Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

250

TAPSOFT '87

Proceedings of the International Joint Conference on Theory and Practice of Software Development Pisa, Italy, March 1987

Volume 2:
Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications (CFLP)

Edited by Hartmut Ehrig, Robert Kowalski, Giorgio Levi and Ugo Montanari

Springer-Verlag

PREFACE

TAPSOFT '87 is the Second International Joint Conference on Theory and Practice of Software Development.

TAPSOFT '87 is being held from March 23 to March 27, 1987 in Pisa. TAPSOFT '87 has been organized by Dipartimento di Informatica (Università di Pisa), I.E.I. - C.N.R. and CNUCE - C.N.R., and has been supported by AICA and EATCS.

TAPSOFT '87 consists of three parts:

Advanced Seminar on Foundations of Innovative Software Development

New directions in software development have been proposed, on the basis of recent technological and theoretical advances. Following these trends, the software production process should be made more rigorous, and its result should be expressed in a more abstract and understandable form.
The aim of the Advanced Seminar is to bring together leading experts in the various fields which form the foundations of this renovation still in progress and to provide a forum to discuss the possible integration of available theories and methods in view of their applications.
The Advanced Seminar will consist of a number of invited talks, two panel discussions and several working groups. The invited talks will be either long, i.e. comprehensive and general, or short, i.e. dedicated to hot topics.

Invited Speakers

E. Astesiano (Univ. Genova)
K. Clark (Imp. C., London)
K. Furukawa (ICOT, Tokyo)
J. Goguen (SRI, Menlo Park)
G. Huet (INRIA, Paris)
R. Milner (Univ.Edinburgh)
M. Nivat (LITP, Paris)
J. Thatcher (IBM, Yorktown Heights)
D. Warren (Univ. Manchester)

Panels

- On Industrial Activity and Trends. Chairman: J. Goguen (SRI, Menlo Park)
- The Future of Software Engineering. Chairman: D. Bjørner (Lyngby)

The seminar organizers are H. Ehrig (Tech. Univ. Berlin) G. Levi (Univ. Pisa)
R. Kowalski (Imperial College, London) U. Montanari (Univ. Pisa)

Colloquium on Trees in Algebra and Programming

Traditionally, the topics of the Colloquium cover a wider area of theoretical Computer Science than that indicated by the title. Actually, topics include the formal aspects and properties of trees and, more generally, of combinatorial and algebraic structures in all fields of Computer Science.
Besides the customary topics, in keeping with the overall theme of TAPSOFT, the program will include contributions related to specifications, communicating systems and type theory.

The preceding eleven colloquia were held in France and Italy as autonomous conferences, except in Berlin 1985, when for the first time CAAP was integrated into the TAPSOFT Conference.

In keeping with the tradition of CAAP as well as with the overall theme of the TAPSOFT conference, the selected papers are presented in the sections listed below.
- Algorithms
- Proving techniques
- Algebraic specifications
- Concurrency
- Foundations

The program committee for CAAP '87 is the following:

A. Arnold, Bordeaux
J. de Bakker, Amsterdam
B. Buchberger, Linz
J. Diaz, Barcelona
Ph. Flajolet, Paris
H. Ganzinger, Dortmund
P. Mosses, Aarhus
J. Thatcher, Yorktown Heights
M. Wirsing, Passau
G. Ausiello, Roma
A. Bertoni, Milano
M. Dauchet, Lille
H. Ehrig, Berlin
N. Francez, Haifa
U. Montanari, Pisa (Chairman)
M. Nivat, Paris
G. Winskel, Cambridge

Colloquium on Functional and Logic Programming and Specifications

In keeping with the overall theme of the TAPSOFT conferences, CFPL focuses on those aspects of Functional and Logic Programming which are most important in innovative software development. The integration of formal methods and practical aspects of software production is also stressed.

The selected papers are presented in six sessions covering the following topics.

- Theory and Semantics of Functional Languages
- Types, Polymorphism and Abstract Data Type Specifications
- Unification of Functional and Logic Programming Languages
- Program Proving and Transformation
- Language Features and Compilation in Logic Programming
- Implementation Techniques

The Programme Committee for CFLP is the following

C. Böhm, Roma
K. Furukawa, Tokyo
C. Ghezzi, Milano
G. Huet, Paris
R. Kowalski, London
B. Mahr, Berlin
R. Milner, Edinburgh
E. Sandewall, Linköping
D. Warren, Manchester
K. Clark, London
H. Gallaire, München
J. Goguen, Menlo Park
G. Kahn, Sophia Antipolis
G. Levi, Pisa (Chairman)
A. Martelli, Torino
L. Moniz Pereira, Lisboa
E. Shapiro, Rehovot

The TAPSOFT '87 Conference proceedings are published in advance of the conference in two volumes. The first volume includes the final versions of 17 papers from CAAP '87, selected from a total of 51 submitted papers. The second volume includes the final versions of 17 papers from CFLP, selected from a total of 80 submitted papers. Invited papers from the Advanced Seminar are divided between the two volumes.

We would like to extend our sincere thanks to all the Program Committee members as well as to the referees listed below for their care in reviewing and selecting the submitted papers:
J. Alegria, A. Alfons, S. Anderson, J.L. Balcázar, F. Barbic, R. Barbuti, M. Bellia, R. Bird, E. Börger, P.G. Bosco, A. Bossi, G. Boudol, K. Broda, D. Brough, D. Chan, L. Carlucci Aiello, G. Castelli, T. Chikayama, T. Chusho, E. Ciapessoni, N. Cocco, L. Colussi, M. Coppo, T. Coquand, B. Courcelle, G. Cousineau, W. Coy, P.L. Curien, A. Davison, P. Degano, R. De Nicola, M. Dezani, M. Dincbas, M. Ducassé, P. Dufresne, J. Ebert, B. Eggers, P. van Emde Boas, R. Enders, G. Engels, K. Estenfeld, E. Fachini, A. Fantechi, I. Foster, D. Frutos, J. Gabarro, D. Gabbay, F. Galdbay, G. Gambosi, G. Ghelli, P. Giannini, M. Goldwurm, A. Goto, S. Goto, G. Guida, C. Gunter, T. Iato, H. Habel, M. Hagiya, N. Halbwacks, H. Hansen, S. Haqqlund, J. Heering, P. Henderson, R. Hennincker, D. Henry de Villeneuve, C. Hogger, F. Honsell, M. Huntback, H. Hussmann, P. Inverardi, R.C.L. Koymans, L. Kott, H.J. Kreowski, F. Kriwaczek, S. Kunifuji, Y. Lafont, B. Lang, R. Lasas, A.

Laville, P. Le Cheradec, K. Leeb, B. Lennartsson, J.J. Levy, M. Lindqvist, A. Llamosi, G. Lolli, G. Longo, J.A. Makowski, V. Manca, P. Mancarella, D. Mandrioli, M. Manny, A. Marchetti Spaccamela, I. Margaria, M. Martelli, L. Mascoet, Y. Matsumoto, G. Mauri, B.H. Mayoh, F. McCabe, J. Meseguer, J.J.Ch. Meyer, C. Moiso, B. Möller, C. Montangero, K. Moody, A. Mycroft, F. Nicki, M. Nielsen, F. Nielson, F. Nürnberg, M.E. Occhiuto, F.J. Oles, F. Orejas, M. Ornaghi, R. Orsini, P. Padawitz, C. Palamidessi, D. Pedreschi, P. Pepper, A. Pettorossi, A. Poigné, A. Porto, M. Protasi, G. Ringwood, J. Roman, S. Ronchi Della Rocca, G. Rossi, I. Kott, T. Sakurai,D. Sannella, D. Sartini, T. Sato, R. Schuster, M. Sergot, D. Siefkes, M. Smyth, T. Streicher, A. Suarez, Y. Takayama, J. Tanaka, A. Tarlecki, W. Thomas, M. Tofte, S. Tomura, J. Toran, M. Torelli, J.V. Tucker, F. Turini, T. Yuasa, F.W. Vaandrager, B. Vauquelin, B. Venneri, M. Venturini Zilli, H. Wagener, E.G. Wagner, M. Wallace, P. Weis, M. Zacchi, B. Zimmermann

We gratefully acknowledge the financial support provided by the following Institutions and Companies:

- Comune di Pisa
- C.N.R.
  - Presidenza
  - Comitato Nazionale per l'Ingegneria
  - Comitato Nazionale per le Scienze Matematiche
  - CNUCE
  - I.E.I.
- Dipartimento di Informatica, Università di Pisa
- Elsag, Genova
- Enidata, Milano
- IBM Italia, Roma
- List, Pisa
- Olivetti, Ivrea
- Selenia, Roma
- Sipe, Roma
- Systems & Management, Torino
- Tecsiel, Roma
- Università di Pisa

We wish to express our gratitude to the members of the Local Arrangement Committee: P. Asirelli, R. Barbuti, P. Degano (Chairman), A. Fantechi, P. Mancarella, M. Martelli, F. Tarini and F. Turini. Without their help, the Conference would not have been possible.

Pisa, March 1987

Hartmut Ehrig
Institut für Software und Theoretische Informatik
Technische Universität Berlin

Robert A. Kowalski
Dept of Computing and Control
Imperial College London

Giorgio Levi
Dipartimento di Informatica
Università di Pisa

Ugo Montanari
Dipartimento di Informatica
Università di Pisa

CONTENTS OF VOLUME 2

CONTENTS OF VOLUME 1

Models and Equality for Logical Programming[1]

Joseph A. Goguen and José Meseguer
SRI International, Menlo Park CA 94025
Center for the Study of Language and Information, Stanford University 94305

Abstract: We argue that some standard tools from model theory provide a better semantic foundation than the more syntactic and operational approaches usually used in logic programming. In particular, we show how initial models capture the intended semantics of both functional and logic programming, as well as their combination, with existential queries having logical variables (for both functions and relations) in the presence of arbitrary user-defined abstract data types, and with the full power of constraint languages, having any desired built-in (computable) relations and functions, including disequality (the negation of the equality relation) as well as the usual ordering relations on the usual built-in types, such as numbers and strings. These results are based on a new completeness theorem for order-sorted Horn clause logic with equality, plus the use of standard interpretations for fixed sorts, functions and relations. Finally, we define "logical programming," based on the concept of institution, and show how it yields a general framework for discussions of this kind. For example, this viewpoint suggests that the natural way to combine functional and logic programming is simply to combine their logics, getting Horn clause logic with equality.

1 Introduction

This paper argues that some very significant benefits are available to logic programming from using certain concepts from first order model theory, namely:

- order-sorted logic and models;
- initial models;
- interpretation into fixed models for certain fixed sorts, functions and relations; and
- true semantic equality.

These techniques, which are all standard in the theory of abstract data types [17, 22, 14], provide an attractive alternative to the more syntactical and operational approaches generally favored in logic programming. Moreover, they provide a powerful approach that supports:

- user-defined abstract data types;
- built-in data types;
- combined logic and functional programming; and
- constraint-based programming, in a way that can utilize standard algorithms for standard problems, such as linear programming.

In addition, we suggest that the more recent theory of institutions [10] may provide conceptual insight and clarification, as well as a broadening of the general scope of logic programming, so as to encompass any logical system satisfying certain simple restrictions.

In a sense, this paper is an attempt to explicate our previous paper on Eqlog [11], by giving a fuller account of its mathematical semantics, as well as further details, polemics, and comparisons with the

[1]Supported in part by Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450, and a gift from the System Development Foundation.

existing literature. One reason that [11] may have been obscure to many readers, is the large number of new ideas that it tried to introduce all at once; here, we attempt to highlight certain ideas by ignoring others. Among the features of Eqlog deliberately downplayed here are: modules, both hierarchical and generic; theories and views; and "attributes" of operators (e.g., associativity and commutativity). Although these features greatly increase the expressive power of Eqlog, they would also distract from the basic foundational and semantic issues that we wish to emphasize here. For similar reasons, this paper does not develop most issues concerning the operational semantics of the various systems that are discussed. Thus, unification, term rewriting, narrowing and resolution are only touched upon. They are discussed in somewhat more detail in [11], and will receive full treatment in [23] and [26].

1.1 Order-Sorted Logic

Ordinary unsorted logic offers the dubious advantage that anything can be applied to anything; for example,

$$3 * \texttt{first-name(age(false))} < 2^{\texttt{birth-place(temperature(329))}}$$

is a well-formed expression. Although beloved by hackers of Lisp and Prolog, unsorted logic is too permissive. The trouble is that the usual alternative, many-sorted logic, is too restrictive, since it does not support overloading of function symbols such as _+_ for integer, rational, and complex numbers. In addition, an expression like

$$(-4 \;/\; -2)!$$

does not, strictly speaking, parse (assuming that factorial only applies to natural numbers). Here, we suggest that **order-sorted logic**, with subsorts and operator loading, plus the additional twist of **retracts** (although they are not discussed here; see [14]), really does provide sufficient expressiveness, while still banishing the truly meaningless.

Although the specialization of many-sorted logic to many-sorted algebra has been very successfully applied to the theory of abstract data types, many-sorted algebra can produce some very awkward specifications in practice, primarily due to difficulties in handling erroneous expressions, such as dividing by zero in the rationals, or taking the top of an empty stack. In fact there *is no* really satisfactory way to define either the rationals or stacks with MSA. However, order-sorted algebra overcomes these obstacles through its richer type system, which supports subsorts, overloaded operators, and total functions that would otherwise have to be partial. Moreover, order-sorted algebra is the basis of both OBJ [9] and Eqlog [11]. Finally, order-sorted algebra solves the **constructor-selector problem**, which, roughly speaking, is to define inverses, called selectors, for constructors; the solution is to restrict selectors to the largest subsorts where they make sense. For example, **pop** and **top** are only defined for non-empty stacks. [15] shows not only that order-sorted algebra solves this problem, but also that many-sorted algebra *cannot* solve it.

The essence of order-sorted logic is to provide a *subsort* partial ordering among the sorts, and to interpret it semantically as subset inclusion, among the carriers of a model, and to support operator overloading that is interpreted as restricting functions to subsorts. Two happy facts are that order-sorted logic is only slightly more difficult than many-sorted logic, and that essentially all results generalize from the many-sorted to the order-sorted case without complication. See [14] for a comprehensive treatment of order-sorted algebra. This paper broadens the logical framework to allow

not only algebras, but also models of arbitrary first-order signatures, with both function and predicate symbols, including equality, and gives rules of deduction for Horn clauses in such a logic, proving their completeness and several other basic results that are directly relevant to our model-theoretic account of logic and functional programming, including initiality and Herbrand theorems.

1.2 Models

Perhaps the origins in proof theory explain the obsession of logic programming theorists with syntactic and proof theoretic constructions. In any case, we believe that more semantic and more abstract tools provide a basis that is both broader and more powerful. In particular, we feel that the usual Herbrand Universe construction is too syntactic and is also unnecessarily restrictive, because:

1. it does not provide for built-in types, such as numbers and infinite trees;
2. it does not provide for user-defined abstract data types;
3. it does not (directly) address the phenomenon of representation independence for terms and for data types, whether built-in or user-defined; and
4. the proofs are more concrete and computational than necessary[2].

Of course, these deficiencies can all be patched without great difficulty -- for example, [19] shows how to include built-in numbers -- but after a few such patches, you have something enough like the initial model approach that you might as well, or better, take advantage of the powerful machinery associated with that tradition.

The reason for being interested in models is just that a standard model can provide the implementer with a clear standard for correctness, and can also provide the programmer and user with a clear model for what to expect when programs are actually run.

The reason for being interested in standard interpretations into particular semantic domains on some sorts, functions and relations (while leaving others free) is that then one can use standard algorithms to solve particular problems over such domains, for example, linear programming algorithms over the real numbers. This gives a great deal of flexibility, since one can still use initiality (i.e., abstraction) over other sorts. We argue below that this provides an elegant foundation for constraint-based programming.

1.3 Equality

Equational logic, which is essentially the logic of substitution of equals for equals, provides a foundation for functional programming languages. For example: [18] gives (what can be seen as) an equational description of Backus' FP [2]; [24] describes an "equational programming" language[3]; and [9] describes OBJ2, a language that combines initial algebra semantics for executable "objects" (defined by very general sets of user-supplied conditional order-sorted equations), with "loose" algebra semantics for non-executable "theories" (defined by arbitrary sets of equations).

---

[2] Not everyone will regard this as a deficiency!

[3] This language has some very strong restrictions, including: no repeated variables on lefthand sides, no overlap among equations, only one sort of data, no conditional equations, and a strong sequentiality condition; on the other hand, it is much easier to compile efficient code from sets of equations that satisfy such restrictions.

In the context of first order logic, equality is generally treated as a special relation, interpreted as real semantic equality in models, rather than merely axiomatized. This is the sense in which one speaks of "first order logic with equality" and of "Horn clause logic with equality." Complete sets of rules of deduction are well-known for these logical systems, and the latter has been used to combine logic and functional programming [11]. This paper later gives corresponding rules for order-sorted Horn clause logic with equality.

Equality is also useful in understanding constraint-based programming, because equations can be used to define the basic data structures, and then various relations of special interest can be defined recursively over these data structures, and/or provided as built-ins.

1.4 Initiality

Initial models free one from commitment to any particular representation; that is, they support *abstraction*. In particular, initiality handles abstract data types for logical programming languages with great fluency and convenience, and similarly it can be used to define functions and relations over built-in types [11]. Initial models also provide an account of the conceptual world of a program, in the sense of being "closed worlds" or "standard models." In particular, they provide a standard of correctness for the implementer, as well as a model for what results to expect for the programmer. Finally, initiality is a so-called "universal property," that there exists a unique mapping satisfying certain conditions, and it is well-known that, in many cases, one gets a much cleaner mathematical theory, with simpler and more conceptual proofs, from using universal characterizations of objects of interest, as compared to using concrete constructions for them [21]. In fact, the familiar characterization of "free" by the existence of a unique mapping with certain properties that extends another, is a special case of initiality.

One can better understand initiality through the so-called "no junk" and "no confusion" conditions (originally from [7]); these can also be seen as "completeness" and "soundness" conditions, respectively. Assume that signatures provide symbols for constructing sentences, including functions and constants (in Σ) and relations (in Π), and that models contain "data elements." Given a signature Σ,Π and a set $\mathcal{C}$ of Σ,Π-sentences, call a Σ,Π-model **standard** if and only if:

1. **No junk**: Every data item is denoted by a term using the function (and constant) symbols in Σ. (A data item that cannot be so constructed is "junk.")
2. **No confusion**: a predicate holds of some data elements if and only if it can be proved from the given sentences; in particular, two elements are identified if and only if they can be proved equal from the given sentences. (Two data items that are equal but cannot be proved so are "confused.")

For Horn clause logic, either with or without equality, either order-sorted, many-sorted, or unsorted, these two conditions define the data items *uniquely* up to renaming, i.e., they define a model up to isomorphism. Moreover, "no junk" is equivalent to structural induction over the signature, and the two conditions together are equivalent to the "unique homomorphism" condition called **initiality** (see [22] for details).

1.5 Constraints

In its general sense, a **constraint** is a logical relation that one wishes to impose on a set of potential solutions. In principle, such constraints could be arbitrary first order sentences involving arbitrary (interpreted and uninterpreted) relations; but in practice, constraints are limited to sets of atomic sentences, such as

$a * X + b * Y < c * Z + d$,
$a * X * X + b * X + c = 0$,
$a * X * Y \cup b * X + c \subseteq Z$,

where the variables in the first two constraints range over some kind of number (e.g., integers, or rationals, or complexes), and in the second range over sets of strings from some fixed alphabet ($*$ is multiplication in the first two, and is concatenation, extended to sets, in the third). Although Prolog would, in principle, be ideal for *constraint-based programming*, it does not suffice in practice, because of the limited capabilities of the built-in relations. Moreover, the usual semantic basis of Prolog does not extend to built-ins without some extra fuss and awkwardness (e.g., as in [19]).

We refer to sorts, functions, and relations upon which interpretation into a fixed (standard) model are imposed as **built-ins**. Two obvious examples of such models are numbers and infinite trees, with their usual functions and relations. The pioneering work of Jaffar and Lassez [19] and of Jaffar and Michaylov [20] treat these and a number of other examples, in the context of a constraint logic programming language called CLP. These authors also treat a number of other topics, some of which are not considered here, including negation as failure and compactness conditions [19].

1.6 Logical Programming

Various aspects of programming languages are captured by various aspects of logic. The functional aspect of programming is captured by equational logic [9]. Strong typing is captured by many-sorted logic. Logic programming (which might be less misleadingly called relational or Horn clause programming) is captured by Horn clause logic. Object-oriented programming is captured by reflective logic, in which there is an abstract data type of program texts built into the language [12]. The perspective of logical programming suggests that the right way to combine various programming paradigms is to discover their underlying logics, combine them, and then base a language upon the combined logic. This permits one to mix and match various programming language features. For example, combined functional and logic programming is captured by Horn clause logic with equality [11]. Combined functional and object-oriented programming is captured by reflective equational logic (we call this language FOOPS, see [12]). We currently feel that reflective order-sorted Horn clause logic with equality is a good candidate for unifying the functional, relational and object-oriented paradigms into a single simple programming language which also has powerful database capabilities.

The theory of institutions [10] can provide a formal basis for the notion of logical programming. Informally, an institution is a logical system, with formal notions of sentence, model, and satisfaction. Then, a **logical programming language** $\mathcal{L}$ has an associated logical system (i.e., institution) I such that:

- the statements of $\mathcal{L}$ are sentences from I;
- the operational semantics of $\mathcal{L}$ is (a reasonably efficient) deduction in I; and
- the denotational semantics of $\mathcal{L}$ is given by a class of models in I (preferably initial models).