G. Comyn N. E. Fuchs
M. J. Ratcliffe (Eds.)

Logic Programming in Action

Cco1-636

Second International Logic Programming
Summer School, LPSS '92
Zurich, Switzerland, September 7-11, 1992
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editor

Jörg Siekmann
University of Saarland
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, W-6600 Saarbrücken 11, FRG

Volume Editors

Gérard Comyn
Michael J. Ratcliffe
European Computer-Industry Research Centre (ECRC)
Arabellastr. 17, W-8000 Munich 81, FRG

Norbert E. Fuchs
University of Zurich-Irchel, Computer Science Institute
Winterthurerstr. 190, Ch-8057 Zurich, Switzerland

CR Subject Classification (1991): I.2, D.1.6, F.4.1, D.2.1, J.1, J.6

[illegible]

ISBN 3-540-55930-2 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55930-2 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

While the First Logic Programming Summer School, *LPSS '90*, addressed the theoretical foundations of logic programming, the Second Logic Programming Summer School, *LPSS '92*, focuses on the relationship between theory and practice, and on practical applications.

Logic programming enjoys a privileged position. On the one side, it is firmly rooted in mathematical logic, on the other side it is immensely practical as a growing number of users in universities, research institutes and industry are realising. Logic programming languages, specifically Prolog, have turned out to be ideal as prototyping and application development languages. Often, one defines an application-specific language that can be translated into a logic language. In this case, logic programming not only helps to conveniently define the syntax of the application-specific language, but also to express its semantics in a direct and understandable way.

There is an interplay between the theory and practice of logic programming that has been essential for its progress. In the introduction to this volume Robert Kowalski - one of the pioneers of the field - addresses this interplay, and identifies a number of problems where further research will be necessary to improve the relation between theory and practice. Much of this research is being done in the framework of the Basic Research Project *Compulog*, and the Network of Excellence in Computational Logic *Compulog-Net* of the European Community's ESPRIT program.

The interplay between theory and practice is also reflected in the relationships between logic programming and other fields of computer science, e.g. deductive databases, knowledge-based systems, computational linguistics, and software engineering. On the one side, these fields have borrowed concepts and methods from logic programming, while on the other they have strongly influenced its research directions. This has led to a strong synergy. To name only two examples, Prolog was originally developed for writing natural language processing applications, while knowledge-based systems continue to profit from the powerful metaprogramming techniques provided by logic programming.

The contributions contained in this volume fall into two categories: tutorials and project presentations. Four tutorials provide an overview of the relation of logic programming to constraint logic programming, deductive databases, language processing and software engineering as well as some theoretical background. Each topic is expanded by project presentations which give detailed accounts of existing applications, some of which are in the prototype stage, while others are in daily use.

In their tutorial, *Constraint Logic Programming - An Informal Introduction*, ECRC's *CORE* team give an insight into constraint logic programming which is a relatively new but rapidly expanding subfield of logic programming. Constraint logic programming (CLP) combines the power of logic programming languages with efficient constraint solving methods. CLP has proved to be extremely useful for scheduling, planning, and optimisation problems. This is borne out by the following project presentations. Michel d'Andrea, in his contribution *Scheduling and Optimisation in the Automobile Industry,* describes a prototype for a job scheduling system developed by Bull for the Renault group. In *Factory Scheduling using Finite Domains,* Owen Evans shows the advantages that constraint logic programming offers for solving problems in a factory environment using the DecisionPower system sold by ICL. Finally, Pierre-Joseph Gailly and his colleagues report on the ESPRIT *Prince Project and Its Applications* in which a practical constraint solving system based on the logic programming language Prolog III is being developed.

Logic programming has always had strong relations with deductive databases and expert systems. In his tutorial, *A (Gentle) Introduction to Deductive Databases,* Shalom Tsur recalls how the limitations and weaknesses of relational databases, especially of relational query languages, led to the ideas of deductive databases, and points out the many interconnections to logic programming. Christoph Beierle presents *Knowledge Based PPS Applications in PROTOS-L* which shows how an enhanced Prolog developed in the context of the Protos Eureka project can be used for Knowledge-Based scheduling applications. Carlo Chiopris describes the development of the *SECReTS Banking Expert System from Phase 1 to Phase 2;* the application is being used by several Italian banks for the analysis of client data. In his contribution, *Logic Engineering and Clinical Dilemmas,* John Fox, who works at the Imperial Cancer Research Fund, focuses on the advantages of logic programming for clinical decision making, while Edward Freeman shows how *A Knowledge-Based Approach to Strategic Planning* helps corporations define their strategic directions based on models that relate critical business factors to business targets. In cooperation with the German mining industry, Lutz Plümer developed two *Expert Systems in Mining,* that are near practical applications: *Schikorre* helps to locate geological seams, while *BUT* solves the planning problem for underground illumination.

As mentioned above, the processing of natural language led to the development of Prolog, i.e. logic programming and language processing have been related from the very beginning. In his tutorial *Natural and Formal Language Processing,* Michael Hess identifies machine translation, interaction with computers in natural languages, and accessing information in natural language as three main goals of natural language processing, and shows how logic programming continues to contribute to achieve these goals. In her project presentation, Deborah Dahl introduces *Pundit - Natural Language Interfaces.* To be domain independent, Pundit consists of a number of modules that separately perform the tasks of syntactic, semantic and pragmatic analysis. The *ESTEAM-316 Dialogue Manager* presented by Thomas Grossi, Didier Bronisy and François Jean-Marie model a part of human dialogue, viz. advice giving in the domain of financial investments. Robert Kowalski points out the syntactic similarities of legal language and logic programming languages, and shows how the formalisation of *Legislation as Logic Programs* suggests ways in which logic programming could be extended. Knowledge representation is essential in natural language processing. Udo Pletat presents in *Knowledge Representation for Natural Language Processing* the knowledge representation formalism L$_{\text{LILOG}}$ which has the power of first-order predicate logic and offers a type system similar to the one in KL-ONE. Language processing is not restricted to natural language alone. Peter Reintjes has developed *A Set of Tools for VHDL Design* which convincingly demonstrates Prolog's strength as an implementation language for language-oriented work in general, and hardware description languages in particular.

Software engineering is another field that profits enormously from the power and conciseness of logic programming languages. Keywords that come immediately to mind are executable specifications, program synthesis and program transformations. Based on the great experience of his many years in the field, Alan Bundy shows how reasoning about logic programs helps to improve the efficiency and the reliability of programs. In his paper, *Tutorial Notes: Reasoning About Logic Programs,* he presents a unified view that encompasses the problems of verification, termination, synthesis, transformation, and abstraction. Formal specifications in logic programming languages are the topic of Abdel Ali Ed-Dbali and Pierre Deransart. In their contribution *Software Formal Specification by Logic Programming: The Example of Standard Prolog* they use as a concrete example the formal specification of the language Prolog itself. This work is part of the emerging international Prolog standard. One of the largest problems facing software engineering is the mass of existing programs, many of them badly or not at all documented. Peter Breuer presents a set of tools demonstrating *The Art of Computer Un-Programming: Reverse Engineering in Prolog* These tools were developed with the goal of improving the comprehensibility and maintainability of existing COBOL programs. A variety of methods for debugging have been suggested in the logic programming community. Though extremely powerful, these methods are not necessarily practical. Mireille Ducassé describes *OPIUM - An Advanced Debugging System* that is based on traces of program executions and combines the power of logic programming with great practicality.

Strangely enough, teaching is not normally considered as an application field though its importance cannot be underestimated. In the framework of the Swiss National Research Project NFP 23, Fabio Baj and Mike Rosner have developed *Automatic Theorem Proving within the Portable AI Lab.* This theorem proving tool helps to teach basic and advanced topics of logic and logic programming.

Producing the contributions for this volume has involved a lot of time and expense. Several organisations have generously supported us in this and we gratefully acknowledge their contributions:

Bull
Commission of the European Communities
European Computer-Industry Research Centre (ECRC)
ICL
Industrade AG (Apple Computer Division)
University of Zurich

The papers appearing in this volume demonstrate convincingly that logic programming fruitfully combines theory and practice. Realistic applications have already been successfully constructed using logic programming languages. We hope this volume will provide inspiration for others in the future.

July 1992

Gérard Comyn
Norbert E. Fuchs
Michael Ratcliffe

Contents

Theory and Practice in Logic Programming

Robert Kowalski

Department of Computing, Imperial College
London, England, U.K.
April 1992

Abstract. Logic Programming enjoys a relatively good relationship between its theory and its practice. Nonetheless, this relationship needs to be improved, and doing so is an important direction for research in the future. The European Community Basic Research Project, Compulog, and the more general "network of excellence", Compulog-net, are concerned with developing such improvements.

1 Procedural versus Declarative Interpretations

The procedural interpretations of Horn clauses and of negation as failure are the basis for both the theory and practice of logic programming. For many applications (e.g. databases and program specifications) the declarative view needs to dominate the procedural. For other applications, the procedural is more important. In many cases, both views are necessary and a smooth progression and interrelationship between the two is necessary. Achieving a harmonious balance is not always as easy in practice as it should be in theory.

Two areas where future research would be useful are improving the data structures and improving the link with object-orientation. Array-like data structures supporting destructive assignment are convenient in practice. At present the theory allows recursive data structures and various approximations of arrays. Sets of clauses, viewed as updatable databases, are a promising alternative.

Many suggestions have been proposed for combining logic programming with object-orientation. In some of these proposals objects are interpreted as terms; in others as predicates; in still others as "theories" or sets of clauses. All of these proposals and their relationships need to be investigated further.

2 Metaprogramming

Programs which manipulate other programs (or sets of clauses) are an important logic programming technique, used for such applications as providing metadata, implementing metainterpreters, programming in the large, and distributed intelligent systems. The Gödel logic programming language is currently under development in Compulog, motivated to a large extent by the goal of providing improved metaprogramming facilities. Additional work is necessary to reconcile the metalogical techniques which have proved useful in practice with the foundations that are needed in theory.

3 Negation as failure

Non-monotonic reasoning is necessary for many applications, including temporal reasoning in artificial intelligence and database systems. In recent years it has been recognised that negation as failure in logic programming provides a practically effective and theoretically sound technique for non-monotonic reasoning. Further research is necessary to understand better the relationships between different semantics for negation as failure and to develop appropriate extensions for disjunctive reasoning, integrity constraints, and the combination of explicit negation and implicit negation as failure.

4 Abduction

Very recently the extension of logic programming to include abductive (hypothetical) reasoning has begun to be investigated. This extension is related to other extensions such as constraint logic programming and conditional answers. It can also be used for non-monotonic reasoning and negation as failure. Further work is needed to relate better its semantics (viewed as a program specification) with its implementation.

5 Program optimisation

One of the main purposes of semantics is to provide a foundation for proving program equivalence and to justify program transformations and optimisations. Such transformation and optimisation can make a major contribution to improving programmer productivity. A number of powerful optimisation methods have been investigated. Much more can be done to put the theory into practice.

6 Wider implications

Logic programming, appropriately extended (e.g. with explicit negation, disjuction, abduction), begins to achieve the expressiveness of a complete, symbolic knowledge representation formalism. It has proved especially promising for formalising legal reasoning. This application is important, both because legal reasoning can be regarded as prototypical of practical reasoning in general, and because rule-based legal reasoning integrates naturally and comfortably with other kinds of reasoning, including case-based reasoning with open-textured concepts. The strong links between logic programming and legal reasoning provide evidence that logic programming may one day prove as useful for computing as legal reasoning is for human affairs. More importantly, it may help us better to achieve the goals of human logic itself: to reason more clearly and effectively as human beings, even without the use of computers.

Related Reading

1. C. Hogger, R. Kowalski: Logic Programming. In Encyclopedia of Artificial Intelligence (ed. S. Shapiro), (second edition, 1992) Vol. 1 (A-L), pp. 873-891

2. R. Kowalski: Problems and Promises of Computational Logic. In Proceedings Symposium Computational Logic (ed. J. Lloyd), Springer-Verlag 1990, pp. 1-36

Constraint Logic Programming - An Informal Introduction[*]

Thom Frühwirth, Alexander Herold, Volker Küchenhoff,
Thierry Le Provost, Pierre Lim, Eric Monfroy, Mark Wallace

ECRC
European Computer-Industry Research Centre
Arabellastr. 17, D-8000 Munich 81, Germany

email: {thom, herold, volker, thierry, pierre, eric, mark}@ecrc.de

Abstract. Constraint Logic Programming (CLP) is a new class of programming languages combining the declarativity of logic programming with the efficiency of constraint solving. New application areas, amongst them many different classes of combinatorial search problems such as scheduling, planning or resource allocation can now be solved, which were intractable for logic programming so far. The most important advantage that these languages offer is the short development time while exhibiting an efficiency comparable to imperative languages. This tutorial aims at presenting the principles and concepts underlying these languages and explaining them by examples. The objective of this paper is not to give a technical survey of the current state of art in research on CLP, but rather to give a tutorial introduction and to convey the basic philosophy that is behind the different ideas in CLP. It will discuss the currently most successful computation domains and provide an overview on the different consistency techniques used in CLP and its implementations.

1 Introduction

During the last decade a new programming paradigm called *"logic programming"* has emerged. The best known representative of this new class of programming languages is *Prolog*, originated from ideas of Colmerauer in Marseille and Kowalski in Edinburgh. Programming in Prolog differs from conventional programming both stylistically and computationally, as it uses logic to declaratively state problems and deduction to solve them.

It has been argued in the literature [Kow79, Ste80] that a program is best divided into two components called *competence* and *performance* or *logic* and *control*. The competence component describes factual information - statements of relationships - which must be manipulated and combined to compute the desired result. The performance component deals with the strategy and control of the manipulations and combinations. The competence part is responsible for the correctness of the program; the performance part is responsible for the efficiency. An ideal programming

[*] This work is partially funded by the ESPRIT project CHIC, Nr. 5291

methodology would first be concerned with the competence (*"what"*), and only then, if at all, worry about the performance (*"how"*). Logic programming provides a means for separation of these concerns. It is based on ***first order predicate logic***, and the performance component is mostly automatic by relying on a built-in computation mechanism called *SLD-resolution*.

In this way, logic programming has the unique property that its ***semantics***, operational and declarative, are both simple and elegant and coincide in a natural way. These semantics, however, have their limitations. Firstly the objects manipulated by a logic program are uninterpreted structures - the set of all possible terms that can be formed from the functions and constants in a given program. Equality only holds between those objects which are syntactically identical. Every semantic object has to be ***explicitly*** coded into a term; this enforces reasoning at a primitive level. ***Constraints*** on the other hand are used to ***implicitly*** describe the relationship between such semantic objects. These objects are often ranging over such rich computation domains, as integers, rationals or reals.

The second problem related to logic programming stems from its uniform but simple computation rule, a depth-first search procedure, resulting in a *generate and test* procedure with its well-known performance problems for large search applications. Constraint manipulation and propagation have been studied in the Artificial Intelligence community in the late 1970s and early 1980s [Mon74, Ste80, Mac86] to make search procedures more intelligent. Techniques like local value propagation, data driven computation, forward checking (to prune the search space) and look ahead have been developed for solving constraints. These techniques can be summarised under the heading *"Consistency Techniques"*.

Constraint Logic Programming (CLP) is an attempt to overcome the difficulties of logic programming by enhancing a Prolog-like language with constraint solving mechanisms. Curiously both of these limitations of logic programming can be lifted using "constraints". However, each limitation is treated by a quite different notion of constraint. CLP has hence two complementary lines of descent.

Firstly it descended from work that aimed at introducing richer data structures to a logic programming system thus allowing semantic objects, e.g. arithmetic expressions, directly to be expressed and manipulated. The core idea here is to replace the computational heart of a logic programming system, unification, by constraint handling in a constraint domain. This scheme, called CLP(X), has been laid out in the seminal paper of Jaffar & Lassez [JL87]. X has been instantiated with several so called computation domains, e.g. reals in CLP($\mathcal{R}$), rationals in CLP($\mathcal{Q}$), and integers in CLP($\mathcal{Z}$).

Secondly CLP has been strongly influenced by the work on consistency techniques. With the objective of improving the search behaviour of a logic programming system Gallaire [Gal85] advocated the use of these techniques in logic programming. He proposed the active use of constraints, pruning the search tree in an a priori way rather than using constraints as passive tests leading to a "generate and test" or "standard backtracking" behaviour. Subsequently the different inference mechanisms underlying the finite domain part of the CLP system CHIP [DVS+88] were developed. The key aspect is the tight integration between a deterministic process, constraint evaluation, and a nondeterministic process, search. It is this active view of constraints which is exploited in CHIP to overcome the well-known performance problems of "generate and test". This new paradigm exhibits a data-driven computation and can be characterised as *"constrain and generate"*.

Constraint solving has been used in many different application areas such as engineering, planning or graphics. Problems like scheduling, allocation, layout, fault diagnosis and hardware design are typical examples of constrained search problems. The most common approach for solving constrained search problems consists in writing a specialised program in a procedural language. This approach requires substantial effort for program development, and the resulting programs are hard to maintain, modify and extend. With CLP systems a large number of constrained search problems have been solved, some of them were previously solved with conventional languages. CLP languages dramatically reduce the development time while achieving a similar efficiency. The resulting programs are shorter and more declarative and hence easier to maintain, modify or extend. The wealth of applications shows the flexibility of CLP to adapt to different problem areas. Many Operations Research problems have been solved with the CLP system CHIP [DVS+88, Van88, DSV90]. Another very promising application domain is circuit design [Sim92, FSTW91]. Extensive work has also been devoted to financial applications [Ber89, LMY87]. More recently applications in user interfaces [HHLM91] and in databases [KKR90] have been studied. As the subsequent tutorial in this summer school focusses on industrial applications of CLP, we will not further discuss them in this article.

The aim of this informal tutorial is to present the most prominent ideas and concepts underlying CLP languages. It is not intended to present the underlying theory of this new class of programming languages or to give an overview on the current state of art in CLP research. There are already technical surveys in the literature, giving more details on those aspects. In particular the article of [Van91] is worth reading. A restricted view is presented in [Coh90, Frü90] discussing work around the CLP scheme. For the usage of "consistency techniques" in CLP, [Van89] is a valuable source going from theory to application with a large number of programming example.

This tutorial is organised as follows: In the next section we will introduce the CLP scheme and review the most important computation domains that have been developed so far, linear and non-linear arithmetic and boolean constraints. Then we will introduce the concept of finite domains, consistency techniques and their extension to arbitrary domains. Next we will explore ways of extending and tuning constraint systems. Then the work on search and optimisation in CLP will be presented. Finally current CLP implementations will be reviewed, amongst them the most well-known systems: CHIP [DVS+88], CLP($\mathcal{R}$) [JMSY90] and Prolog III [Col90].

2 The CLP Scheme

In this section we will introduce in an informal way the basics of the Constraint Logic Programming Scheme (called CLP(X)), as developed by Jaffar and Lassez [JL87]. The key aspect in the CLP scheme is to provide the user with more expressiveness and flexibility concerning the primitive objects the language can manipulate. Clearly the user wants to design his application using concepts that are as close as possible to his domain of discourse, e.g. he wants to use sets, boolean expressions, integers,