Herbert Kuchen
Kazunori Ueda (Eds.)

LNCS 2024

Functional and Logic Programming

5th International Symposium, FLOPS 2001
Tokyo, Japan, March 2001
Proceedings

Springer

Herbert Kuchen Kazunori Ueda (Eds.)

Functional and Logic Programming

5th International Symposium, FLOPS 2001
Tokyo, Japan, March 7-9, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Herbert Kuchen
Westfälische Wilhelms-Universität Münster, Institut für Wirtschaftsinformatik
Steinfurter Straße 109, 48149 Münster, Germany
E-mail: kuchen@uni-muenster.de

Kazunori Ueda
Waseda University, Department of Information and Computer Science
4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169-8555, Japan
E-mail: ueda@ueda.info.waseda.ac.jp

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Functional and logic programming : 5th international symposium ;
proceedings / FLOPS 2001, Tokyo, Japan, March 7 - 9, 2001. Herbert
Kuchen ; Kazunori Ueda (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 2001
(Lecture notes in computer science ; Vol. 2024)
ISBN 3-540-41739-7

CR Subject Classification (1998): D.1.1, D.1.6, D.3, F.3, I.2.3

ISSN 0302-9743
ISBN 3-540-41739-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH
http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10782353 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 2024

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Preface

This volume contains the proceedings of the *Fifth International Symposium on Functional and Logic Programming*, FLOPS 2001, held in Tokyo, Japan, March 7–9, 2001, and hosted by Waseda University.

FLOPS is a forum for research on all issues concerning functional programming and logic programming. In particular, it aims to stimulate the cross-fertilization as well as the integration of the two paradigms. The previous FLOPS meetings took place in Fuji-Susono (1995), Shonan (1996), Kyoto (1998), and Tsukuba (1999). The proceedings of FLOPS'99 were published by Springer-Verlag as Lecture Notes in Computer Science, volume 1722.

There were 40 submissions, 38 of which were considered by the program committee. They came from Australia (5), Belgium (1/3), Denmark (3), Egypt (1), France (1/2), Germany (2 1/3), Italy (4 2/3), Japan (5), Korea (1 1/2), Mexico (1), The Netherlands (1 1/6), Spain (10 1/6), Switzerland (1), UK (1 5/6), and USA (1 1/2). Each paper was reviewed by at least three, and mostly four, reviewers. The program committee meeting was conducted electronically for the period of two weeks in November 2000. As a result of active discussions, 21 papers (52.5%) were selected for presentation, which appear in this volume. In addition, we are very pleased to include in this volume full papers by three distinguished invited speakers, namely Gopalan Nadathur, George Necula, and Taisuke Sato.

On behalf of the program committee, the program chairs would like to thank the invited speakers who agreed to give talks and contribute papers, all those who submitted papers, and all the referees for their careful work in the reviewing and selection process. The support of our sponsors is also gratefully acknowledged. In particular, we would like to thank the Japan Society for Software Science and Technology (JSSST), Special Interest Group on Principles of Programming, and the Association for Logic Programming (ALP). Finally, we would like to thank the members of the organizing committee, notably Zhenjiang Hu, Yasuhiro Ajiro, Kazuhiko Kakehi, and Madoka Kuniyasu, for their invaluable support throughout the preparation and organization of the symposium.

January 2001

Herbert Kuchen
Kazunori Ueda

Symposium Organization

Program Chairs

Herbert Kuchen	University of Münster, Germany
Kazunori Ueda	Waseda University, Tokyo, Japan

Program Committee

Sergio Antoy	Portland State University, USA
Gopal Gupta	University of Texas at Dallas, USA
Michael Hanus	University of Kiel, Germany
Fergus Henderson	University of Melbourne, Australia
Zhenjiang Hu	University of Tokyo, Japan
Herbert Kuchen	University of Münster, Germany
Giorgio Levi	University of Pisa, Italy
Michael Maher	Griffith University, Brisbane, Australia
Dale Miller	Pennsylvania State University, USA
I. V. Ramakrishnan	State University of New York at Stony Brook, USA
Olivier Ridoux	IRISA, Rennes, France
Mario Rodríguez-Artalejo	Complutense University, Madrid, Spain
Colin Runciman	University of York, UK
Akihiko Takano	Hitachi, Ltd., Japan
Peter Thiemann	Freiburg University, Germany
Yoshihito Toyama	Tohoku University, Japan
Kazunori Ueda	Waseda University, Tokyo, Japan

Local Arrangements Chair

Zhenjiang Hu	University of Tokyo, Japan

List of Referees

The following referees helped the program committee in evaluating the papers. Their assistance is gratefully acknowledged.

Yohji Akama
Joseph Albert
Kenichi Asai
Gilles Barthe
Cristiano Calcagno
Manuel M. T. Chakravarty
Alessandra Di Pierro
Rachid Echahed
Moreno Falaschi
Adrian Fiech
Peter Flach
Maurizio Gabbrielli
Maria García de la Banda
Antonio Gavilanes
Robert Glück
Stefano Guerrini
Hai-feng Guo
Bill Harrison
Simon Helsen
Martin Henz
Hideya Iwasaki
Mark Jones
Kazuhiko Kakehi
Owen Kaser
Robert Kowalski
K. Narayan Kumar
Keiichirou Kusakari
Javier Leach-Albert
Francisco López-Fraguas
Wolfgang Lux
Narciso Martí-Oliet
Andrea Masini
Bart Massey
Hidehiko Masuhara
Aart Middeldorp
Yasuhiko Minamide
Luc Moreau
Shin-ya Nishizaki
Susana Nieva
Mizuhito Ogawa
Satoshi Okui
Fernando Orejas
Giridhar Pemmasani
Marek Perkowski
Enrico Pontelli
C. R. Ramakrishnan
Francesca Rossi
Salvatore Ruggieri
Masahiko Sakai
Chiaki Sakama
Takafumi Sakurai
R. Sekar
Don Smith
Tran Cao Son
Harald Søndergaard
Frank Steiner
Eijiro Sumii
Taro Suzuki
Izumi Takeuti
Naoyuki Tamura
Tetsuro Tanaka
David Wakeling

Lecture Notes in Computer Science

For information about Vols. 1–1920
please contact your bookseller or Springer-Verlag

Vol. 1921: S.W. Liddle, H.C. Mayr, B. Thalheim (Eds.), Conceptual Modeling for E-Business and the Web. Proceedings, 2000. X, 179 pages. 2000.

Vol. 1922: J. Crowcroft, J. Roberts, M.I. Smirnov (Eds.), Quality of Future Internet Services. Proceedings, 2000. XI, 368 pages. 2000.

Vol. 1923: J. Borbinha, T. Baker (Eds.), Research and Advanced Technology for Digital Libraries. Proceedings, 2000. XVII, 513 pages. 2000.

Vol. 1924: W. Taha (Ed.), Semantics, Applications, and Implementation of Program Generation. Proceedings, 2000. VIII, 231 pages. 2000.

Vol. 1925: J. Cussens, S. Džeroski (Eds.), Learning Language in Logic. X, 301 pages 2000. (Subseries LNAI).

Vol. 1926: M. Joseph (Ed.), Formal Techniques in Real-Time and Fault-Tolerant Systems. Proceedings, 2000. X, 305 pages. 2000.

Vol. 1927: P. Thomas, H.W. Gellersen, (Eds.), Handheld and Ubiquitous Computing. Proceedings, 2000. X, 249 pages. 2000.

Vol. 1928: U. Brandes, D. Wagner (Eds.), Graph-Theoretic Concepts in Computer Science. Proceedings, 2000. X, 315 pages. 2000.

Vol. 1929: R. Laurini (Ed.), Advances in Visual Information Systems. Proceedings, 2000. XII, 542 pages. 2000.

Vol. 1931: E. Horlait (Ed.), Mobile Agents for Telecommunication Applications. Proceedings, 2000. IX, 271 pages. 2000.

Vol. 1658: J. Baumann, Mobile Agents: Control Algorithms. XIX, 161 pages. 2000.

Vol. 1756: G. Ruhe, F. Bomarius (Eds.), Learning Software Organization. Proceedings, 1999. VIII, 226 pages. 2000.

Vol. 1766: M. Jazayeri, R.G.K. Loos, D.R. Musser (Eds.), Generic Programming. Proceedings, 1998. X, 269 pages. 2000.

Vol. 1791: D. Fensel, Problem-Solving Methods. XII, 153 pages. 2000. (Subseries LNAI).

Vol. 1799: K. Czarnecki, U.W. Eisenecker, Generative and Component-Based Software Engineering. Proceedings, 1999. VIII, 225 pages. 2000.

Vol. 1812: J. Wyatt, J. Demiris (Eds.), Advances in Robot Learning. Proceedings, 1999. VII, 165 pages. 2000. (Subseries LNAI).

Vol. 1932: Z.W. Raś, S. Ohsuga (Eds.), Foundations of Intelligent Systems. Proceedings, 2000. XII, 646 pages. (Subseries LNAI).

Vol. 1933: R.W. Brause, E. Hanisch (Eds.), Medical Data Analysis. Proceedings, 2000. XI, 316 pages. 2000.

Vol. 1934: J.S. White (Ed.), Envisioning Machine Translation in the Information Future. Proceedings, 2000. XV, 254 pages. 2000. (Subseries LNAI).

Vol. 1935: S.L. Delp, A.M. DiGioia, B. Jaramaz (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2000. Proceedings, 2000. XXV, 1250 pages. 2000.

Vol. 1936: P. Robertson, H. Shrobe, R. Laddaga (Eds.), Self-Adaptive Software. Proceedings, 2000. VIII, 249 pages. 2001.

Vol. 1937: R. Dieng, O. Corby (Eds.), Knowledge Engineering and Knowledge Management. Proceedings, 2000. XIII, 457 pages. 2000. (Subseries LNAI).

Vol. 1938: S. Rao, K.I. Sletta (Eds.), Next Generation Networks. Proceedings, 2000. XI, 392 pages. 2000.

Vol. 1939: A. Evans, S. Kent, B. Selic (Eds.), «UML» – The Unified Modeling Language. Proceedings, 2000. XIV, 572 pages. 2000.

Vol. 1940: M. Valero, K. Joe, M. Kitsuregawa, H. Tanaka (Eds.), High Performance Computing. Proceedings, 2000. XV, 595 pages. 2000.

Vol. 1941: A.K. Chhabra, D. Dori (Eds.), Graphics Recognition. Proceedings, 1999. XI, 346 pages. 2000.

Vol. 1942: H. Yasuda (Ed.), Active Networks. Proceedings, 2000. XI, 424 pages. 2000.

Vol. 1943: F. Koornneef, M. van der Meulen (Eds.), Computer Safety, Reliability and Security. Proceedings, 2000. X, 432 pages. 2000.

Vol. 1944: K.R. Dittrich, G. Guerrini, I. Merlo, M. Oliva, M.E. Rodriguez (Eds.), Objects and Databases. Proceedings, 2000. X, 199 pages. 2001.

Vol. 1945: W. Grieskamp, T. Santen, B. Stoddart (Eds.), Integrated Formal Methods. Proceedings, 2000. X, 441 pages. 2000.

Vol. 1946: P. Palanque, F. Paternò (Eds.), Interactive Systems. Proceedings, 2000. X, 251 pages. 2001.

Vol. 1947: T. Sørevik, F. Manne, R. Moe, A.H. Gebremedhin (Eds.), Applied Parallel Computing. Proceedings, 2000. XII, 400 pages. 2001.

Vol. 1948: T. Tan, Y. Shi, W. Gao (Eds.), Advances in Multimodal Interfaces – ICMI 2000. Proceedings, 2000. XVI, 678 pages. 2000.

Vol. 1949: R. Connor, A. Mendelzon (Eds.), Research Issues in Structured and Semistructured Database Programming. Proceedings, 1999. XII, 325 pages. 2000.

Vol. 1950: D. van Melkebeek, Randomness and Completeness in Computational Complexity. XV, 196 pages. 2000.

Vol. 1951: F. van der Linden (Ed.), Software Architectures for Product Families. Proceedings, 2000. VIII, 255 pages. 2000.

Vol. 1952: M.C. Monard, J. Simão Sichman (Eds.), Advances in Artificial Intelligence. Proceedings, 2000. XV, 498 pages. 2000. (Subseries LNAI).

Vol. 1953: G. Borgefors, I. Nyström, G. Sanniti di Baja (Eds.), Discrete Geometry for Computer Imagery. Proceedings, 2000. XI, 544 pages. 2000.

Vol. 1954: W.A. Hunt, Jr., S.D. Johnson (Eds.), Formal Methods in Computer-Aided Design. Proceedings, 2000. XI, 539 pages. 2000.

Vol. 1955: M. Parigot, A. Voronkov (Eds.), Logic for Programming and Automated Reasoning. Proceedings, 2000. XIII, 487 pages. 2000. (Subseries LNAI).

Vol. 1956: T. Coquand, P. Dybjer, B. Nordström, J. Smith (Eds.), Types for Proofs and Programs. Proceedings, 1999. VII, 195 pages. 2000.

Vol. 1957: P. Ciancarini, M. Wooldridge (Eds.), Agent-Oriented Software Engineering. Proceedings, 2000. X, 323 pages. 2001.

Vol. 1960: A. Ambler, S.B. Calo, G. Kar (Eds.), Services Management in Intelligent Networks. Proceedings, 2000. X, 259 pages. 2000.

Vol. 1961: J. He, M. Sato (Eds.), Advances in Computing Science – ASIAN 2000. Proceedings, 2000. X, 299 pages. 2000.

Vol. 1963: V. Hlaváč, K.G. Jeffery, J. Wiedermann (Eds.), SOFSEM 2000: Theory and Practice of Informatics. Proceedings, 2000. XI, 460 pages. 2000.

Vol. 1964: J. Malenfant, S. Moisan, A. Moreira (Eds.), Object-Oriented Technology. Proceedings, 2000. XI, 309 pages. 2000.

Vol. 1965: Ç. K. Koç, C. Paar (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2000. Proceedings, 2000. XI, 355 pages. 2000.

Vol. 1966: S. Bhalla (Ed.), Databases in Networked Information Systems. Proceedings, 2000. VIII, 247 pages. 2000.

Vol. 1967: S. Arikawa, S. Morishita (Eds.), Discovery Science. Proceedings, 2000. XII, 332 pages. 2000. (Subseries LNAI).

Vol. 1968: H. Arimura, S. Jain, A. Sharma (Eds.), Algorithmic Learning Theory. Proceedings, 2000. XI, 335 pages. 2000. (Subseries LNAI).

Vol. 1969: D.T. Lee, S.-H. Teng (Eds.), Algorithms and Computation. Proceedings, 2000. XIV, 578 pages. 2000.

Vol. 1970: M. Valero, V.K. Prasanna, S. Vajapeyam (Eds.), High Performance Computing – HiPC 2000. Proceedings, 2000. XVIII, 568 pages. 2000.

Vol. 1971: R. Buyya, M. Baker (Eds.), Grid Computing – GRID 2000. Proceedings, 2000. XIV, 229 pages. 2000.

Vol. 1972: A. Omicini, R. Tolksdorf, F. Zambonelli (Eds.), Engineering Societies in the Agents World. Proceedings, 2000. IX, 143 pages. 2000. (Subseries LNAI).

Vol. 1973: J. Van den Bussche, V. Vianu (Eds.), Database Theory – ICDT 2001. Proceedings, 2001. X, 451 pages. 2001.

Vol. 1974: S. Kapoor, S. Prasad (Eds.), FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science. Proceedings, 2000. XIII, 532 pages. 2000.

Vol. 1975: J. Pieprzyk, E. Okamoto, J. Seberry (Eds.), Information Security. Proceedings, 2000. X, 323 pages. 2000.

Vol. 1976: T. Okamoto (Ed.), Advances in Cryptology – ASIACRYPT 2000. Proceedings, 2000. XII, 630 pages. 2000.

Vol. 1977: B. Roy, E. Okamoto (Eds.), Progress in Cryptology – INDOCRYPT 2000. Proceedings, 2000. X, 295 pages. 2000.

Vol. 1978: B. Schneier (Ed.), Fast Software Encryption. Proceedings, 2000. VIII, 315 pages. 2001.

Vol. 1979: S. Moss, P. Davidsson (Eds.), Multi-Agent-Based Simulation. Proceedings, 2000. VIII, 267 pages. 2001. (Subseries LNAI).

Vol. 1983: K.S. Leung, L.-W. Chan, H. Meng (Eds.), Intelligent Data Engineering and Automated Learning – IDEAL 2000. Proceedings, 2000. XVI, 573 pages. 2000.

Vol. 1984: J. Marks (Ed.), Graph Drawing. Proceedings, 2001. XII, 419 pages. 2001.

Vol. 1987: K.-L. Tan, M.J. Franklin, J. C.-S. Lui (Eds.), Mobile Data Management. Proceedings, 2001. XIII, 289 pages. 2001.

Vol. 1989: M. Ajmone Marsan, A. Bianco (Eds.), Quality of Service in Multiservice IP Networks. Proceedings, 2001. XII, 440 pages. 2001.

Vol. 1991: F. Dignum, C. Sierra (Eds.), Agent Mediated Electronic Commerce. VIII, 241 pages. 2001. (Subseries LNAI).

Vol. 1992: K. Kim (Ed.), Public Key Cryptography. Proceedings, 2001. XI, 423 pages. 2001.

Vol. 1993: E. Zitzler, K. Deb, L. Thiele, C.A.Coello Coello, D. Corne (Eds.), Evolutionary Multi-Criterion Optimization. Proceedings, 2001. XIII, 712 pages. 2001.

Vol. 1995: M. Sloman, J. Lobo, E.C. Lupu (Eds.), Policies for Distributed Systems and Networks. Proceedings, 2001. X, 263 pages. 2001.

Vol. 1998: R. Klette, S. Peleg, G. Sommer (Eds.), Robot Vision. Proceedings, 2001. IX, 285 pages. 2001.

Vol. 2000: R. Wilhelm (Ed.), Informatics: 10 Years Back, 10 Years Ahead. IX, 369 pages. 2001.

Vol. 2003: F. Dignum, U. Cortés (Eds.), Agent Mediated Electronic Commerce III. XII, 193 pages. 2001. (Subseries LNAI).

Vol. 2004: A. Gelbukh (Ed.), Computational Linguistics and Intelligent Text Processing. Proceedings, 2001. XII, 528 pages. 2001.

Vol. 2006: R. Dunke, A. Abran (Eds.), New Approaches in Software Measurement. Proceedings, 2000. VIII, 245 pages. 2001.

Vol. 2009: H. Federrath (Ed.), Designing Privacy Enhancing Technologies. Proceedings, 2000. X, 231 pages. 2001.

Vol. 2010: A. Ferreira, H. Reichel (Eds.), STACS 2001. Proceedings, 2001. XV, 576 pages. 2001.

Vol. 2024: H. Kuchen, K. Ueda (Eds.), Functional and Logic Programming. Proceedings, 2001. X, 391 pages. 2001.

Table of Contents

The Metalanguage λProlog and Its Implementation

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS Building, 200 Union Street SE
Minneapolis, MN 55455
gopalan@cs.umn.edu
Home Page: http://www.cs.umn.edu/~gopalan

Abstract. Stimulated by concerns of software certification especially as it relates to mobile code, formal structures such as specifications and proofs are beginning to play an explicit role in computing. In representing and manipulating such structures, an approach is needed that pays attention to the binding operation that is present in them. The language λProlog provides programming support for a higher-order treatment of abstract syntax that is especially suited to this task. This support is realized by enhancing the traditional strength of logic programming in the metalanguage realm with an ability for dealing directly with binding structure. This paper identifies the features of λProlog that endow it with such a capability, illustrates their use and and describes methods for their implementation. Also discussed is a new realization of λProlog called *Teyjus* that incorporates the implementation ideas presented.

1 Introduction

The language λProlog is based on the higher-order theory of hereditary Harrop formulas that embodies a rich interpretation of the abstract idea of logic programming [18]. Through a systematic exploitation of features present in the underlying logic, this language realizes several capabilities at the programming level such as ones for typing, scoping over names and procedure definitions, representing and manipulating complex formal structures, modularly constructing code and higher-order programming. Our interest in this paper is in one specific facet of λProlog: its role as a metalanguage.

The manipulation of symbolic expressions has been of longstanding interest and some of the earliest computational tasks to have been considered and systematically addressed have, in fact, concerned the realization of reasoning processes, the processing of human languages and the compilation and interpretation of programming languages. The calculations involved in these cases are typically metalinguistic and syntactic in nature and a careful study of their structure has produced a universally accepted set of concepts and tools relevant to this form of computing. An important component in this collection is the

H. Kuchen and K. Ueda (Eds.): FLOPS 2001, LNCS 2024, pp. 1–20, 2001.

idea of *abstract syntax* that moves away from concrete presentation and focuses instead on the essential relationships between the constituent parts of symbolic constructs. A complementary development has been that of languages that provide programming support for computing with abstract syntax. These languages, which include Lisp, ML and Prolog amongst them, contain mechanisms that simplify the representation, construction and deconstruction of abstract syntax and that permit the implicit management of space relative to such manipulations. Effort has also been invested in implementing these languages efficiently, thereby making them practical vehicles for realizing complex symbolic systems.

One may wonder against this backdrop if anything new really needs to be added to the capabilities already available for symbolic computation. The answer to this question revolves around the treatment of scope and binding. Many symbolic objects whose manipulation is of interest involve forms of these operations in their structure in addition to the compositionality that is traditionally treated in abstract syntax. This is true, for instance, of quantified formulas that are considered within reasoning systems and of procedures with arguments that are of interest to programming language compilers. The conventional approach in these cases has been to use auxiliary mechanisms to avoid explicit reference to binding in representation. Thus, reasoning systems eliminate quantifiers from formulas through a preprocessing phase and compilers utilize symbol tables to create binding environments when these are needed in the analysis of programs. While such methods have been successful in the past, there is now an increasing interest in formal constructs with sophisticated and diverse forms of scope whose uniform treatment requires a reflection of the binding operation into abstract syntax itself. The desire to reason in systems different from classical logic provides one example of this kind. The elimination of quantifiers may either not be possible or desirable in many of these cases, requiring them to be explicitly represented and dynamically treated by the reasoning process. In a similar vein, motivated by the proof-carrying-code approach to software certification [29], attention has been paid to the representation of proofs. The discharge of assumptions and the treatment of genericity are intrinsic to these formal structures and a convenient method for representing such operations involves the use of binding constructs that range over their subparts. As a final example, relationships between declarations and uses are an important part of program structure and a formal treatment of these in representation can influence new approaches to program analysis and transformation.

Driven by considerations such as these, much effort has recently been devoted to developing an explicit treatment of binding in syntax representation, culminating in what has come to be known as *higher-order abstract syntax* [31]. The main novelty of λProlog as a metalanguage lies in the support it offers for this new approach to encoding syntactic objects. It realizes this support by enriching a conventional logic programming language in three essential ways. First, it replaces first-order terms—the data structures of a logic programming language—by the terms of a typed lambda calculus. Attendant on these lambda terms is a notion of equality given by the α-, β- and η-conversion rules. The main

difference in representational power between first-order terms and lambda terms is that the latter are capable of also capturing binding structure in a logically precise way. Thus, this enhancement in term structure endows λProlog with a means for representing higher-order abstract syntax. Second, λProlog uses a unification operation that builds in the extended notion of equality accompanying lambda terms. This change provides the language with a destructuring operation that can utilize information about binding structure. Finally, the language incorporates two new kinds of goals, these being expressions of the form $\forall x G$ and D => G, in which G is a goal and D is a conjunction of clauses.[1] A goal of the form $\forall x G$ is solved by replacing all free occurrences of x in G with a new constant and then solving the result and a goal of the form D => G is solved by enhancing the existing program with the clauses in D and then attempting to solve G. Thus, at a programming level, the new forms of goals, which are referred to as *generic* and *augment*, respectively, provide mechanisms for scoping over names and code. As we shall see presently, these scoping abilities can be used to realize recursion over binding structure.

Our objective in this paper is to show that the new features present in λProlog can simplify the programming of syntax manipulations and that they can be implemented with sufficient efficiency to be practical tools in this realm. Towards this end, we first motivate the programming uses of these features and then discuss the problems and approaches to realizing them in an actual system. The ideas we discuss here have been used in a recent implementation of λProlog called *Teyjus* [25] that we also briefly describe. We assume a basic familiarity with lambda calculus notions and logic programming languages and the methods for implementing them that are embedded, for instance, in the Warren Abstract Machine (WAM) [35]. Further, in keeping with the expository nature of the paper, we favor an informal style of presentation; all the desired formality can be found in references that are cited at relevant places.

2 Higher-Order Abstract Syntax in λProlog

A common refrain in symbolic computation is to focus on the essential functional structure of objects. This is true, for instance, of systems that manipulate programs. Thus, a compiler or interpreter that manipulates an expression of the form *if* B *then* T *else* E must recognize that this expression denotes a conditional involving three constituents: B, T and E. Similarly, a theorem prover that

[1] To recall terminology, a goal is what appears in the body of a procedure or as a top level query and is conventionally formed from atomic goals via conjunction, disjunction and existential quantification. Clauses correspond to procedure definitions. While a free variable in a clause is usually assumed to be implicitly universally quantified at the head of the clause, there is ambiguity about the scope and force of such quantification when the clause appears in an expression of the form D => G. λProlog interprets the scope in this case to be the entire expression of which D => G itself may only be a part, and it bases the force on whether this expression is a goal or a clause. All other interpretations need to be indicated through explicit quantification.

encounters the formula $P \wedge Q$, must realize that this is one representing the conjunction of P and Q. Conversely, assuming that we are not interested in issues of presentation, these are the *only* properties that needs to be recognized and represented in each case. The 'abstract syntax' of these expressions may therefore be captured by the expressions *cond(B,T,E)* and *and(P,Q)*, where *cond* and *and* are suitably chosen function symbols or constructors.

Another important idea in syntax based computations is that of structural operational semantics that advocates the description of computational content through rules that operate on abstract syntax. For example, using as an infix notation for the evaluation relation, the operational meaning of a conditional expression can be described through the rules

$$\frac{B \quad true \qquad T \quad V}{cond(B,T,E) \quad V}$$

$$\frac{B \quad false \qquad E \quad V}{cond(B,T,E) \quad V}$$

Similarly, assuming that $\Gamma \longrightarrow F$ represents the judgement that F follows from a set of assumptions Γ, the logical content of a conjunction can be captured in the rule

$$\frac{\Gamma \longrightarrow P \qquad \Gamma \longrightarrow Q}{\Gamma \longrightarrow and(P,Q)}$$

Rules such as these can be used in combination with some control regimen determining their order of application to actually evaluate programs or to realize reasoning processes.

The appropriateness of a logic programming language for symbolic computation arises from the fact that it provides natural expression to both abstract syntax and rule based specifications. Thus, expressions of the form *cond(B,T,E)* and *and(P,Q)* are directly representable in such a language, being first-order terms. Depending on what they are being matched with, the unification operation relative to these terms provides a means for constructing, deconstructing or recognizing patterns in abstract syntax. Structural operational rules translate directly to program clauses. The evaluation rules for conditional expressions can, for instance, be represented by the clauses

```
eval(cond(B,T,E),V) :- eval(B,true), eval(T,V).
eval(cond(B,T,E),V) :- eval(B,false), eval(E,V).
```

Using these rules to realize interpretation may require capturing additional control information, but this can be done through the usual programming devices.

2.1 The Explicit Representation of Binding

Many syntactic objects involve a form of binding and it may sometimes be necessary to reflect this explicitly in their representation. Binding structure can be

represented only in an approximate manner using conventional abstract syntax or first order terms. For example, consider the formula $\forall x P(x)$. This formula may be represented by the expression $all(x, P(x))$. However, this representation misses important characteristics of quantification. Thus, the equivalence of $\forall x P(x)$ and $\forall y P(y)$ is not immediately present in the 'first-order' rendition and has to be built in through auxiliary processes. In a related sense, suppose it is necessary to instantiate the outer quantifier in the formula $\forall x \exists y P(x, y)$ with the term $t(y)$. The renaming required in carrying out this operation has to be explicitly programmed under the indicated representation.

The availability of lambda terms in λProlog provides a different method for dealing with these issues. A binding operator has two different characteristics: it determines a scope and it identifies a particular kind of term. In λProlog, the latter role may be captured by a suitably chosen constructor while the effect of scope may be reflected into a (metalanguage) abstraction. This form of representation is one of the main components of the higher-order approach to abstract syntax. Using this approach, the formula $\forall x P(x)$ might be rendered into the term *(all λx(P x))*, where *all* is a constructor chosen to represent the predicative force of the universal quantifier; we employ an infix, curried notation for application here and below as is customary for higher-order languages, but the correspondence to the first-order syntax should be evident. Similarly, the program fragment

*lambda (x) if (x = 0) then (x - 2) else (2 * x)*

in a Lisp-like language might be represented by the term

(abs λx(cond (eq x 0) (minus x 2) (times 2 x)))

where *abs* is a constructor that identifies an object language abstraction and *eq*, *plus*, *minus*, *0*, and *2* are constructors corresponding to the relevant programming language primitives. As a final, more involve example, consider the following code in a functional programming language:

*fact m n = if (m = 0) then n else (fact (m - 1) (m * n))*

This code identifies *fact* as a function of two arguments that is defined through a fixed point construction. Towards making this structure explicit, the given program fragment may be rewritten as

fact = (fixpt (f) (lambda (m) lambda (n)
*if (m = 0) then n else (f (m - 1) (m * n))))*

assuming that *fixpt* represents a binding operator akin to *lambda*. Now, using the constructor *fix* to represent this operator and *app* to represent object language application, the expression that is identified with *fact* may be rendered into the following λProlog term:[2]

[2] We are taking liberties with λProlog syntax here: the language employs a different notation for abstraction and all expressions in it are typed. In a more precise presen-