M. Bruynooghe M. Wirsing (Eds.)

CC01-631

Programming Language Implementation and Logic Programming

4th International Symposium, PLILP '92
Leuven, Belgium, August 26-28, 1992
Proceedings

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

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
W-7500 Karlsruhe, FRG

Juris Hartmanis
Department of Computer Science
Cornell University
5149 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Maurice Bruynooghe
Catholic University of Leuven, Dept. of Computer Science
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium

Martin Wirsing
University of Munich, Institute of Computer Science
Leopoldstr. 11 B, W-8000 Munich 40, FRG

CR Subject Classification (1991): F.4.1-2, D.3.1, D.3.4, F.3.3, I.2.3

ISBN 3-540-55844-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55844-6 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

This volume contains the papers which have been accepted for presentation at the Fourth International Symposium on Programming Language Implementation and Logic Programming (PLILP'92) held in Leuven, Belgium, August 26–28, 1992. The Symposium was preceded by three meetings which took place in Orléans, France May 16–18, 1988, in Linköping, Sweden, August 20–22, 1990 and in Passau, Germany, August 26–28, 1991 (their proceedings were published by Springer-Verlag as Lecture Notes in Computer Science, volumes 348, 456, and 528 respectively).

The aim of the Symposium was to explore new declarative concepts, methods and techniques relevant for implementation of all kinds of programming languages, whether algorithmic or declarative. The intention was to gather researchers from the fields of algorithmic programming languages as well as logic, functional and object-oriented programming.

In response to the call for papers, 82 papers were submitted. The Program Committee met in Leuven on April 27 and selected 29 papers, chosen on the basis of their scientific quality and relevance to the Symposium. At the Symposium, two invited talks were given by Michael Hanus and Patrick Cousot. Several software systems were presented, showing new developments in the implementation of programming languages and logic programming.

This volume contains the two invited presentations, the selected papers and abstracts of the system demonstrations.

On behalf of the Program Committee the Program Chairmen would like to thank all those who submitted papers and the referees who helped to evaluate the papers.

The support of

Association for Logic Programming,
Belgian National Fund for Scientific Research,
BIM,
Katholieke Universiteit Leuven

is gratefully acknowledged. Bart Demoen, Brigitte Gelders, Gerda Janssens, Baudouin Le Charlier, Bern Martens, Anne Mulkers and several other members of the department provided invaluable help throughout the preparation and organization of the Symposium. We also would like to thank Springer-Verlag for their excellent cooperation concerning the publication of this volume.

June 1992
Leuven
München

Maurice Bruynooghe
Martin Wirsing

Conference Chairmen

Maurice Bruynooghe, K.U. Leuven (Belgium)
Martin Wirsing, Univ. of München (Germany)

Program Committee

Maurice Bruynooghe, K.U. Leuven (Belgium)
John Darlington, Imperial College, London (UK)
Saumya Debray, Univ. of Arizona, Tucson (USA)
Wlodek Drabent, Linköping Univ. (Sweden) and Warsaw Univ. (Poland)
Gérard Ferrand, Université d' Orléans (France)
Stefan Jähnichen, TU Berlin (Germany)
Bharat Jayaraman, State Univ. of New York, Buffalo (USA)
Claude Kirchner, INRIA Lorraine & CRIN, Nancy (France)
Feliks Kluźniak, Warsaw Univ. (Poland)
Heikki Mannila, Univ. of Helsinki (Finland)
Torben Mogensen, Univ. of Copenhagen (Denmark)
Alan Mycroft, Cambridge (UK)
Lee Naish, Univ. of Melbourne (Australia)
Jaan Penjam, Estonian Academy of Science, Tallinn (Estonia)
Jiro Tanaka, Fujitsi Laboratories, Tokyo (Japan)
Franco Turini, Universita di Pisa (Italy)
Andrei Voronkov, Novosibirsk (Russia) and ECRC, München (Germany)
Reinhard Wilhelm, Univ. des Saarlandes, Saarbrücken (Germany)
Martin Wirsing, Univ. of München (Germany)

Organizing Committee

Maurice Bruynooghe, K.U. Leuven
Bart Demoen, K.U. Leuven
Gerda Janssens, K.U. Leuven (organizing chairman)
Baudouin Le Charlier, F.U.N.D.P. (Namur)
Bern Martens, K.U. Leuven
Anne Mulkers, K.U. Leuven

List of Referees

Many other referees helped the Program Committee in evaluating papers. Their assistance is gratefully acknowledged.

M. Alt
N. Andersen
J.-M. Andreoli
M. Anlauff
F. Baiardi
A. Bansal
R. Barbuti
F. Barthélemy
R.N. Bol
S. Bonnier
D. Boulanger
A. Brogi
M.V. Cengarle
M.M.T. Chakravarty
M. Codish
P. Codognet
P. Dague
B. Demoen
A. De Niel
P. Deransart
D. De Schreye
R. Dietrich
E. Domenjoud
M. Dorochevsky
H. Emmelmann
C. Fecht
C. Ferdinand
U. Fraus
L. Fribourg
M. Fujita
S. Gastinger
M. Gengenbach
U. Geske
R. Giacobazzi
J. Grosch
Y. Guo
G. Gupta
J. Hannan
F. Henglein
A.V. Hense
L. Hermosilla
A. Herold
W. Hesse
K. Hirano
K. Hirata
J.-M. Hufflen
J. Hughes
H. Hußmann
N. Ichiyoshi
J.-P. Jacquot
D. Jana
G. Janssens
P. Kilpeläinen
H. Kirchner
F. Klay
E. Klein
M. Koshimura
U. Lechner
H.C.R. Lock
M. Maeda
J. Małuszyński
P. Mancarella
R. Manthey
L. Maranget
B. Martens
M. Meier
A. Mück
F. Nickl
U. Nilsson
A. Ohsuga
J. Paakki
C. Palamidessi
D. Parigot
R. Paterson
D. Pedreschi
H. Perkmann
H. Peterreins
S. Prestwich
M. Raber
I. Ramakrisnan
B. Reus
O. Ridoux
M. Rittri
H. Rohtla
K.H. Rose
G. Sander
P.-Y. Schoebbens
K. Sieber
M. Simons
R. Stabl
H. Sugano
A. Takeuchi
T. Tammet
H. Tsuda
H. Ueda
E. Ukkonen
P. Van Hentenryck
M. Vittek
M. Weber

Table of Contents

Improving Control of Logic Programs by Using Functional Logic Languages

Michael Hanus

Max-Planck-Institut für Informatik
Im Stadtwald
W-6600 Saarbrücken, Germany
e-mail: michael@mpi-sb.mpg.de

Abstract. This paper shows the advantages of amalgamating functional and logic programming languages. In comparison with pure functional languages, an amalgamated functional logic language has more expressive power. In comparison with pure logic languages, functional logic languages have a better control behaviour. The latter will be shown by presenting methods to translate logic programs into a functional logic language with a narrowing/rewriting semantics. The translated programs produce the same set of answers and have at least the same efficiency as the original programs. But in many cases the control behaviour of the translated programs is improved. This requires the addition of further knowledge to the programs. We discuss methods for this and show the gain in efficiency by means of several examples.

1 Introduction

Many proposals have been made to integrate functional and logic programming languages during the last years (see [3, 11] for surveys). Recently, these proposals became relevant for practical applications because efficient implementations have been developed [5, 8, 19, 33, 35, 48]. This raises the natural question for the advantages of such amalgamated languages. In comparison with pure functional languages, functional logic languages have more expressive power due to the availability of features like function inversion, partial data structures and logic variables [42]. In comparison with pure logic languages, functional logic languages allow to specify functional dependencies and to use nested functional expressions. Although this improves the readability of logic programs, it is not clear whether this is only a minor syntactic improvement (which can be added to logic languages by a simple preprocessor [37]) or there is a genuine advantage of functional logic languages compared to pure logic languages. In this paper we show that the latter is true: functional logic languages have a better operational behaviour than logic languages. We show this by presenting methods to translate logic programs into a functional logic language. These methods ensure that the translated programs produce the same set of answers and have at least the same efficiency as the original programs. But in many cases the translation improves the control behaviour of logic programs which will be demonstrated by several examples.

```
sort(L,M) :- perm(L,M), ord(M).

perm([],[]).
perm([E|L],[F|M]) :- del(F,[E|L],N), perm(N,M).

del(E,[E|L],L).
del(E,[F|L],[F|M]) :- del(E,L,M).

ord([]).
ord([E]).
ord([E,F|L]) :- le(E,F), ord([F|L]).

le(0,E).
le(s(E),s(F)) :- le(E,F).
```

Figure 1. Permutation sort (natural numbers are represented by s-terms)

Logic programming allows the specification of problems at an abstract level and permits the execution of the specifications. However, these specifications are often very slowly executed because a lot of search is performed under the standard Prolog computation rule. For instance, Figure 1 specifies the notion of a sorted list (cf. [44], p. 55): a list `M` is a sorted version of a list `L` if `M` is a permutation of `L` and all elements of `M` are in ascending order. We can use this Prolog program to sort the list `[4,3,2,1]` by solving the query `?- sort([4,3,2,1],S)`. But this runs very inefficiently under the standard computation rule because all permutations must be enumerated and tested in order to solve this goal.

Therefore several proposals have been made in order to improve the control of Prolog programs. Naish [36] has extended the standard computation model of Prolog by a coroutining mechanism. He allows the addition of "wait" declarations to predicates. Such declarations have the effect that the resolution of a literal is delayed until the arguments are sufficiently instantiated. If a variable of a delayed literal is bound to a non-variable term, this literal is woken and executed in the next step if it is now sufficiently instantiated. In the permutation sort example, the programmer can add a wait declaration to the predicate `ord` and change the ordering in the first clause into

```
    sort(L,M) :- ord(M), perm(L,M).
```

Now the goal `?- sort([3,2,1],S)` is executed in the following way: After the application of the first clause to this goal the literal `ord(S)` is delayed and the literal `perm([3,2,1],S)` will be executed. If `S` is bound to the first part of a permutation of `[3,2,1]` (i.e., a list with two elements and a variable at the tail), then `ord(S)` is activated. If the first two elements of `S` are in the wrong order, then the computation fails and another permutation is tried, otherwise `ord` is delayed again until the next part of the permutation is generated. Thus with this modification not *all* permutations are completely computed and therefore the execution time is better than in the naive approach. Naish has also presented an algorithm which generates the wait declarations from a given program and transforms the program by reordering the goals in a clause. Although this approach seems to be attractive, it has some problems. For instance, the generation of wait declarations is based on heuristics and therefore it is unclear whether these heuristics are generally successful. Moreover, it is possible that the annotated program flounders, i.e., all goals are delayed which is considered as a run-time error. Hence completeness of SLD-resolution can be lost when transforming a logic program into a program with wait declarations (see example at the end of Section 3.3 or the goodpath example in [46]).

Another approach to improve control has been developed by Bruynooghe's group [7]. They try to avoid the overhead of coroutining execution by transforming a logic program with coroutining into a logic program with an equivalent behaviour executed under the standard computation rule. The transformation is done in several steps. In the first step a symbolic trace tree of a goal is created where the user has to decide which literal is selected and whether a literal is completely executed or only a single resolution step is made, i.e., the user must supply the system with a *good computation rule*. If a goal in the trace tree is a renaming of a goal in an ancestor node, an arc from this goal to the ancestor node is inserted. This results in a symbolic trace graph which is then reduced and in the last step translated into a logic program simulating the symbolic trace under the standard computation rule. The crucial point in this approach is to find a *good* computation rule for the program with respect to the initial goal. In a recent paper [46] a method for the automated generation of an efficient computation rule is presented. The method is based on a global analysis of the program by abstract interpretation techniques in order to derive the necessary information. Since the arguments for choosing a "good" computation rule are heuristics, it is unclear whether the transformed programs are in any case more efficient than the original ones. Another problem is due to the fact that their method uses a given call pattern for the initial goal. Therefore different versions of the program are generated for different call modes of the goal.

In this paper we propose a much simpler method to improve control of logic programs. This method ensures that the new programs have at least the same efficiency as the original ones. But for a large class of programs ("generate-and-test" programs like permutation sort) we obtain a better efficiency similar to other approaches to improve control. The basic idea is to use a functional logic language and to translate logic programs into functional programs (without considering the initial goal). The motivation for the integration of functional and logic programming languages is to combine the advantages of both programming paradigms in one language: the possibility of *solving* predicates and equations between terms together with the efficient *reduction* paradigm of functional languages. A lot of the proposed amalgamations of functional and logic languages are based on Horn clause logic with equality [40] where the user can define predicates by Horn clauses and functions by (conditional) equations. Predicates are often omitted because they can be represented as Boolean functions. A complete operational semantics is based on the narrowing rule [14, 29, 30]: *narrowing* combines unification of logic languages with rewriting of functional languages, i.e., a narrowing step consists of the unification of a subterm of the goal with the left-hand side of an equation, replacing this subterm by the right-hand side of the equation and applying the unifier to the whole goal. Since we have to take into account *all* subterms of a goal in the next narrowing step, this naive strategy produces a large search space and is less efficient than SLD-resolution (SLD stands for selecting *one* literal in the next resolution step). Also the advantage of functional languages, namely the deterministic reduction principle, is lost by this naive approach.

Therefore a lot of research has been done to improve the narrowing strategy without loosing completeness. Hullot [29] has shown that the restriction to *basic* subterms, i.e., subterms which are not created during unification, is complete. Fribourg [15] has proved that the restriction to subterms at innermost positions is also complete provided that all functions are reducible on all ground terms. Finally, Hölldobler [28] has proved completeness of the combination of basic and innermost narrowing where a so-called innermost reflection rule must be added for partially defined functions. But innermost basic narrowing is not better than SLD-resolution since it has been shown that innermost basic narrowing corresponds to SLD-resolution if a functional program is translated into a logic program by flattening [6]. On the other hand, we can also translate a logic program into a functional one without loosing efficiency if we use the innermost basic narrowing strategy. But now we are able to improve the execution by simplifying the goal by deterministic rewriting before a narrowing step is applied (rewriting is similar to reduction in functional languages with the difference that rewriting is also applied to terms containing variables). The simplification phase cuts down the search space without loosing completeness [28, 39].

We will see in the next sections that the operational behaviour of innermost basic narrowing combined with simplification is similar to SLD-resolution with a particular dynamic control rule. Hence we get an improvement in the execution comparable to previous approaches [7, 36] but with the following advantages:

- The translation technique from logic programs into functional logic programs is simple.
- It is ensured that the translated programs have at least the same efficiency as the original ones. For many programs the efficiency is much better.
- It is ensured that we do not loose completeness: there exists an answer w.r.t. the translated program iff there exists an answer w.r.t. the original program.

The last remark is only true if we use a fair computation strategy. If we use a backtracking implementation of SLD-resolution as in Prolog, the completeness may be lost because of infinite computations. However, infinite paths in the search tree can be cut by the simplification process [15], i.e., it is also possible that we obtain an answer from the functional logic program where the original logic program does not terminate.

These theoretical considerations are only relevant if there is an implementation of the functional logic language which has the same efficiency as current Prolog implementations. Fortunately, this is the case. In [19, 21, 24] it has been shown that it is possible to implement a functional logic language very efficiently by extending the currently known Prolog implementation techniques [47]. The language

ALF ("**Al**gebraic **L**ogic **F**unctional language") is based on the operational semantics sketched above. Innermost basic narrowing and simplification is implemented without overhead in comparison to Prolog's computation strategy, i.e., functional programs are executed with the same efficiency as their relational equivalents by SLD-resolution (see [21] for benchmarks). Therefore it is justified to improve the control of logic programs by translation into a functional logic language.

In the next section we give a precise description of ALF's operational semantics and in Section 3 we present our approach to improve control of logic programs in more detail.

2 Operational semantics of ALF

As mentioned in the previous section, we want to improve the control behaviour of logic programs by translating them into a functional logic language. We have also mentioned that in order to compete with SLD-resolution we have to use a functional logic language with a refined operational semantics, namely innermost basic narrowing and simplification. Hence the target language of the translation process is the language ALF [19, 21] which is based on this semantics. ALF has more features than actually used in this paper, e.g., a module system with parameterization, a type system based on many-sorted logic, predicates which are resolved by resolution etc. (see [25] for details). In the following we outline the operational semantics of ALF in order to understand the translation scheme presented in the next sections.

ALF is a constructor-based language, i.e., the user must specify for each symbol whether it is a constructor or a defined function. Constructors must not be the outermost symbol of the left-hand side of a defining equation, i.e., constructor terms are always irreducible. Hence constructors are used to build data types, and defined functions are operations on these data types (similarly to functional languages like ML [27] or Miranda [45]). The distinction between constructors and defined function symbols is necessary to define the notion of an *innermost position* [15].

An ALF program consists of a set of (conditional) equations which are used in two ways. In a narrowing step an equation is applied to *compute a solution* of a goal (i.e., variables in the goal may be bound to terms), whereas in a rewrite step an equation is applied to *simplify* a goal (i.e., without binding goal variables). Therefore we distinguish between *narrowing rules* (equations applied in narrowing steps) and *rewrite rules* (equations applied in rewrite steps). Usually, all conditional equations of an ALF program are used as narrowing and rewrite rules, but it is also possible to specify rules which are only used for rewriting. Typically, these rules are inductive axioms or CWA-valid axioms (see below). The application of such rules for simplification can reduce the search space and is justified if we are interested in ground-valid answers [15, 39] (i.e., answers which are valid for each ground substitution applied to it).

Figure 2 shows an ALF module to sort a list of naturals. Naturals are represented by the constructors `0` and `s`, `true` and `false` are the constructors of the data type