *Universita di Pisa*)

Session 9

(Chair: J.-P. Jouannaud. *Universite Paris-Sud-Orsay*)

Invited Speaker

Chair

A. Pnueli

Weizmann Institute of Science

Speaker

R. Milner

University of Edinburgh

Some Uses of Maximal Fixed Points

(Abstract of Invited Lecture)

R. Milner
Department of Computer Science
University of Edinburgh, Hope Park Square
Meadow Lane
Edinburgh EH8 9NW
SCOTLAND

The notions of indistinguishability and "lack of discrepancy" are captured by maximal fixed points. Results in concurrent processes and operational semantics will be discussed.

Session 1

Chair

L. Cardelli

Digital Equipment Corporation

Polymorphism is conservative over simple types
(Preliminary Report)

Val Breazu-Tannen *Albert R. Meyer*

Laboratory for Computer Science
MIT
Cambridge, MA 02139

Abstract. We prove that the addition of the Girard-Reynolds polymorphic constructs to arbitrary simply typed equational lambda theories is *conservative*. This implies that polymorphism can be superimposed on familiar programming languages without changing their behavior.

Using a purely syntactic method, we give an effective proof of conservative extension in the case of equational reasoning that is complete when all types are assumed non-empty. When polymorphic types may be empty, we prove the stronger result that any model of the simply typed lambda calculus can be *fully and faithfully embedded* in a model of the polymorphic lambda calculus.

1 Introduction

This paper is a sequel to a previous one, [BM87], where the main result presented here was briefly announced. We will not, however, assume that the reader is familiar with [BM87]; we now recapitulate some of our motivation.

In programming languages of universal power, the *computational* data type domains must be distinguished from the *classical* data types because of the "divergent" element. This is illustrated in [MR86], [BM87], by a typical example in which one starts with a straightforward algebraic specification (for an integer data type with a conditional operator) and adds to it the ability to have recursive function declarations. Using the "copy rule" (on recursive calls) and the axioms of the specification one can then prove equations between (algebraic) data type terms that the specification alone cannot prove [1]. Thus, the equational theory of the programming language with recursion is *not* a *conservative extension* of the data type specification.

In order to reason about the underlying data types in a semantics that accommodates recursion, we need a logic that takes non-termination into account. LCF [GMW79] or the partial lambda calculus [Plo85], [Mog], are such logics that take recursion as a must and try to reason about the resulting data domains. In both logics, however, when reasoning about expressions of data element type one needs to worry about *more* than the data type specification, namely about whether certain subexpressions terminate or whether they are defined.

In [BM87], we took a different course: we aimed to preserve classical reasoning about the data by achieving the kind of conservative extension that fails above. Instead of recursion, we added to the data type the constructions made possible by procedural and polymorphic abstraction.

Following familiar tradition [Lan65], we take lambda calculi with reduction rules as models of programming languages and their evaluation, and in particular the Girard-Reynolds polymorphic lambda calculus, denoted by $\lambda^{\forall}$, [Gir72], [Rey74], *cf.* [FLO83] or

This work was supported in part by NSF Grant DCR-8511190 and in part by ONR Grant N00014-83-K-0125. The first author was partially supported by an IBM Graduate Fellowship.

[1] In fact, in the example in [MR86] [BM87] such reasoning is *inconsistent*, i.e., *any* equation is provable.

[Mit84] as a formal model of polymorphic programming [2]. Its syntax is reviewed in Section 2. First, we modeled data type specifications by algebraic theories [GTW78]. Let $\alpha(\Sigma, E)$ be a many-sorted algebraic theory , where Σ is a many-sorted signature and E is a set of algebraic axioms. Let $\lambda^{\forall}(\Sigma, E)$ be the polymorphic lambda theory (an extension of $\lambda^{\forall}$) in which the sorts of Σ are added as type constants, the function symbols of Σ are added as constants (of suitably curried type), and the equations in E are added to the axioms of $\lambda^{\forall}$.

In [BM87], we proved that the addition of the polymorphic constructs to any algebraic data type specifications is conservative, *i.e.*, $\lambda^{\forall}(\Sigma, E)$ is a *conservative extension* of $\alpha(\Sigma, E)$.

We now go further and enrich our model for specifications from many-sorted algebras to certain higher-order theories, specifically *simply (finitely) typed lambda theories*. We will denote the (pure) simply typed lambda calculus [Fri75], [Bar84] with $\lambda^{\rightarrow}$. A simply typed theory $\lambda^{\rightarrow}(\Sigma, E)$ consists of base (ground) types out of which one builds simple (finite) types using the $\rightarrow$ operator, of a signature Σ of constant symbols of arbitrary simple type out of which one builds simply typed lambda terms and of a set E of arbitrary equational axioms between simply typed lambda terms which are added to the axioms of $\lambda^{\rightarrow}$. Let $\lambda^{\forall}(\Sigma, E)$ be the polymorphic lambda theory in which the base types, the constants in Σ and the additional axioms in E are added to $\lambda^{\forall}$ ($\lambda^{\rightarrow}$ is already contained in $\lambda^{\forall}$). The main result of this paper is

Theorem 1
For any simply typed theory $\lambda^{\rightarrow}(\Sigma, E)$, the extension $\lambda^{\forall}(\Sigma, E)$ is conservative over $\lambda^{\rightarrow}(\Sigma, E)$. That is, for any $\lambda^{\rightarrow}(\Sigma)$-terms M and N,

$$E \vdash^{\lambda^{\forall}} M = N \iff E \vdash^{\lambda^{\rightarrow}} M = N .$$

We remark that since adding $\lambda^{\rightarrow}$ to arbitrary algebraic theories is conservative [MR86], Theorem 1 implies the earlier result of [BM87].

In our view, what makes Theorem 1 considerably more interesting than the earlier result is the fact that more features, such as function and data type declarations, can be better and more naturally modeled by simply typed lambda theories than by algebraic theories. Indeed, while the pure simply typed lambda calculus does not get very far, the capability of having extra constants and extra equations to govern their behavior is quite powerful. For example, simply typed theories can be used to model full-fledged programming languages [THM84] by modeling unrestricted recursion via higher-order fixed point operators. Even arbitrary recursively defined types can be modeled, by axioms asserting isomorphism between types. For example, the untyped lambda calculus can be captured by declaring a type u together with constants $rep : (u \rightarrow u) \rightarrow u$ and $abs : u \rightarrow (u \rightarrow u)$ and axioms asserting that *rep* and *abs* are inverse to each other (*cf.* [GMW79] or [Sco80]).

Polymorphic type disciplines have recently enjoyed increased attention as the naturalness and usefulness of the types-as-values paradigm which they embody was recognized. As a result, the design of programming languages has witnessed the widespread adoption of polymorphic type systems. A number of examples and a survey of this field can be found in [CW85]. Theorem 1 shows that polymorphic constructs and reasoning can be added to any programming language features that can be described within the simple type discipline without changing the familiar behavior of these features. From this perspective, the adoption polymorphic type systems is *safe*.

There are two technical variants of the main theorem because there are two related proof systems for polymorphic lambda-calculus which in general yield set-theoretically incomparable theories from the same axioms. The systems differ in the assumption of whether polymorphic types may be empty. The original polymorphic proof system is sound and complete for deriving semantic consequences over all models with all types non-empty [BM84]. But this system is *not sound* in models with empty types. After arguing that such models are of interest, [MMMS87] gives a new proof system that is sound and complete for deriving semantic consequences over *all* models.

The bulk of this paper (Section 3) focuses on establishing conservative extension for the older, nonempty-types, proof system of [BM84]. Using purely syntactic methods, we give an *effective* proof

[2] The version we consider here has *universal* types but it does not have *existential* types.