Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

87

5th Conference on Automated Deduction

Les Arcs, France, 1980

Edited by W. Bibel and R. Kowalski

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

87

5th Conference on Automated Deduction

Les Arcs, France, July 8–11, 1980

Edited by W. Bibel and R. Kowalski

Springer-Verlag
Berlin Heidelberg New York 1980

Editorial Board

W. Brauer P. Brinch Hansen D. Gries C. Moler G. Seegmüller
J. Stoer N. Wirth

Editors

Wolfgang Bibel
Institut für Informatik, Technische Universität München, Postfach 20 24 20
8000 München 2/Germany

Robert Kowalski
Department of Computing and Control, Imperial College,
180 Queen's Gate
London, SW7 2 BZ/England

AMS Subject Classifications (1979): 68 A 40, 68 A 45
CR Subject Classifications (1974): 5.21, 3.60

ISBN 3-540-10009-1 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10009-1 Springer-Verlag New York Heidelberg Berlin

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically those of translation, reprinting, re-use of illustrations, broadcasting, reproduction by photocopying machine or similar means, and storage in data banks. Under § 54 of the German Copyright Law where copies are made for other than private use, a fee is payable to the publisher, the amount of the fee to be determined by agreement with the publisher.

© by Springer-Verlag Berlin Heidelberg 1980
Printed in Germany

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Lecture Notes in Computer Science

Vol. 1: GI-Gesellschaft für Informatik e.V. 3. Jahrestagung, Hamburg, 8.–10. Oktober 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von W. Brauer. XI, 508 Seiten. 1973.

Vol. 2: GI-Gesellschaft für Informatik e.V. 1. Fachtagung über Automatentheorie und Formale Sprachen, Bonn, 9.–12. Juli 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von K.-H. Böhling und K. Indermark. VII, 322 Seiten. 1973.

Vol. 3: 5th Conference on Optimization Techniques, Part I. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 565 pages. 1973.

Vol. 4: 5th Conference on Optimization Techniques, Part II. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 389 pages. 1973.

Vol. 5: International Symposium on Theoretical Programming. Edited by A. Ershov and V. A. Nepomniaschy. VI, 407 pages. 1974.

Vol. 6: B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, Matrix Eigensystem Routines – EISPACK Guide. XI, 551 pages. 2nd Edition 1974. 1976.

Vol. 7: 3. Fachtagung über Programmiersprachen, Kiel, 5.–7. März 1974. Herausgegeben von B. Schlender und W. Friedinghaus. VI, 225 Seiten. 1974.

Vol. 8: GI-NTG Fachtagung über Struktur und Betrieb von Rechensystemen, Braunschweig, 20.–22. März 1974. Herausgegeben im Auftrag der GI und der NTG von H.-O. Leilich. VI, 340 Seiten. 1974.

Vol. 9: GI-BIFOA Internationale Fachtagung: Informationszentren in Wirtschaft und Verwaltung. Köln, 17./18. Sept. 1973. Herausgegeben im Auftrag der GI und dem BIFOA von P. Schmitz. VI, 259 Seiten. 1974.

Vol. 10: Computing Methods in Applied Sciences and Engineering, Part 1. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 497 pages. 1974.

Vol. 11: Computing Methods in Applied Sciences and Engineering, Part 2. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 434 pages. 1974.

Vol. 12: GFK-GI-GMR Fachtagung Prozessrechner 1974. Karlsruhe, 10.–11. Juni 1974. Herausgegeben von G. Krüger und R. Friehmelt. XI, 620 Seiten. 1974.

Vol. 13: Rechnerstrukturen und Betriebsprogrammierung, Erlangen, 1970. (GI-Gesellschaft für Informatik e.V.) Herausgegeben von W. Händler und P. P. Spies. VII, 333 Seiten. 1974.

Vol. 14: Automata, Languages and Programming – 2nd Colloquium, University of Saarbrücken, July 29–August 2, 1974. Edited by J. Loeckx. VIII, 611 pages. 1974.

Vol. 15: L Systems. Edited by A. Salomaa and G. Rozenberg. VI, 338 pages. 1974.

Vol. 16: Operating Systems, International Symposium, Rocquencourt 1974. Edited by E. Gelenbe and C. Kaiser. VIII, 310 pages. 1974.

Vol. 17: Rechner-Gestützter Unterricht RGU '74, Fachtagung, Hamburg, 12.–14. August 1974, ACU-Arbeitskreis Computer-Unterstützter Unterricht. Herausgegeben im Auftrag der GI von K. Brunnstein, K. Haefner und W. Händler. X, 417 Seiten. 1974.

Vol. 18: K. Jensen and N. E. Wirth, PASCAL – User Manual and Report. VII, 170 pages. Corrected Reprint of the 2nd Edition 1976.

Vol. 19: Programming Symposium. Proceedings 1974. V, 425 pages. 1974.

Vol. 20: J. Engelfriet, Simple Program Schemes and Formal Languages. VII, 254 pages. 1974.

Vol. 21: Compiler Construction, An Advanced Course. Edited by F. L. Bauer and J. Eickel. XIV. 621 pages. 1974.

Vol. 22: Formal Aspects of Cognitive Processes. Proceedings 1972. Edited by T. Storer and D. Winter. V, 214 pages. 1975.

Vol. 23: Programming Methodology. 4th Informatik Symposium, IBM Germany Wildbad, September 25–27, 1974. Edited by C. E. Hackl. VI, 501 pages. 1975.

Vol. 24: Parallel Processing. Proceedings 1974. Edited by T. Feng. VI, 433 pages. 1975.

Vol. 25: Category Theory Applied to Computation and Control. Proceedings 1974. Edited by E. G. Manes. X, 245 pages. 1975.

Vol. 26: GI-4. Jahrestagung, Berlin, 9.–12. Oktober 1974. Herausgegeben im Auftrag der GI von D. Siefkes. IX, 748 Seiten. 1975.

Vol. 27: Optimization Techniques. IFIP Technical Conference. Novosibirsk, July 1–7, 1974. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by G. I. Marchuk. VIII, 507 pages. 1975.

Vol. 28: Mathematical Foundations of Computer Science. 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974. Edited by A. Blikle. VII, 484 pages. 1975.

Vol. 29: Interval Mathematics. Procedings 1975. Edited by K. Nickel. VI, 331 pages. 1975.

Vol. 30: Software Engineering. An Advanced Course. Edited by F. L. Bauer. (Formerly published 1973 as Lecture Notes in Economics and Mathematical Systems, Vol. 81) XII, 545 pages. 1975.

Vol. 31: S. H. Fuller, Analysis of Drum and Disk Storage Units. IX, 283 pages. 1975.

Vol. 32: Mathematical Foundations of Computer Science 1975. Proceedings 1975. Edited by J. Bečvář. X, 476 pages. 1975.

Vol. 33: Automata Theory and Formal Languages, Kaiserslautern, May 20–23, 1975. Edited by H. Brakhage on behalf of GI. VIII, 292 Seiten. 1975.

Vol. 34: GI – 5. Jahrestagung, Dortmund 8.–10. Oktober 1975. Herausgegeben im Auftrag der GI von J. Mühlbacher. X, 755 Seiten. 1975.

Vol. 35: W. Everling, Exercises in Computer Systems Analysis. (Formerly published 1972 as Lecture Notes in Economics and Mathematical Systems, Vol. 65) VIII, 184 pages. 1975.

Vol. 36: S. A. Greibach, Theory of Program Structures: Schemes, Semantics, Verification. XV, 364 pages. 1975.

Vol. 37: C. Böhm, λ-Calculus and Computer Science Theory. Proceedings 1975. XII, 370 pages. 1975.

Vol. 38: P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaille, M. Vanbegin. An Optimized Translation Process and Its Application to ALGOL 68. IX, 334 pages. 1976.

Vol. 39: Data Base Systems. Proceedings 1975. Edited by H. Hasselmeier and W. Spruth. VI, 386 pages. 1976.

Vol. 40: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 1. Proceedings 1975. Edited by J. Cea. XIV, 854 pages. 1976.

Vol. 41: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 2. Proceedings 1975. Edited by J. Cea. XIII, 852 pages. 1976.

Vol. 42: James E. Donahue, Complementary Definitions of Programming Language Semantics. VII, 172 pages. 1976.

Vol. 43: E. Specker und V. Strassen, Komplexität von Entscheidungsproblemen. Ein Seminar. V, 217 Seiten. 1976.

Vol. 44: ECI Conference 1976. Proceedings 1976. Edited by K. Samelson. VIII, 322 pages. 1976.

Vol. 45: Mathematical Foundations of Computer Science 1976. Proceedings 1976. Edited by A. Mazurkiewicz. XI, 601 pages. 1976.

Vol. 46: Language Hierarchies and Interfaces. Edited by F. L. Bauer and K. Samelson. X, 428 pages. 1976.

Vol. 47: Methods of Algorithmic Language Implementation. Edited by A. Ershov and C. H. A. Koster. VIII, 351 pages. 1977.

Vol. 48: Theoretical Computer Science, Darmstadt, March 1977. Edited by H. Tzschach, H. Waldschmidt and H. K.-G. Walter on behalf of GI. VIII, 418 pages. 1977.

F O R E W O R D

This FIFTH CONFERENCE ON AUTOMATED DEDUCTION, held at Les Arcs, Savoie, France, July 8 - 11, 1980, was preceded by earlier meetings at Argonne, Illinois (1974), Oberwolfach, West Germany (1976), Cambridge, Massachusetts (1977), and Austin, Texas (1979).

This volume contains the papers which were selected by the program committee from the 62 papers submitted to the conference. These papers range over most of the main approaches to the automation of deductive reasoning. They describe both theoretical status and practical investigations of implementations and their applications, especially in computing science.

PROGRAM COMMITTEE

W. Bibel, München (program co-chairman);
W.W. Bledsoe, Austin;
A. Colmerauer, Marseille;
M. Davis, New York;
L. Henschen, Evanston;
R. Kowalski, London (program co-chairman);
D. Mc Dermott, New Haven;
R. Milner, Edinburgh;
U. Montanari, Pisa;
J S. Moore, Menlo Park;
D. Oppen, Stanford;
V. Pratt, Cambridge;
J.A. Robinson, Syracuse;
S. Sickel, Santa Cruz (past program chairman).

ORGANISER

G. Huet, INRIA, Rocquencourt.

此为试读,需要完整PDF请访问: www.ertongbook.com

Vol. 49: Interactive Systems. Proceedings 1976. Edited by A. Blaser and C. Hackl. VI, 380 pages. 1976.

Vol. 50: A. C. Hartmann, A Concurrent Pascal Compiler for Minicomputers. VI, 119 pages. 1977.

Vol. 51: B. S. Garbow, Matrix Eigensystem Routines – Eispack Guide Extension. VIII, 343 pages. 1977.

Vol. 52: Automata, Languages and Programming. Fourth Colloquium, University of Turku, July 1977. Edited by A. Salomaa and M. Steinby. X, 569 pages. 1977.

Vol. 53: Mathematical Foundations of Computer Science. Proceedings 1977. Edited by J. Gruska. XII, 608 pages. 1977.

Vol. 54: Design and Implementation of Programming Languages. Proceedings 1976. Edited by J. H. Williams and D. A. Fisher. X, 496 pages. 1977.

Vol. 55: A. Gerbier, Mes premières constructions de programmes. XII, 256 pages. 1977.

Vol. 56: Fundamentals of Computation Theory. Proceedings 1977. Edited by M. Karpiński. XII, 542 pages. 1977.

Vol. 57: Portability of Numerical Software. Proceedings 1976. Edited by W. Cowell. VIII, 539 pages. 1977.

Vol. 58: M. J. O'Donnell, Computing in Systems Described by Equations. XIV, 111 pages. 1977.

Vol. 59: E. Hill, Jr., A Comparative Study of Very Large Data Bases. X, 140 pages. 1978.

Vol. 60: Operating Systems, An Advanced Course. Edited by R. Bayer, R. M. Graham, and G. Seegmüller. X, 593 pages. 1978.

Vol. 61: The Vienna Development Method: The Meta-Language. Edited by D. Bjørner and C. B. Jones. XVIII, 382 pages. 1978.

Vol. 62: Automata, Languages and Programming. Proceedings 1978. Edited by G. Ausiello and C. Böhm. VIII, 508 pages. 1978.

Vol. 63: Natural Language Communication with Computers. Edited by Leonard Bolc. VI, 292 pages. 1978.

Vol. 64: Mathematical Foundations of Computer Science. Proceedings 1978. Edited by J. Winkowski. X, 551 pages. 1978.

Vol. 65: Information Systems Methodology, Proceedings, 1978. Edited by G. Bracchi and P. C. Lockemann. XII, 696 pages. 1978.

Vol. 66: N. D. Jones and S. S. Muchnick, TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages. IX, 118 pages. 1978.

Vol. 67: Theoretical Computer Science, 4th GI Conference, Aachen, March 1979. Edited by K. Weihrauch. VII, 324 pages. 1979.

Vol. 68: D. Harel, First-Order Dynamic Logic. X, 133 pages. 1979.

Vol. 69: Program Construction. International Summer School. Edited by F. L. Bauer and M. Broy. VII, 651 pages. 1979.

Vol. 70: Semantics of Concurrent Computation. Proceedings 1979. Edited by G. Kahn. VI, 368 pages. 1979.

Vol. 71: Automata, Languages and Programming. Proceedings 1979. Edited by H. A. Maurer. IX, 684 pages. 1979.

Vol. 72: Symbolic and Algebraic Computation. Proceedings 1979. Edited by E. W. Ng. XV, 557 pages. 1979.

Vol. 73: Graph-Grammars and Their Application to Computer Science and Biology. Proceedings 1978. Edited by V. Claus, H. Ehrig and G. Rozenberg. VII, 477 pages. 1979.

Vol. 74: Mathematical Foundations of Computer Science. Proceedings 1979. Edited by J. Bečvář. IX, 580 pages. 1979.

Vol. 75: Mathematical Studies of Information Processing. Proceedings 1978. Edited by E. K. Blum, M. Paul and S. Takasu. VIII, 629 pages. 1979.

Vol. 76: Codes for Boundary-Value Problems in Ordinary Differential Equations. Proceedings 1978. Edited by B. Childs et al. VIII, 388 pages. 1979.

Vol. 77: G. V. Bochmann, Architecture of Distributed Computer Systems. VIII, 238 pages. 1979.

Vol. 78: M. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF. VIII, 159 pages. 1979.

Vol. 79: Language Design and Programming Methodology. Proceedings, 1979. Edited by J. Tobias. IX, 255 pages. 1980.

Vol. 80: Pictorial Information Systems. Edited by S. K. Chang and K. S. Fu. IX, 445 pages. 1980.

Vol. 81: Data Base Techniques for Pictorial Applications. Proceedings, 1979. Edited by A. Blaser. XI, 599 pages. 1980.

Vol. 82: J. G. Sanderson, A Relational Theory of Computing. VI, 147 pages. 1980.

Vol. 83: International Symposium Programming. Proceedings, 1980. Edited by B. Robinet. VII, 341 pages. 1980.

Vol. 84: Net Theory and Applications. Proceedings, 1979. Edited by W. Brauer. XIII, 537 Seiten. 1980.

Vol. 85: Automata, Languages and Programming. Proceedings, 1980. Edited by J. de Bakker and J. van Leeuwen. VIII, 671 pages. 1980.

Vol. 86: Abstract Software Specifications. Proceedings, 1979. Edited by D. Bjørner. XIII, 567 pages. 1980

Vol. 87: 5th Conference on Automated Deduction. Proceedings, 1980. Edited by W. Bibel and R. Kowalski. VII, 385 pages. 1980.

CONTENTS

USING META-THEORETIC REASONING TO DO ALGEBRA

Luigia Aiello (†,‡)
Richard W. Weyhrauch (‡)

(†) Istituto di Elaborazione della Informazione, CNR, Pisa, Italy
(‡) Artificial Intelligence Laboratory, Stanford University, Stanford, USA

ABSTRACT

We report on an experiment in interactive reasoning with FOL. The subject of the reasoning is elementary algebra. The main point of the paper is to show how the use of meta-theoretic knowledge results in improving the *quality* of the resulting proofs in that, in this environment, they are both easier to find and easier to understand.

1. INTRODUCTION

In this paper we report on an experiment in interactive reasoning, namely reasoning with FOL about elementary algebra.

FOL is a conversational system designed by Richard Weyhrauch at the Stanford Artificial Intelligence Laboratory. It runs in LISP on a KL10. It implements First Order Logic using the natural deduction formalism of Prawitz [Pra65] enriched in many ways. In this paper we make no attempt to be self-contained, mostly because of space limitations. We refer to the literature [Wey77,78,79;Aie80;Fil78,79;Tal80] for an introduction to FOL, its many features and some examples of its use.

We chose elementary algebra as the subject of our reasoning because it is a commonly known field and has an established axiomatic mathematical presentation. Another reason for this choice is that future applications of FOL will require knowledge of the algebraic facts of arithmetic. Hence, it is important to provide FOL with the ability to reason about arithmetic and to discover what arithmetic and meta-arithmetic facts are used in ordinary conversations about numbers.

The goal of our experiment was epistemological: to verify the adequacy of FOL as a framework for knowledge representation. We were interested in verifying the ability of FOL to "be told" about elementary mathematics. In order to understand the spirit of the experiment it is important to explain our criterion for determining when a reasoning system is adequate. It is *not* simply its ability to carry out proofs that is relevant. In the case of arithmetic, a special purpose theorem prover designed for performing algebraic manipulations can certainly attain the same theorems as our treatment in FOL, sometimes in a faster way (in terms of cpu time). The central issue is how rich a mode of expression is allowed by the system. That is, what facts are implicit by being part of the code of the system and what facts can be explicitly explained to it. It is the *quality* of the conversation that matters. From this viewpoint not many existing theorem provers can be

considered adequate. The treatment of elementary algebra and algebraic manipulations presented in this paper uses modes of expression (namely, meta-theoretic) that are simply not explicitly available to most theorem proving systems.

FOL provides a rich enough conversational facility so that an agreement with the user is made about the subject of the conversation and a domain of consensus is established in which to carry on the reasoning. The first thing to be agreed with FOL is which language to speak, i.e. which tokens we are going to use in our conversation and what syntactic role they will play. FOL then expects to know what facts we assume as basic truths (axioms) of our subject domain. At this point it is ready to do reasoning with us.

Making the above kind of conversation explicit is part of the epistemological flexiblity of FOL. Another aspect of the flexibility of FOL is that it is capable of being told to relativize things to the right context. In fact, it can deal with many theories at the same time. One of them, named META, generally contains meta-theoretic knowledge. It plays a special role because of the way it can communicate with other theories. The possibility of representing meta-theoretic knowledge at the right level, along with the possibility of using it intermixedly with knowledge at the theory level, is a major feature of FOL.

In ordinary math books the distinction between statements at the theory level and those at the meta-theory level is often blurred, if not completely absent. Conversely, if the reasoning system you are dealing with has no capability of explicitly representing meta-theoretic knowledge, many of the statements in elementary math books cannot even be expressed. This might not be interesting if such meta-statements never appeared in practice. On the contrary, they arise *very* often in mathematics books (as well as in ordinary conversation, but this is not the point of the present paper).

In order to have a reasonable conversation with a system about algebra, you want it to have an *understanding* of algebra. In other words, you require that the ability of the system to perform manipulations and answer questions is (at least) as good as yours. We have chosen to bring FOL to this understanding by following a *foundational* approach. It consists in starting with some axioms and incrementally build a theory using in new proofs only facts that are either axioms or have already been proved. Math books are frequently written from a foundational point of view, with the intent of producing an understanding in the reader.

From an epistemological point of view we consider this experiment a success as an examination of our experience with FOL shows that: (1) Proofs expressed to FOL closely resemble the informal proofs of math books, both for their length and for the kind of knowledge they use. (2) Proofs become shorter and shorter along the way, i.e. the more facts you have already proved, the simpler it becomes to prove new ones. (3) What in the book is left as an "easy exercise for the reader", usually is an easy exercise for FOL too - it can be proved with a single line proof.

Our developement of algebra in FOL follows the presentation in [BML65], paying particular attention to consider representing the content of each sentence in the book. We have proved theorems about integers considered both as an integral domain and as an ordered integral domain. We omit the presentation of order and concentrate on the first part. This gives us chances to speak about the use of meta-theoretic knowledge in building proofs, which is one the main purposes of this paper.

The use of meta-theoretic knowledge has been prompted by the observation that in rewrit-

此为试读,需要完整PDF请访问: www.ertongbook.com

ing (i.e. "simplifying") arithmetic expressions we have to deal with the commutativity of the operators plus and times. It is known that commutativity cannot be used as a rewrite rule, since it would cause the rewritings to loop.

This observation has led some authors to deal with equivalences for equational theories involving associative and commutative operators in special ways. For example, finding complete sets of equations that could be used as rewriting rules, or special code to deal with particular cases. The approach we have followed to perform algebraic manipulations on arithmetc expressions is different. The fact that the operators plus and times are commutative, instead of being used at the theory level (by using the relevant axioms), has been embedded into FOL at the meta-theory level. This has been done by specifying what kind of manipulations are allowed on the symbols occurring in an arithmetic expression. Namely, arguments to the functions plus and times can be reordered (for instance to check that `x*y` and `y*x` are the same, hence that `x*y=y*x` holds). As a result, we have devised a "simplification" algorithm that manipulates arithmetic *expressions* by using both theoretical and meta-theoretical knowledge.

The paper is organized as follows. We first explain in detail what has been done at the theory level, and make remarks about proof building within the theory. Then we explain what has been described at the meta-theory level, and finally, how the the simplification algorithm for arithmetic expressions works, in particular, how it uses at the same time knowledge represented in both the theory and the meta-theory.

2. REASONING AT THE THEORY LEVEL

FOL [Wey77,78,79] interacts with the user in a sorted first order language. One of the characterizing features of FOL is that knowledge is represented in the form of *L/S structures* (or *L/S pairs*). These can be explained as a pair of descriptions: a syntactic one (a sorted language and a set of axioms) and a semantic one (a domain of interpretation and information about the interpretation of some of the symbols of the language). The specification of the semantics is done by attaching LISP objects and LISP code to syntactic entities, via what is called *semantic attachment*. The semantic description is also called a *simulation structure*. It functions in FOL as its internal mechanizable analogue of a model for the theory specified by the given syntax. Note that, with an abuse of language, we frequently call an L/S structure a theory.

In order to build a theory for algebra we start by telling FOL that we want to build a new L/S structure, name it `ARITH`, and direct our attention to it. Then, by means of the following declarations:

```
DECLARE SORT INTEGER,NATNUM,NEGNUM;
MG INTEGER ≥ {NATNUM,NEGNUM};
DECLARE INDVAR u v w x y z ∈ INTEGER;
DECLARE OPCONST -(INTEGER)=INTEGER;
DECLARE OPCONST + * - (INTEGER,INTEGER)=INTEGER;
```

we tell FOL that we are going to speak about integers (i.e. `INTEGER` is a sort), natural and negative numbers. We specify that natural and negative numbers are integers, i.e.

the sort INTEGER is more general (MG) then both the sort NATNUM and the sort NEGNUM. We then introduce some individual variables ranging over the integers and some operator constants along with their arity and their domains and ranges.

After the language has been established, we can tell FOL the axioms for integral domains.

```
AXIOM COMMLAW:  ∀ u v.(u+v)=(v+u),
                ∀ u v.(u*v)=(v*u);
       ASSOLAW: ∀ u v w.u+(v+w)=(u+v)+w,
                ∀ u v w.u*(v*w)=(u*v)*w;
       DISTLAW: ∀ u v w.u*(v+w)=(u*v+u*w);
       ZERO:    ∀ u.u+0=u;
       UNITY:   ∀ u.u*1=u;
       NONTRIV: ¬ 0=1;
       ADDINV:  ∀ u.∃ v.u+v=0;
       CANCEL:  ∀ u v w.(¬ u=0 ∧ u*v=u*w ⊃ v=w);;
```

As for the simulation structure, the objects of sort NATNUM are attached to the LISP natural numbers. There is no need for an explicit attachment to the negative numbers because, as an example, the expression "-3" in FOL is interpreted to be the operator unary minus applied to the natural number three. The operators plus and times are attached to the LISP functions PLUS and TIMES, respectively.

As already noted, we adopt the foundational approach, hence no use is made of the operators unary and binary minus in the axioms. After the theorem (named UNINV)

```
∀ u v w.(u+v=0 ∧ u+w=0 ⊃ v=w)
```

stating the unicity of the inverse for the operator plus has been proved, unary minus is introduced by giving FOL the implicit definition (named INVAX)

```
∀ u.u+(-u)=0
```

and attaching the symbol "-" to the LISP function MINUS. We can then introduce the binary minus by the definition

```
∀ x y.(x-y)=(x+-y)
```

In our interaction with FOL, there is no attempt to derive each proof by starting with a minimal set of notions: we introduce and use new notions when they contribute to making proofs more natural. The axiom ADDINV, together with the theorem UNINV may seem to convey the same information as the axiom INVAX, hence they may appear to be of the same use in proof building. Actually this is false: when performing deductions, the axiom INVAX can be used directly but the use of ADDINV involves the application of the deduction rule for existential elimination.

The goal of building proofs that are as natural as possible has been pursued not only by introducing the relevant notions as soon as they are available, but also by exploiting many built-in features of FOL. No attempt has been made to build proofs using the bare logic, i.e., the deductive apparatus of Prawitz's natural deduction alone. We have freely used both the tautology tester and the rewrite/eval commands of FOL. The tautology tester checks whether or not a well formed formula follows as a tautological consequence from a given set of formulas.

The use of the rewrite/eval commands has played a central role in many proofs. The command REWRITE expects a term (or a wff) and a simpset, which is a set of rewrite (simplification) rules and rewrites the term (wff) according to the given simpset. The command EVAL is similar to REWRITE, but it makes use of both the syntactic and the semantic knowledge it is provided with, in the form of a simpset and of semantic attachments, respectively. More details about the workings of REWRITE and EVAL can be found in [We77,78,79;Aie80].

To provide an example of use of REWRITE, a proof of the statement:

$$\forall x.((x*x=x) \supset (x=0 \vee x=1))$$

can be produced starting from the instantiation of the cancellation law on the terms x, x and 1, i.e., $\neg x=0 \wedge x*x=x*1 \supset x=1$. This can be rewritten by a simpset containing $\forall x.x*1=x$ and yields $\neg x=0 \wedge x*x=x \supset x=1$. The theorem follows as a tautological consequence.

The example just presented is very simple. The situation is more complicated when large simpsets and/or many rewritings are involved. This prompted us to construct a "standard" simpset which allowed us to perform most of the standard algebraic manipulations by a single REWRITE, or EVAL.

The following simpset, named SS for Simplification Set, contains either axioms or formulas which were proved using FOL.

```
UNIT   ∀ u.(u*1)=u
       ∀ u.(1*u)=u
NULL   ∀ u.(u+0)=u
       ∀ u.(0+u)=u
       ∀ u.(0*u)=0
       ∀ u.(u*0)=0
INVX   ∀ u.(u+-u)=0
MINS   ∀ u v.(u-v)=(u+-v)
       ∀ u v.-(u+v)=(-u+-v)
       ∀ u.--u=u
        -0=0
       ∀ u v.(u*-v)=-(u*v)
       ∀ u v.(-u*v)=-(u*v)
ASSO   ∀ u v w.((u+v)+w)=(u+(v+w))
       ∀ u v w.((u*v)*w)=(u*(v*w))
DIST   ∀ u v w.(u*(v+w))=((u*v)+(u*w))
       ∀ u v w.((u+v)*w)=((u*w)+(v*w))
```

The evaluation of an arithmetic expression by means of the above simpset (i.e. EVAL ... BY SS;) puts it in the form of a sum of monomials, with no redundant occurrences of zeroes, ones and minus signs, and where a minus sign possibly prefixes some of the monomials.

It should be noticed that the commutativity laws for plus and times do not *explicitly* appear in the above simpset. Hence, whenever commutativity has to be used in a proof, the user has the choice of either instantiating the relevant axiom (which often results

in a pretty long and "unnatural" proof) or better, to use the meta-theoretic knowledge (explained in the following) to further process the arithmetic expression(s) he is dealing with.

3. REASONING AT THE META-THEORY LEVEL

In order to speak with FOL about facts in the meta-theory of arithmetic expressions we have to set up the language (actually, the meta-language) of arithmetic. We start by telling FOL, within the L/S structure named META, that AREX (ARithmetic EXpressions) is a sort and that objects of sort AREX are TERMs. The sort TERM, as well as the sorts INDCONST, OPCONST, etc., appearing in the following are part of the description in META of the language objects (which has been developed by R. Weyhrauch and C. Talcott).

In META we give names to the objects of the theory. We start by introducing individual constants for the operator symbols of the theory, and attaching them to the corresponding objects.

```
DECLARE INDCONST Minus, Sum, Prod ϵ OPCONST;
MATTACH Sum   ⇔  ARITH:OPCONST: + ;
MATTACH Prod  ⇔  ARITH:OPCONST: * ;
MATTACH Minus ⇔  ARITH:OPCONST: - ;
```

The command MATTACH binds names of individual constants in META to objects in a particular theory. In this case the theory is ARITH and the objects are OPCONSTs.

The sort Integer-INDVAR is then introduced in META in order to speak about the individual variables of sort INTEGER in the theory. We specify that objects of sort Integer-INDVAR are particular arithmetic expressions and also particular individual variables. We also give names in META to the variables of sort INTEGER introduced in the theory (i.e. T-u is MATTACHed to u, T-v to v, etc.).

```
DECLARE PREDCONST Integer-INDVAR(AREX);
MG AREX ≥ {Integer-INDVAR};
MG INDVAR ≥ {Integer-INDVAR};
DECLARE INDCONST T-u,T-v,T-w,T-x,T-y,T-z ϵ Integer-INDVAR;
MATTACH T-u ⇔ ARITH:INDVAR: u;
...
```

In order to speak in META about arithmetic expressions in their most general form (i.e. expressions in ARITH that are built out of both individual variables and constants and the operators plus, times and minus, as well as function symbols applied to some arguments), we have to describe in META function application at the theory level. This is done by extending the language in META with the new sort ARGS (ARGumentS).

```
DECLARE SORT ARGS;
MG ARGS ≥ {AREX};
```

We have specified that the sort ARGS is more general than the sort AREX. This implies that an arithmetic expression can be an argument to a function. Conversely, no requirement is imposed on arguments to consist of one or more arithmetic expressions. Hence,

in an arithmetic expression, function symbols can occur that have been declared to map objects of any sort into objects of any other sort, as far as they eventually yield to an arithmetic expression. In other words, an arithmetic expression in ARITH is any term that is hereditarily well sorted and whose sort is INTEGER.

The constructors and selectors needed to handle arguments are also introduced, as well as those needed to handle the application of function symbols to arguments. These constructors and selectors are then attached to the relevant LISP code.

4. COMPUTING IN META

To provide an example of how computations are performed in META and how the relevant information is retrieved from the theory ARITH, whenever needed, let's discuss how the predicate MONOMIAL is defined and how its evaluation goes.

```
AXIOM MONOMIAL: ∀ ae.(MONOMIAL(ae) ≡ INDEL(ae) ∨ ¬(funof(ae)=Sum));;
```

where

```
AXIOM INDEL:  ∀ ae.(INDEL(ae)   ≡  INDSYM(ae) ∨ NEGNUMRAL(ae));;

AXIOM INDSYM: ∀ o.(INDSYM(o)    ≡  INDCONST(o) ∨ INDVAR(o) ∨ INDPAR(o)),
              ∀ o.(INDCONST(o)  ≡  syntype(o)=Indconst),
              ∀ o.(INDVAR(o)    ≡  syntype(o)=Indvar),
              ∀ o.(INDPAR(o)    ≡  syntype(o)=Indpar);;

AXIOM NEGNUM: ∀ ae.(NEGNUMRAL(ae) ≡
                    mainsym(ae)=Minus ∧ NATNUMRAL(arg(1,ae)));;
```

Note that the predicate MONOMIAL is supposed to be applied to arithmetic expressions (aes) that have the form of a sum of monomials (namely, to arithmetic expressions that have already been simplified at the theory level by SS). Hence, to check if an arithmetic expression in this form consists of only one monomial we have to check whether it consists of an individual element or if its function part is not a plus.

Individual elements are defined to be either individual symbols (i.e., individual constants, variables or parameters) or negative numerals. In FOL negative "numerals" aren't individual symbols: they are natural numerals prefixed by the unary minus sign.

To check whether or not an object is an INDCONST (or an INDVAR, or an INDPAR) its syntactic type has to be determined in ARITH. This is done by the function **syntype** which is attached to the following LISP code.

```
ATTACH syntype ⇔  (LAMBDA(X)(SYNT-DN X 'ARITH));
```

The LISP function SYNT-DN applied to X and ARITH switches attention from the L/S structure META to ARITH, computes the syntactic type of X in ARITH, then switches back to META and brings back its result.

As an example, MONOMIAL(T-u) will evaluate to true. This is because **syntype(T-u)** evaluates to Indvar since u has been declared to be an INDVAR in ARITH. On the other hand, if AE has be declared (in META) to be of sort AREX and has been MATTACHed to the term **x+y** (in ARITH), then MONOMIAL(AE) will evaluate to false. Its syntactic type (in

ARITH) is neither INDCONST, nor INDVAR, nor INDPAR, and its function part is the operator plus (i.e., funof(AE)=Sum is true). Note also that the evaluation of funof(AE) involves an L/S structure switching, to check that the main symbol of the expression attached to AE actually is an operator, i.e., its syntactic type in ARITH is either OPCONST or OPVAR.

5. THE SIMPLIFICATION ALGORITHM

Due to space limitations we cannot report the complete set of meta-axioms that implements in FOL the simplification algorithm for arithmetic expressions. We can only sketch it.

In order to simplify an arithmetic expression, it is first evaluated at the theory level by means of the simpset SS. This process, as already noted, puts it in the form of a sum of monomials. Then, the manipulations that are performed on the resulting expression at the meta-theory level consist in a reordering and merging of the monomials constituting it. The reordering of a monomial is performed by matching the variable (and function) symbols occurring in it with a lexicon, which is the list of all the variable and function symbols occurring in it, in the order they first occur. Thus, in META, given an arithmetic expression, its lexicon is built first, then it is used to reorder the monomials and merge them. The choice to build a lexicon for each arithmetic expression in the way just described has been done on the assumption that there is no pre-extablished order among the symbols of an arithmetic expression. On the contrary, the order is intrinsic in the expression itself. In other words, there is no reason for considering the expression y*x "less ordered" then the expression x*y, per se, while the expression x*y+y*x has to be transformed into 2*x*y and not into 2*y*x (i.e., the reordering is to be done according to the lexicon (x y), because this is the order in which the variables x and y occur).

After the reordering and merging is done, some further simplification of the arithmetic expression can still be possible (for instance if a monomial ends up with 0 or 1 as a coefficient). Thus, the arithmetic expression is processed again at the theory level, evaluating it by means of the following simplification set, named SSS (Small Simplification Set).

```
∀ u. (1*u) = u
∀ u. (0*u) = 0
∀ u.(-1*u) =-u
∀ u. (0+u) = u
∀ u. (u+0) = u
```

The behaviour of the entire algorithm that simplifies arithmetic expressions can then be summarized as follows:

At the theory level: evaluate the arithmetic expression by means of the simpset SS.

At the meta-theory level: build the lexicon; simplify all the arguments of functions (if any) that are arithmetic expressions; reorder each monomial according to the lexicon and merge similar monomials.

At the theory level: evaluate the arithmetic expression by means of the simpset SSS.

This entire process is invoked at the meta-theory level, by the following statement.

此为试读,需要完整PDF请访问: www.ertongbook.com