BIBLIOTHEQUE DU CERIST

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

456

P. Deransart J. Małuszyński (Eds.)

Programming Language Implementation and Logic Programming

International Workshop PLILP '90
Linköping, Sweden, August 20–22, 1990
Proceedings

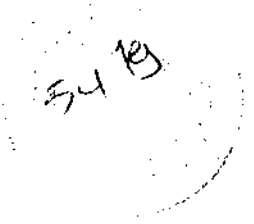
Springer-Verlag
Berlin Heidelberg New York London
Paris Tokyo Hong Kong Barcelona

Editorial Board
D. Barstow W. Brauer P. Brinch Hansen D. Gries D. Luckham
C. Moler A. Pnueli G. Seegmüller J. Stoer N. Wirth

Editors
Pierre Deransart
INRIA-Rocquencourt, Domaine de Voluceau
B.P. 105, F-78153 Le Chesnay Cedex, France

Jan Maluszyński
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden

BIBLIOTHEQUE DU CERIST



CR Subject Classification (1987): F.4.1–2, D.3.1, D.3.4, F.3.3, I.2.3

ISBN 3-540-53010-X Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-53010-X Springer-Verlag New York Berlin Heidelberg

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in other ways, and storage in data banks. Duplication of this publication or parts thereof is only permitted under the provisions of the German Copyright Law of September 9, 1965, in its version of June 24, 1985, and a copyright fee must always be paid. Violations fall under the prosecution act of the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1990
Printed in Germany

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210 – Printed on acid-free paper

Preface

This volume consists of the papers accepted for presentation at the Second International Workshop on Programming Language Implementation and Logic Programming (PLILP '90) held in Linköping, Sweden, August 20–22, 1990. Its predecessor was held in Orléans, France, May 16–18, 1988 and the proceedings of PLILP '88 were published by Springer-Verlag as Lecture Notes in Computer Science, Volume 348.

The aim of the workshop was to identify concepts and techniques used both in implementation of programming languages, regardless of the underlying programming paradigm, and in logic programming. The intention was to bring together researchers working in these fields. The papers accepted can be divided into two categories. The first of them presents certain ideas from the point of view of a particular class of programming languages, or even a particular language. The ideas presented seem to be applicable in other classes of languages and we hope that the discussions during the workshop contribute to clarification of this question. The second category addresses directly the problem of the integration of various programming paradigms.

The volume includes 26 papers selected from 96 contributions submitted in response to the Call for Papers. The contributions originated from 23 countries (Australia, Austria, Belgium, Bulgaria, Canada, P. R. of China, Denmark, Finland, France, FRG, India, Israel, Italy, Japan, Korea, Netherlands, Poland, Rumania, Sweden, Soviet Union, UK, USA and Yugoslavia). The selection was made by the Program Committee at its meeting in Linköping May 26 and 27, 1990. The choice was based on the reviews made by the Program Committee members and other reviewers selected by them. We are very grateful to all people involved in the reviewing process. They are listed on the following pages.

We gratefully acknowledge the financial support provided by Linköping University.

Le Chesnay, Linköping
June 1990

P. Deransart
J. Małuszyński

BIBLIOTHEQUE DU CERIST

Conference Chairmen

Pierre Deransart, INRIA, Rocquencourt (France)

Jan Małuszyński, Linköping University (Sweden)

Program Committee

Maurice Bruynooghe, Katholieke Univ. Leuven (Belgium)

Saumya Debray, Univ. of Arizona (USA)

Paul Franchi Zannettacci, Univ. of Nice (France)

Harald Ganzinger, Univ. of Dortmund (FRG)

Seif Haridi, SICS, Stockholm (Sweden)

Neil D. Jones, Univ. of Copenhagen (Denmark)

Feliks Kluźniak, Univ. of Bristol (UK) and Warsaw Univ. (Poland)

Vadim Kotov, Academy of Sciences, Novosibirsk (USSR)

Bernard Lang, INRIA, Rocquencourt (France)

Giorgio Levi, Univ. of Pisa (Italy)

Gary Lindstrom, Univ. of Utah, Salt Lake City (USA)

Jaan Penjam, Estonian Academy of Sciences, Tallinn (Estonia)

Masataka Sassa, Univ. of Tsukuba (Japan)

Péter Szeredi, Univ. of Bristol (UK) and SzKI (Hungary)

Martin Wirsing, Univ. of Passau (FRG)

BIBLIOTHEQUE DU CERIST

Referees

V. Akella
V. Ambriola
L.O. Andersen
N. Andersen
A. Arnold
I. Attali
R. Bahgat
R. Barbuti
T. Beaumont
M. Bellia
N. Bidoit
A. Blikle
A. Bondorf
S. Bonnier
F. Boussinot
A. Bouverot
G. Bracha
J.P. Briot
A. Brogi
B. Bruderlin
M. Bruynooghe
M.A. Bulyonkov
A. Burt
A. Callebout
P. Casteran
J. Chazarain
P.H. Cheong
L. Chernoboed
G.D. Chinin
P. Ciancarini
D. Clément
P. Codognet
P. Cousot
D. Craeynest
J. Daels
M. Danelutto
O. Danvy
J.-F. Dazy
A. De Niel
D. De Schreye
S. Debray
P. Dembinski
B. Demoen
M. Denecker
P. Deransart
J. Despeyroux
T. Despeyroux
P. Devienne
Y. Deville
J.-L. Dewèz
V. Donzeau-Gouge
W. Drabent
H. Dybkjaer
C. Fecht
A. Feng
G. Ferrand
P. Franchi Zannettacci
U. Fraus
P. Fritzson
I. Futó
M. Gabbrielli
J. Gallagher
G. Gallo
H. Ganzinger
M. Gengenbach
L. George
G. Ghelli
R. Giegerich
C. Gomard
J. Goossenaerts
G. Gopalakrishnan
S. Gregory
K. Grue
A. Guendel
I. Guessarian
G. Gupta
M. Hanus
A. Haraldsson
T. Hardin
S. Haridi
L. Hascoet
B. Hausman
R. Hennicker
P. Van Hentenryck
P. Hill
A. Hirschowitz
K.H. Holm
C.K. Holst

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

I. Holyer
H. Hussmann
S. Janson
G. Janssens
T.P. Jensen
N.D. Jones
M. Jourdan
K. Kaijiri
F. Kluźniak
J. Komorowski
V. Kotov
P. Kreuger
B. Krieg-Brückner
K. Kuchcinski
B. Lang
P. Lebègue
O. Lecarme
B. Legeard
X. Leroy
G. Levi
J.-J. Lévy
V. Lextrait
G. Lindstrom
A. Lomp
B. Lorho
E. Madeleine
D. Maier
K. Malmkjaer
J. Małuszyński
L. Maranget
A. Marien
M. Martelli
B. Mayoh
M. Méristé
M. Miłkowska
T. Mogensen
J. Montelius
P.D. Mosses
T. Muchnick
A. Mulkers
V.A. Nepomniaschy
F. Nickl
M. Nilsson
U. Nilsson
T. Nishino
T. Ogi
C. Palamidessi
D. Parigot
M. Patel
J. Penjam
K. Petersson
L. Pottier
A. Quéré
S. Raina
P. Richard
M. Rosendahl
R. Rousseau
M. Rueher
V. Sabelfeld
D. Sahlin
M. Sassa
Y. Sato
R. Schäfers
M. Schwartzbach
P. Sestoft
J. Shepherdson
Y. Shinoda
M. Simi
T. Sjöland
H. Søndergaard
R. Sosic
K. Studziński
P. Szeredi
N. Tamura
L. Tan
F. Turini
K. Verschaetse
U. Waldmann
J. Wang
P. Weemeeuw
P. Weis
U. Wertz
R. Wilhelm
J. Winkowski
M. Wirsing
Y. Yamashita
D. Yeh

Table of Contents

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

BIBLIOTHEQUE DU CERIST

Implementing Parallel Rewriting*

Claude Kirchner **Patrick Viry**
INRIA Lorraine & CRIN
615 Rue du Jardin Botanique, BP101
54600 Villers les Nancy, France
E-mail: {ckirchner,viry}@loria.crin.fr

Abstract

We present in this paper a technique for the implementation of rewriting on parallel architectures. Rewriting is a computation paradigm that allows to implement directly an equational specification (eg. an abstract data type). Much work has been done about theoretical aspects of rewriting, which has made this technique of practical interest for programming. The next step for rewriting to be used in practice is now to provide an efficient implementation for it. We present here an implementation technique that enables to take advantage of the computational power of loosely-coupled parallel architectures with any grain size. Restricted to one processor, the efficiency of this technique is in the same order of magnitude as those of functional languages such as interpreted LISP or ML, and we expect an almost linear increase of the efficiency when increasing the number of processors. It is important to notice that this approach allows parallel execution of programs directly from their equationally axiomatized specification, without having to make explicit at all the potential parallelism, thus providing a simple and precise operational semantics.

1 Introduction

Rewriting is a computational paradigm that is now widely recognized and used. As a mathematical object, rewrite systems have been studied for more than ten years, and the reader can find in [16] and [2] general surveys describing properties and applications either in theorem proving or in programming languages. Rewriting implementations on sequential machines are numerous and a survey of most of them is made in [12].

Rewriting for computing has been developing for several years [1]. It is in particular used as an operational semantics in many programming languages like OBJ [4,5], ASF [11] or SLOG [3] among many others. It is thus crucial, in order to get realistic performances, to have efficient implementations of the rewriting concept. Several alternatives have been explored. A first one is to compile rewriting either using abstract machines like in [15,19] or using a functional language like in [13,17]. A second one (possibly complementary), on which this paper is based, is to implement rewriting on parallel machines. This is not a fashion effect: rewriting is really a computational paradigm which specifies the actions and not the control (but strategies may be added if explicit control is needed). Moreover for linear rules the computations needed to apply a rule are completely local. It is thus a paradigm which can be directly implemented on a parallel machine and has the advantage of freeing the programmer of any explicit parallelization directive to the program. Moreover it allows elimination of the intermediate steps between the program description and its implementation: an implementation of rewriting is an implementation of an operational semantics

*This research has been partially supported by the *GRECO de Programmation of CNRS*, the Basic Research Workshop COMPASS of the CEC and contract MRT 89P0423.

of abstract data types. This is already a main concept in the rewrite rule machine project [6,8] whose model of computation is concurrent rewriting [7]. The goal of this project was to design a hardware with a rewrite rules machine code. Our purpose here is quite different, since it consists in implementing rewriting on *existing* parallel machines like the connection machine or transputer based machines.

In order to implement term rewriting, several steps are involved. Let us take as example the following rewrite program specifying the computation of the length of a list of integers:

op	nil	:	$\longrightarrow ListInt$	$length(nil)$	$\rightarrow$	0
op	$\_\_$	: $Int, ListInt$	$\longrightarrow ListInt$	$length(n.L)$	$\rightarrow$	$length(L)+1$
op	$length$: $ListInt$		$\longrightarrow Nat$			

where we assume known the usual operations on integers. These rules will be directly used to compute the length of the list $(3.(-4.(3.nil)))$ by applying the rules on the term $length(3.(-4.(3.nil)))$. No intermediate compilation neither of the term nor of the rewrite rules will be necessary.

In order to perform these computations, a pattern should first be *matched* against the term to be reduced. For example $length(n.L)$ matches the term $length(3.(-4.(3.nil)))$, and the *substitution* allowing the match, in this case $\{n \mapsto 3, L \mapsto (-4.(3.nil))\}$, is computed. Then the right-hand side of the rule is *instantiated* with the match substitution: here $length(L)+1$ is instantiated into $length(-4.(3.nil))+1$. Finally the redex should be *replaced* by the instantiated right-hand side: in the example $length(3.(-4.(3.nil)))$ is replaced by $length(-4.(3.nil))+1$. And the same process can be iterated until an irreducible term may be obtained. Notice that in this example, the computations are performed only locally since the rewriting system is *left linear*. i.e. the variables occur only once in all the left-hand sides of the rewrite rules.

The crucial idea is that if one wants to reduce the term $length(2.(3.nil))+length(3.(-4.(3.nil)))$, the computations can be performed independently on the subterms $length(2.(3.nil))$ and $length(3.(-4.(3.nil)))$. Moreover, since no control is a priori given in a specification based on rewrite systems, the implementation can freely use the inherent parallelism contained, but not explicitly specified, in the rewrite program: *A rewrite program is a parallel specification.* In this paper we show how to get an implementation of rewriting that exploits this remark. Note that strategies can be specially designed for controlling parallelism, see [7].

Let us summarize what we call *parallel rewriting*. The first idea is to use as model of parallel rewriting the notion of concurrent rewriting as defined in [7]. Concurrent rewriting, that we will precisely define in the next section, is the relation describing *simultaneous* rewrite of a non-empty set of *disjoint* redexes. But enforcing this in the implementation will require some synchronization, operation that we would like to forbid as much as possible. Thus, we consider that the processors independently detect redexes throughout the term and reduce them without synchronization. For this to be correct, datas should be represented as DAGs (directed acyclic graphs) in order to (1) allow rewritings to occur everywhere even on non-disjoint redexes, (2) allow the substitutions computed at matching time to subsist after other (independent) rewritings. Moreover this DAG structure allows the sharing of common parts of the term, so that its representation is more compact and computation can be shared. Since we would like the computation to occur everywhere in the term, we consider each node of the DAG to be a process communicating with the other processes through channels following the edges of the DAG.

The second key idea is to perform matching, i.e. detection of redexes, using only local informations that are not necessary up-to-date with respect to the other ongoing reductions. This point is developed in Section 3, based on a parallel version of the bottom-up matching algorithm of [14]. The computations remain local only when the left-hand side of the rules are linear. When not, we postpone their applications after all linear computations have been performed, as described in Section 4.2.

Section 4 precises how parallel rewriting, built from the two main ideas above, is a correct implementation of concurrent rewriting. This leads to the implementation described in Section 5. The program runs currently on one processor with performances eight to ten times slower than

BIBLIOTHEQUE DU CERIST

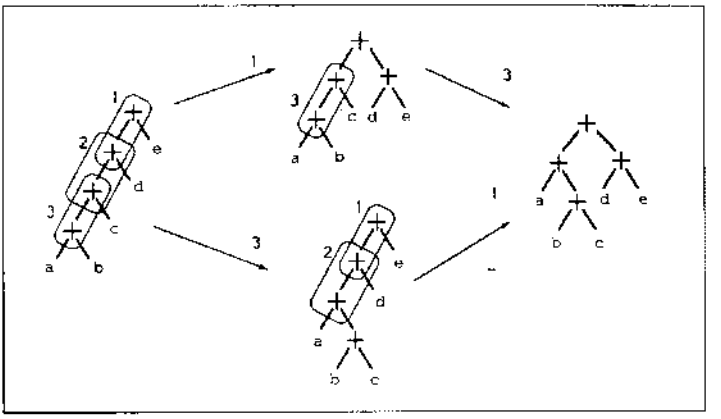
interpreted LISP, but it is not at all optimized, and we expect an almost linear increase in terms of the number of processors.

We do not recall the formal definitions of the concepts needed in rewriting systems and refer to [2,16,7]. In particular we suppose the reader familiar with the notions of term, position (or occurrence), equations, overlap.

2 Concurrent Rewriting

The concurrent rewriting relation on terms has been introduced in [7] to formalize the idea that many redexes in a term may be rewritten simultaneously while keeping the semantics of rewriting.

Let us give an example: the following term may be rewritten by the right-associativity rule $(x + y) + z \rightarrow x + (y + z)$ at the three positions (redexes) numbered 1, 2, 3. We would like to rewrite in a single step the redexes 1 and 3, giving the same result as rewriting successively the positions 1 then 3, or 3 then 1. But we can see that it would have no sense to rewrite in a single step 1 and 2, because after rewriting redex 1, the redex 2 disappears.



So in order to formally define concurrent rewriting, we have to introduce the notion of non-overlapping set of redexes:

Definition 1 *Let t be a term and R a term rewriting system. Let $R(t) = \{(p_i, l_i, r_i)\}$ be the set of all the redexes in t under R; i.e.*

$$(p_i, l_i, r_i) \in R(t) \Leftrightarrow l_i \rightarrow r_i \in R \text{ and } \exists \sigma \text{ such that } t_{|p_i} = \sigma(l_i)$$

A subset W of $R(t)$ is said to be **nonoverlapping** *iff for any redexes (p, l, r) and (p', l', r') in W,*

- *p and p' are incomparable (none is a substring of the other)*
- *or p is a substring of p' and there exists a variable position q in l such that $p.q$ is a substring of p'*
- *if l is non-linear for the variable x and if there exists a position q of x in l such that $p' = p.q.r$ for some r, then $(p.q'.r) \in W$ for all positions q' of x in l.*

The third condition comes from the fact that if a non left-linear rule is applyable, then some subterms have to be equal. If we apply this rule concurrently with some others, this equality must be preserved or concurrent rewriting would not be correct, so we have to perform the same rewritings in these equal subterms.

We can now define the concurrent rewriting relation: let $\Delta(t)$ be the set of all nonoverlapping subsets of redexes in t.

BIBLIOTHEQUE DU CERIST