University of Novi Sad
Faculty of Science
Institute of Mathematics

Ratko Tošić
Zoran Budimac (Eds.)

LIRA '97

Proceedings of VIII International Conference on Logic and Computer Science

- Theoretical Foundations of Computer Science -

September 1 - 4, 1997.
Novi Sad, Yugoslavia

Ministarstvo za nauku i tehnologiju Republike Srbije
finansiralo je štampanje ove publikacije

Typesetting: Authors and Dragan Mašulović, with precious help of Mihal Badjonski and Lehel Szarapka.
Printed in: 250 copies

Printer: "Mala knjiga", Novi Sad, Yugoslavia

Preface

The first conference on Logic and Computer Science was held in Novi Sad in 1987. The following conferences were held respectively in Ohrid, Kragujevac, Dubrovnik, Cavtat, and Novi Sad (twice.) The aim of the conferences was to gather logicians and computer scientists and to encourage their joint work and the interchange of ideas.

The VIII conference on Logic and Computer Science LIRA '97 is taking place at the Institute of Mathematics, Faculty of Science, University of Novi Sad, Novi Sad, Yugoslavia, September 1 - 4, 1997. The conference is international and its intention is to explore all fields related to theoretical and mathematical foundations of computer science.

The conference features four preliminary and invited lectures and presentations of 31 papers. 29 papers are printed in this volume, while the rest will be included in the accompanying booklet (due to strict printer's deadline.) All papers were reviewed by at least two members of the program committee and/or by other competent specialists.

We use this opportunity to express our thanks to the conference organizers, all members of the program committee, referees, and finally to the conference sponsor NIS - Novi Sad.

Ratko Tošić and Zoran Budimac, editors

Conference Organization

Institute of Mathematics, Faculty of Science, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
tel.: (+381-21)58-136, (+381-21)58-888
fax.: (+381-21)350-458
http: www.im.ns.ac.yu/events/lira97/
e-mail: lira@unsim.ns.ac.yu

Conference Topics

- Relationships of logic, algebra, and computing,
- Applications of logic and algebra in computing,
- Applications of computer science in logic and algebra,
- Automata and formal languages,
- Logic, functional, and parallel programming,
- Programming languages,
- Compilers and compiler generators,
- Database theory and relational algebras,
- Algorithms, complexity, and computability,
- Combinatorics and graph theory,

and other topics related to theoretical foundations of computer science.

Organizing Committee

- Đura Paunić, Chair
- Mirjana Ivanović, Secretary
- Zoran Budimac, Secretary
- Dragoslav Herceg
- Ratko Tošić
- Gradimir Vojvodić
- Silvia Ghilezan
- Dragan Mašulović
- Mihal Bađonski
- Boža Tasić

Program Committee

Referees

Contents

Preliminary and Invited Lectures

Papers

Late papers

Sponsor: NIS - Novi Sad

Preliminary and Invited Lectures

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 1–3

Von Neumann Regularity in Applied Algebra, Computing and Logic

Sam L. Blyumin

LSTU
30 Moskovskaya, Lipetsk 398 055, Russia
e-mail: sam@blyumin.lipetsk.su

Abstract. 1. Von Neumann regular element, firstly defined in ring R [1], is naturally defined in semigroup S [2]: $\{a \in S \text{ is regular}\} \Longleftrightarrow \{g \in S \text{ exist such that } a \cdot g \cdot a = a\}$; g is called the generalized inverse for a [3]; any such g is denoted by a^-.

2. Investigation and solution of equation $a \cdot x = b$ in groupoid G is the important area for application of regularity [3]: (i) if G is S and a is regular, then $\{a \cdot x = b \text{ is solvable}\} \Longleftrightarrow \{a \cdot a^- \cdot b = b \text{ for some } a^-$; some solution is $x = a^- \cdot b\}$; (ii) if G is R and a is regular, then the same hold, general solution is $x = a^- \cdot b + y - a^- \cdot a \cdot y$, any $y \in R$ (related topic: [4]).

3. In some applications (e.g. usual and fuzzy set theory, usual and fuzzy logic, decision theory a.o.[5]) semirings SR are used, i.e. algebraic structures with two distributively connected associative operations (distributive bisemigroups); regularity leads to notions of generalized inverse as well as generalized opposite elements; investigation and solution of equations in such (more general than in rings) situation are discussed in the lecture (related topic: [6]).

4. In some applications nonassociative algebraic structures, e.g. arbitrary groupoids G, are used; then the notion of regularity bifurcates, e.g. the regularity of the first or the second kind: $(a \cdot g) \cdot a = a$ or $a \cdot (g \cdot a) = a$; investigation and solution of equations in such (more general than in semigroups) situation are discussed in the lecture (related topic: [7]).

5. Some another identities than associativity identity may hold in groupoid, e.g.: $x \cdot x = x$ (idempotency), $(x \cdot y) \cdot x = x \cdot (y \cdot x)$ (elasticity), $x \cdot (y \cdot (x \cdot z)) = ((x \cdot y) \cdot x) \cdot z$ (moufang) a.o. Thus problem arises of identities and corresponding groupoids' manifolds generation and classification with search of nonisomorphic classes and their representatives as in [8] for the case of semigroups; this problem needs use of computer realization. Some results are presented in [9] and are discussed in the lecture.

6. Wide area of algebra applications in geometry and physics deals with Clifford algebras (in wide sense, including Grassmann algebras); simplest are the complex number field $\mathsf{C} = \{a + b \cdot i, i^2 = -1\}$, the dual "number" algebra V

$= \{a + b \cdot \varepsilon, \varepsilon^2 = 0\}$, and the double "number" algebra $\mathsf{U} = \{a + b \cdot e, e^2 = 1\}$ $(a, b \in \mathsf{R})$, because of any Clifford algebra A is the skew tensor product of some copies of C,V,U [10]. Regularity in V,U is considered in [4]. Thus problem arises of expression of skew tensor product element generalized inverse via generalized inverses of factors. Some results are presented in [11] and are discussed in the lecture.

7. Clifford algebras are Z_2- graduated algebras, or superalgebras [10]. Some results on invertibility of such algebra elements are presented in [12] and are discussed in the lecture.

8. If criterion of solvability of equation (see item 2 above) doesn't hold then exact solution of equation doesn't exist; it is possible however in some cases to find approximate (in some sense) solution using corresponding special type of generalized inverse. Classical type is the weighted pseudoinverse $G = A^+_{MN}$ for complex matrix A which is defined by the relations [3] $M \cdot A \cdot G \cdot A = M \cdot A, N \cdot G \cdot A \cdot G = N \cdot G, (M \cdot A \cdot G)^* = M \cdot A \cdot G, (N \cdot G \cdot A)^* = N \cdot G \cdot A$ and defines the general weighted pseudosolution $X = A^+_{MN} \cdot B + Y - A^+_{MN} \cdot A \cdot Y$, any matrix Y, of matrix equation $A \cdot X = B$, minimizing the weighted Frobenius matrix norm of residual $A \cdot X - B$. Some applications to optimal control problems are presented in [13] and are discussed in the lecture.

9. Optimizational aspects of generalized inversion outlined above are discussed in [14] and in the lecture.

10. Computational aspects of generalized inversion are connected with elaboration of effective algorithms. Classical Greville recurrent column-wise pseudoinversion algorithm is extended in [4] to recurrent column-wise algorithm for generalized inversion of a matrix over an associative ring. In combination with its row-wise version it allows to give the constructive proof of classical theorem: matrix ring over regular ring is regular ring too. Recurrent nature of such algorithms finds usefull applications in different areas. As an example, the superpositional nonlinear regression is developed in [15] and is discussed in the lecture.

Some another related topics will be discussed in the lecture too.

References

1. Neumann, J. von. On Regular Rings. Proc.Nat.Acad.Sci.U.S.A. **22**, 707-713, 1936.
2. Clifford A., Preston G. The Algebraic Theory of Semigroups. A.M.S., Rhode Island, 1964.
3. Rao C., Mitra S. Generalized Inverses of Matrices and their Applications. Wiley, New York, 1970.
4. Blyumin S., Milovidov S. Investigation and Solution of Matrix Equations over Associative Rings. Comp.Maths Math.Phys. **34**, 133-142, 1994.
5. Applied Fuzzy Systems. Omsya, Tokyo, 1989.
6. Blyumin S. Algebraic Foundations of Fuzzy Petri Nets Theory: Fuzzy Sets Algebra Axioms. IFSA'97 (Int. Fuzzy Systems Assoc. World Congress). Prague, Czech Republic, June 1997.
7. Blyumin S., Milovidov S. Equations in Semigroups. LSTU, Lipetsk, 1996.

8. Crvenkovic S., Stojmenovic I. An Algorithm for Cayley Tables of Algebras. Zb.Rad.Prir.-Mat.Fac.Ser.Mat.Univ.Novom Sadu **23**, 221-231, 1993.
9. Blyumin S., Ivanova G. Computer Generation of Groupoids. Current Informatization Problems (II Republic Electron. Sci. Conf., Voronezh) 121, Apr. 1997.
10. Postnikov M. Lie Groups and Algebras. Nauka, Moscow, 1982.
11. Blyumin S., Krivovyaz E., Milovidov S., Mishachev N. Regularity of Skew Tensor Products of Operators. Systems Modelling and Stability Investigation (VIII Ukrain. Conf., Kiev) 21, May 1997.
12. Blyumin S., Krivovyaz E., Milovidov S., Mishachev N. Z_2-Graduated Algebra Elements Invertibility. Pontryagin's Talk - VIII (Spring Math. School, Voronezh) 23, May 1997.
13. Blyumin S., Milovidov S. Weighted Pseudoinversion in Optimal Control for Discrete Argument Systems. Proc. AS USSR. Techn. Cybernet. N 6, 203, 1990.
14. Blyumin S. Generalized Inversion in Optimization. 5th SIAM Conf. on Optimization. Victoria, British Columbia, Canada, May 1996.
15. Blyumin S. Superposition in Nonlinear Programming: Superpositional Regression. Int. Symp. on Operation Research. Jena, Germany, Sept. 1997.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 5–5

Coordinated products of iterative algebras

Ivan A. Malcev

Institut matematiki SO RAN
Novosibirsk 90, 630090, Russia
e-mail: anatoly@doc.nsk.su

(The text of the lecture will be available during the conference.)

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 7–16

On two counterfeit coins conjecture

Ratko Tošić

Institute of Mathematics, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: ratosic@unsim.im.ns.ac.yu

Abstract. The purpose of this paper is to survey the results concerning the combinatorial search problems, with special attention to two counterfeit coins conjecture.

Key Words and Phrases: search theory, group tests, optimal procedure

AMS Subject Classification (1991): 90B40

1 Introduction

The determination of "defective" elements in a population of a series of group tests has received considerable attention in recent years. While traditionally group testing literature employs probabilistic models, the combinatorial model has cut his own share. Futhermore, combinatorial group testing has tied its knots with many computer science subjects: complexity theory, computational geometry and lerning models among others. It has also been used in multiaccess communication and coding.

Unlike many other mathematical problems which can trace back to earlier centuries and divergent sources, the origin of group testing is pretty much pinned down to a fairly recent event – World War II, and is usually credited to a single person – Robert Dorfman. The problem goes back to questions arising in connection with medical examination during the World War II; for some early papers see, e.g., Dorfman [27], Sterrett [80] or Sobel [78]. Katona [54] gives an excellent overwiew of the subject.

Consider the following problem. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n coins, indistinguishable except that exactly m of them are slightly heavier than the rest (in the sense specified below). Given a balance scale, we want to find an optimal

Work supported by Ministry of Science and Technology of Serbia.

weighing procedure, i.e., a procedure which minimizes the maximum number of steps (weighings) which are required to identify all heavier coins.

We suppose that all heavier coins are of equal weight, and so are all light coins. If λ is the weight of a light coin, then the weight of a heavy coin is less than $\frac{m+1}{m}\lambda$, so that the larger of two numerically unequal subsets of X is always the heavier. So it is clear that no information is gained by balancing two numerically unequal subsets of X. We also suppose that the scale reveals which, if either, of two subsets of X is heavier but not by how much.

Step (A, B) will mean the balancing of A agains B, where A and B are disjoint subsets of X of the same cardinality. The possible outcomes are:

(a) $A = B$ (the sets balance),

(b) $A \neq B$ (the sets do not balance).

We use the notations $A < B$, $A > B$, where $<$ and $>$ between two sets mean "is lighter than" and "is heavier than" respectively.

By $\mu_m(n)$ we denote the number of steps (weighings) in an optimal procedure. It follows by information–theoretical reasonings that

$$\mu_m(n) \geq \lceil \log_3 \binom{n}{m} \rceil.$$

2 One Counterfeit Coin

The question of finding a single counterfeit coin from a set of regular coins in the fewest number of weighings using just a balance beam has been a notorious problem. The regular coins are all of the same weight while the counterfeit coin is a different weight. A large number of ingenious solutions exist, some based on sequential procedure and some not.

The problem was popular during World War II; see [33,34,40,41,42,53,64,74,90] for some history. Many authors gave the following general solution to the problem of underweight counterfeit coins:

If $3^{k-1} \leq n < 3^k$, then k weighings suffice to show if there is (and to identify) a counterfeit coin among n coins.

If it is known that a counterfeit coin exists, then k weighings will suffice to identify the coin from among n coins if $3^{k-1} < n \leq 3^k$. In the case when it is not known if the counterfeit coin is heavy or light, Dyson [29] gave an elegant solution using ternary labels. In this case, k weighings suffice

(a) if $n \leq \frac{3^k-3}{2}$ and it is required to find if the counterfeit coin is heavy or light;

(b) if $n \leq \frac{3^k-1}{2}$, given an extra coin known to be good, and it is required to find if the counterfeit coin is heavy or light;

(c) $n \leq \frac{3^k+1}{2}$ if there is a good coin but the relative weight of of the counterfeit coin is not required.

Blanche Descartes [26] gave an interpretation of these results in verse.

All the solutions so far cosider coins to be distinguishable when in the balance pan. Guy and Nowakowski [44] show that if the coins in a scale pan are to be cosidered as a single set, then k weighings will find a coin amongst $n \leq \frac{7 \cdot 3^{k-2}-1}{2}$.

Shapiro's problem [76] assumes n coins, $n-1$ of weight a and one of weight b, where a and b are known, and an accurate scale. He asks for the least number of weighings to determine which coin has weight b, where the weighing scheme must be given in advance. Söderberg and Shapiro [79] ask the more general question of how many weighings are needed to determine which of n coins are of weight a and which of weight b if the numbers of each are not known in advance. They show that

(a) $\mu_1(n) \geq \frac{n}{\log_2(n+1)}$;
(b) $\mu_1(3^{k-1}(3+k)) \leq 3^k$;
(c) $\mu_1(5^{k-1}(2k+5)) \leq 5^k$;
(d) $\mu_1(n) = O(\frac{n}{\ln n})$.

Erdös and Rényi [30] show that

$$\mu_1(n) = \frac{n}{\log_4 n} + O(\frac{n \ln \ln n}{(\ln n)^2}).$$

Liu [61], Cantor and Mills [16] and Lindström [56,57,58] give explicit weighing schemes for $n = 2^{k-1}k$ (also see [1]).

The "Lower Slobbovian Counterfeiters" [13,47] and ApSimon's Mints problem [8] are examples of another variant of deciding which coins are irregular out of n coins when the number of weighings is fixed.

3 Two Counterfeit Coins

Oddly enough, the corresponding problem for more than one defective coin has attracted little attention. At the suggestion of R. Bellman, the problem for two coins was investigated for the first time by Cairns (1955). The problem is of significance because it represents one of the simplest examples of a sequential testing problem replete with the difficulties of combinatorial nature and with the difficulties inherent in the concept of "information". A common phenomenon in combinatorial search theory is that while it is often straightforward to find an optimal procedure searching for one object, it is immensely more difficult to search optimally for two objects. Bellman and Gluss [10] studied the problem of identifying two irregular coins in a set of n coins with a balance scale. They wrote: "A small amount of analysis discloses the enormous difference in complexity between the one–coin and the two–coins problem."

Tošić [82] show that

$$\lceil \log_3 \binom{n}{2} \rceil \leq \mu_2(n) \leq \lceil \log_3 \binom{n}{2} \rceil + 1.$$

He also prove that an optimal algorithm can be constructed for all n's belonging to the set

$$\cup_{k \geq 1}([\lceil 3^k\sqrt{2}+1\rceil, 2 \cdot 3^k] \cup [\lceil 3^k\sqrt{6}+1\rceil, 3^{k+1}].$$

Lately, this result were improved several times, see [11,7,86]. The results are summerized in the following table:

$\mu_2(n) = 2k+1$	$\mu_2(n) = 2k+2$	
$[\lceil 3^k\sqrt{2}+1\rceil, 2\cdot 3^k]$	$[\lceil 3^k\sqrt{6}+1\rceil, 3^{k+1}]$	Tošić [82]
$[\lceil 3^k\sqrt{2}+1\rceil, 20\cdot 3^{k-2}]$	$[\lceil 3^k\sqrt{6}+1\rceil, 4\cdot 3^k]$	Bošnjak, Tošić [11]
$[\lceil 3^k\sqrt{2}+1\rceil, 21\cdot 3^{k-2}]$	$[\lceil 3^k\sqrt{6}+1\rceil, 4\cdot 3^k]$	Anping [7]
$[\lceil 3^k\sqrt{2}+1\rceil, 64\cdot 3^{k-3}]$	$[\lceil 3^k\sqrt{6}+1\rceil, 4\cdot 3^k]$	Tošić [86]

In [44] Guy and Nowakowski ask: In which cases is $\mu_2(n) = \lceil \log_3 \binom{n}{2} \rceil + 1$? Is $n = 13$ the first? The answer is: No. In what follows we give an algorithm which proves that $\mu_2(13) = 4$.

The first step is (A, B) where $A = \{x_1, x_2, x_3, x_4\}$ and $B = \{x_5, x_6, x_7, x_8\}$.

(a) If $A > B$, the second step is (C_1, C_2), where $C_1 = \{x_1, x_9, x_{10}\}$ and $C_2 = \{x_2, x_{11}, x_{12}\}$.

(a1) If $C_1 > C_2$, the third step is $(\{x_9\}, \{x_{10}\})$. It is easy to check that, independently of the answer, the space of solutions after this step will contain at most three elements and one additional step will suffice to find counterfeit coins.

(a2) If $C_1 = C_2$, than the third step is $(\{x_1\}, \{x_{13}\})$. After this step, the space of solutions contains at most three elements, and again one additional weighing is sufficient.

(a3) The case $C_1 < C_2$ is symmetrical to (a1). Now, the third step is $(\{x_{11}\}, \{x_{12}\})$.

(b) If $A = B$, the second step is (C_3, C_4), where $C_3 = \{x_1, x_5, x_6, x_9, x_{10}\}$ and $C_4 = \{x_2, x_7, x_8, x_{11}, x_{12}\}$.

(b1) If $C_3 > C_4$, the third step is $(\{x_5\}, \{x_6\})$. The space of solutions after this step will contain at most three elements and one additional step will suffice to find counterfeit coins.

(b2) If $C_1 = C_2$, than the third step is $(\{x_1\}, \{x_7\})$. After this step, the space of solutions contains at most three elements, and again one additional weighing is sufficient.

(b3) The case $C_1 < C_2$ is symmetrical to (b1). Now, the third step is $(\{x_7\}, \{x_8\})$. □

The following conjecture in various forms can be found in several papers.

Conjecture.

$$\mu_2(n) = \lceil \log_3 \binom{n}{2} \rceil.$$

There are many other variants in identifying two irregulars [1,17,23,62]. Forsythe, responding to [33], seems to be the firsth to ask the question using a spring balance, i.e. a weighing device that will return the exact weight; see also [72,76]. In [19] Christen investigates the case of 2 counterfeit coins but of complementary weights.

Hwang [49] proposes and analyses many weighing schemes. While a balance scheme provides information about the irregulares by comparing the weights of two subsets of coins, the irregulars can be also detected by a spring scale which provides information by weighing a subset of coins. Instead of talking about the balance scale and the spring scale, the more general terms of the comparison–type device and the test–type device can be used. Hwang [49] survey results

from various models of the test–type device in identifying two irregulars (not necessarily identical).

Aigner [2] studies the following natural generalization to graphs of model Q (Hwang [49]).

Main Problem. *Given a finite simple graph G with vertex–set V and edge–set E, and an unknown edge $e \in E$. In order to find e we choose a sequence of test–sets $A \subseteq V$ where after every test we are told whether e has both end–vertices in A, one end–vertex, or none. Find the minimum number $c(G)$ of tests required.*

Since we perform a sequence of ternary tests, we have the usual information–theoretic lower bound for $c(G)$:

$$c(G) \geq \lceil \log 3q \rceil, \quad q = |E|.$$

The following two particular cases of main problem have received particular attention [].

Problem 1. *Given a finite set S with $|S| = n$. We know that there are precisely two defective elements $x^*, y^* \in S$. In order to find the defective elements we choose a sequence of test–sets $A \subseteq S$. At the end of every test A we receive as answer how many defective elements A contains. Determine the minimum number of tests required to find x^* and y^*.*

Problem 2. *Consider two disjoint sets S and T with $|S| = m$ and $|T| = n$. It is known that either set contains precisely one defective element. We again perform a sequence of tests $A \subseteq S \cup T$ as just described. What is the minimum number of tests required in this situation?*

It is clear that the two problems mentioned above correspond to $G = K_n$ (complete graph) and $G = K_{m,n}$ (complete bipartite graph).

As is common in search theory we distinguishe between sequential and predetermined procedures (see e.g. Ahlswede–Wegener [4] or Katona [54]). The case of predetermined procedures has been thouroughly studied by many authors (see Lindström [56,57], Cantor–Mills [16], Erdös–Renyi [30] and the bibliography in [57]).

A binary variant of search problem in graph is the following. Again we perform a sequence of tests $A \subseteq V$ on the given graph $G = (V, E)$ with unknown edge e. After every test we now receive as an answer whether e has at least one end–vertex in A or none. Again we are asked to find the minimum number $\bar{c}(G)$ of tests required. Obviously, $c(G) \leq \bar{c}(G)$ for any graph G, and the information–theoretic lower bound is $\lceil \log q \rceil$:

$$\bar{c}(G) \geq \lceil \log q \rceil, \quad q = |E|.$$

A graph $G = (V, E)$ with at least two edges is called *optimal (2–optimal)* if $c(G)$ ($\bar{c}(G)$) achieves the information–theoretic lower bound.

Aigner [2] showed that (a) any forest with maximum degree ≤ 2 is optimal, and (b) a cycle C_n ($n \geq 3$) is optimal iff n is not a power of 3.

Aigner [2] proved that any forest is 2–optimal. Tosic and Masulovic [86] proved that any planar graph is 2–optimal.

4 Many Counterfeit Coins

The case $m = 3$ was the subject of a problem discussed by Collings [25]. Tošić [83] proved that for $n = 3^k$, $n = 3^k + 3^{k-1}$ and $n = 2 \cdot 3^k$,

$$\lceil log_3 \binom{n}{3} \rceil \leq \mu_3(n) \leq \lceil log_3 \binom{n}{3} \rceil + 1.$$

In [12] Bošnjak proved that

$$\mu_{2,1}(3^k, 2 \cdot 3^{k-1}) \leq 3k - 1, \quad k \geq 1,$$

and for each natural n,

$$\mu_3(n) \leq \lceil log_3 \binom{n}{3} \rceil + 1.$$

It is easy to see that for some natural numbers the information–theoretical lower bound cannot be achieved. For example, $\lceil \log_3 \binom{50}{3} \rceil = \lceil \log_3(19600) \rceil = 9$, but it is not difficult to verify that $\mu_3(50) = 10$.

For $m = 4$, it is known that $\mu_4(3^k) \leq 4k - 1$ (Guy, Nowakowski [43]), $\mu_4(3^k + 3^{k-1}) \leq 4k$ (Tošić [84]), and $\mu_4(2 \cdot 3^k) \leq 4k + 2$ (Tošić [84]). For some natural numbers, the information–theoretical lower bound cannot be achieved. For example, $\mu_4(8) = 5$; although $\binom{8}{4} = 70 < 3^4$, it is not possible to make a weighing among 8 coins with 4 counterfeit each of whose outcomes leave less than 26 possibilities and while $26 < 3^3$, they cannot be separated by three weighings.

If $m = 5$, it is known that $\mu_5(3^k) \leq 5k$ (Guy, Nowakowski [43]), $\mu_5(2 \cdot 3^k) \leq 5k + 1$ (Tošić [85]).

Pyber [68] showed that

$$\mu_m(n) \leq \lceil \log_3 \binom{n}{m} \rceil + 15m.$$

The constant $15m$ can be much improved. The guess of Pyber is that

$$\mu_m(n) \geq \lceil \log_3 \binom{n}{m} \rceil + \epsilon m \text{ for some } \epsilon > 0.$$

Some interesting connections between counterfeit coins problem and many other combinatorial problems are established in [5] and some other papers given at the and of this article.

References

1. Aigner, M., Combinatorial Search, Wiley – Teubner, New York, 1988.
2. Aigner, M., Search Problems on Graphs, Discrete Appl. Math. 14(1986)215–230.
3. Aigner, M., Schughart, M., Determining Defectives in a Linear Order, J. Statist. Plann. Inf., 12(1985)359–368.
4. Ahlswede, Wegener, I., Suchprobleme, Teubner, Stuttgart, 1979.

5. Alon, N., Kozlov, D. N., Vu, V. H., The Geometry of Coin–Weighing Problems, Proceedings of 37th Annual Symposium on Foundations of Computer Science, October 14–16, 1996, Burlington, Vermont, 524–532.
6. Andreae, T., A Ternary Search Problem on Graphs, Discrete Appl. Math. 23(1989)1–10.
7. Anping, L., On the Conjecture of two Counterfeit Coins, Discrete Math. 133(1994)301–306.
8. ApSimon, H., Mathematical Byways in Ayling, Beeling and Ceiling, Oxford University Press, 1984.
9. Bellman, R., Dynamic Programming, Princeton Univ. Press, Princeton, 1957.
10. Bellman, R., Gluss, B., On Various Versions of the Defective Coin Problem, Information and Control 4(1961)118–131.
11. Bošnjak, I., Tošić, R., Some New Results Concerning Two Counterfeit Coins, Univ. u Novom Sadu, Zb. Rad. Prirod.–Mat. Fak., Ser. Mat. 22, 1(1992)133–140.
12. Bošnjak, I., Some New Results Concerning Three Counterfeit Coins, Discrete Appl. Math. 48(1994)81–85.
13. Braun, J., The Counterfeiters of Lower Slobbovia, Amer. Math. Monthly 61(1954)472–473. [But see corrections by Carlitz and Selfridge, 62(1955)40–41.]
14. Cairns, S., Balance Scale Sorting, Paper P–736.
15. Cairns, S., Balance Scale Sorting, Amer. Math. Monthly 70(1963)136–148.
16. Cantor, D. G., Mills, W. H., Determining a Subset from Certain Combinatorial Properties, Canad. J. Math. 18(1966)42–48.
17. Chang, G., Hwang, F., Lin, S., Testing with Two Defectives, Discrete Appl. Math,. 4(1982)97–102.
18. Chang, G. J., Hwang, F. K., A Group Testing Problem on Two Disjoint Sets, SIAM J. Algebraic Discrete Methods 2(1981)35–38.
19. Christen, C., Optimal Detection of Two Complementary Coins, SIAM J. Alg. Discrete Methods, 4(1982)97–102.
20. Christen, C., Search Problems: One, Two or Many Rounds, Discrete Math. 136(1994)39–51.
21. Christen, C., Adaptive Versus Non–Adaptive Quantitative Detection, in: 3rd SIAM Conf. on Discrete Mathematics, Clemson, 1986.
22. Christen, C., A Fibonaccian Algorithm for the Detection of Two Elements, Publ. 341, Dépt. d'IRO, Université de Montréal, Montréal, 1980.
23. Christen, C., Hwang, F. K., Detection of a Defective Coin with Partial Weight Information, Amer. Math. Monthly 91(1984)173–179.
24. Chen, P. D., Hu, X. D., Hwang, F. K., A New Competitive Algorithm for the Counterfeit Coin Problem, Information Processing Letters 51(1994)213–218.
25. Collings, S. N. (editor), Puzzle Corner, Bull. Inst. Math. Appl. 20(1984), p. 94, Puzzle number 79, Unbalanced Coins II; p. 126, Solution; p. 153, Puzzle number 81, A Coulorless Corollary; pp. 184–185, Solution.
26. Descartes, B., The Twelve Coin Problem, Eureka, 13(1950)7, 20.
27. Dorfman, R., The Detection of Defective Members of Large Population, Ann. of Math. Statist. 14(1943)436–440.
28. Du, D., Hwang, F., Combinatorial Group Testing and its Applications, Applied Math Series, World Scientific Publications, Singapore, 1993.
29. Dyson, F. J., Note 1931 – The Problem of the Pennies, Math. Gaz., 30(1946)231–234.
30. Erdős, P., Rényi, A., On Two Problems of Information Theory, Publ. Hung. Acad. Sci., 8(1963)241–254.

31. Erdös, P., Frankl, P., Furedi, Z., Families of Finite Sets in which no Set is Covered by the Union of Two Others, J. Combin. Theory Ser. A 33(1982)158–166.
32. Erdös, P., Frankl, P., Furedi, Z., Families of Finite Sets in which no Set is Covered by the Union of r Others, Israel J. Math.51(1985)79–89.
33. Eves, D., Problem E712 – The Extended Coin Problem, Amer. Math. Monthly 53(1946)156; Solutions, E. D. Schell and J. Rosenbaum, Amer. Math. Monthly 54(1947)46–48.
34. Fine, N. J., Problem 4203 – The Generalized Coin Problem, Amer. Math. Monthly 53(1946)278; Solution, 54(1947)489–491.
35. Fixx, J., More Games for the Superintelligent, Warner Books, New York, 1972, p. 88.
36. Fujimura, K., Hunter, J. A. H., There's Always a Way, Recreational Math. Mag., 6(1961)67; editorial solution, 7(1962)53.
37. Fujimura, K., Another Balance Scale Problem, Recreational Math. Mag., 10(1962)34.
38. Fujimura, K., Another Balance Scale Problem, Recreational Math. Mag., 11(1962)42.
39. Gargano, L., Montuori, V., Setaro, G., Vaccaro, U., An Improved Algorithm for Quantitative Group Testing, Discrete Appl. Math. 36(1992)299–306.
40. Goodstein, R. L., Note 1845 – Find the Penny, Math. Gaz. 29(1945)227–229. [Erroneous solution, purporting to find dud among $(3^n - 2n + 3)/2$ coins.] Editorial note, Note 1930 – Addendum 30(1946)231, gives correct solution.
41. Grossman, H. D., The Twelve–Coin Problem, Scripta Math., 11(1945)360–361.
42. Grossman, H. D., Ternary Epitaph on Coin Problem, Scripta Math., 14(1948)69–71.
43. Guy, R. K., Nowakowski, R. J., ApSimon's Mints Problem, Amer. Math. Monthly 101(1994)358–359.
44. Guy, R. K., Nowakowski, R. J., Coin–Weighing Problems, Amer. Math. Monthly 102(1995)164–167.
45. Hammersley, J. M., A geometrical illustration of a principle of experimental directives, Phil. Mag. 39(1948)460–466.
46. Hao, F., The Optimal Procedures for Quantitative Group Testing, Discrete Appl. Math. 26(1990)79–86.
47. Hendy, M., The Retrial of the Lower Slobbovian Counterfeiters, Amer. Math. Monthly 87(1980)200–201. [But see 62(1955)40–41.]
48. Hu, X. D., Hwang, F. K., A Competitive Algorithm for the Counterfeit Coin Problem, to appear.
49. Hwang, F. K., A Tale of Two Coins, Amer. Math. Monthly 94(1987)121–129.
50. Hwang, F. K., Updating a Tale of Two Coins, in: Capobianco, Guan, Hsu and Tian, eds., Graph Theory and its Applications: East and West, New York Acad. Sci., (1989)259–265.
51. Hwang, F. K., Sos, V., Non–Adaptive Hypergeometric Group Testing, Studia Sci. Math. Hung. 22(1987)257–263.
52. Itkin, K., A generalization of the Twelve–Coin Problem, Scripta Math., 14(1948)67–68.
53. Karapetoff, V., The Nine Coin Problem and the Mathematics of Sorting, Scripta Math., 11(1945)186–187.
54. Katona G. O. H., Combinatorial Search Problems, A Survey of Combinatorial Theory, ed. J. N. Srivastava, North Holland, 1973, 285–308.

55. Katona G. O. H., Renyi and the Combinatorial Search Problems, Studia Sci. Math. Hung. 26(1991)363–376.
56. Lindström, B., On a Combinatory Detection Problem I, Magyar Tud. Akad. Mat. Kutató. Int. Kösl. 9(1964)195–207; MR 29:5750.
57. Lindström, B., On a Combinatory Detection Problem II, Publ. Hung. Akad. Sci. 1(1966)353–361.
58. Lindström, B., On a Combinatorial Problem in Number Theory, Canad. Math Bull. 8(1965)477–490.
59. Lindström, B., Determination of Two Vectors from the Sum, J. Combin. Theory 6(1969)402–407.
60. Lindström, B., On B_2–Sequences of Vectors, J. Number Theory 4(1972)261–265.
61. Liu Teng–Sun, To Weigh $5 + 2n$ coins of Two Different Weights in $4 + n$ times, J. Tianjin Univ., 1986 no. 4, 77–85.
62. Manvel, B., Counterfeit Coin Problems, Math. Magazine 50(1977)90–92.
63. Mead, D. G., The Average Number of Weighings to Locate a Counterfeit Coin, IEEE Trans. Inf. Theory 25(1979)616–617.
64. Mood, A. M., On Hotelling's Weighing Problem, Ann. Math. Statist., 17(1946)432–446.
65. Newbery, E. V., The Penny Problem, Note 2342, Math. Gaz. 37(1953)130.
66. Pelc, A., Detecting a Counterfeit Coin with Unreliable Weighings, Ars Combin. 27(1989)181–192.
67. Pippenger, N., Sorting and Selecting in Rounds, SIAM J. Comput. 16(1985)1032–1038.
68. Pyber, L., How to Find Many Counterfeit Coins, Graphs Combin., 2(1986)173–177.
69. Raine, C. W., Another Approach to the Twelve–Coin Problem, Scripta Math., 14(1948)66–67.
70. Rivest, R. L., Meyer, A. R., Kleitman, D. J., Winklmann, K., Spencer, J., Coping with Errors in Binary Search Procedures, J. Comput. System. Sci. (1980)396–404.
71. Robertson, J. A., Those Twelve Coins Again, Scripta Math., 16(1950)111–115.
72. Mauldon, J. G., Problem E3023, Amer. Math. Monthly 90(1983)645. Various solutions, 96(1989)254–258.
73. Robbins, D. P., Problem 6224, Amer. Math. Monthly 85(1978); Partial solution, Z. C. Motteler and A. Nijenhuis, Determining Heavy and Light Balls by Weighings, 87(1980)828–829.
74. Schell, E. D., Problem E651 – Weighed and Found Wanting, Amer. Math. Monthly 52(1945)42. Solution, M. Dernham, 52(1945)397.
75. Schwartz, B. L., Letter: Truth About False Coins, Math. Mag. 51(1978)254. [States that Schell told Michael Goldberg in 1945 that he had originated the problem.]
76. Shapiro, H. S., Problem 1399 – Counterfeit Coins, Amer. Math. Monthly 67(1960)82. Solution, Nathan J. Fine, 67(1960)697–698.
77. Smith, C. A. B., The Counterfeit Coin Problem, Math. Gaz. 31(1947)31–39.
78. Sobel, M., Binomial and Hypergeometric Group Testing, Studia Sci. Hung. 3(1968)19–42.
79. Söderberg, S., Shapiro, H. S., A Combinatory Detection Problem, Amer. Math. Monthly 70(1963)1066–1070.
80. Sterrett, A., On the detection of defective members of large populations, Ann. Math. Stat. 28(1957)1033–1036.
81. Stewart, D. A., The Counterfeit Coin, Proposed in L. I. Graham, Ingenious Mathematical Problems and Methods, Dover, 1959, pp. 37–38. Solutions, D. B. Parkinson

and Lester H. Green, pp. 196–198. [Problem appeared in the Graham Dial, October 1945.]
82. Tošić, R., Two Counterfeit Coins, Discrete Math., 46(1983)295–298.
83. Tošić, R., Three Counterfeit Coins, Rev. Res. Svi. Univ. Novi Sad, 15(1985)225–233.
84. Tošić, R., Four Counterfeit Coins, Rev. Res. Svi. Univ. Novi Sad, 14(1984)99–108.
85. Tošić, R., Five Counterfeit Coins, J. Stat. Plan. Inf., 22(1989)197–202.
86. Tošić R., On Two Counterfeit Coins Conjecture, unpublished.
87. Tošić R., Mašulović D., A Binary Search Problem on Graphs, Review of Research, Univ. Novi Sad, Zb. Rad. Prirod.–Mat. Fak., Ser. Mat. 24, 1(1994)231–243.
88. Unger, P., The Cutoff Point in Group Testing, Comm. Pure Appl. Math. 13(1960)49–54.
89. Winkelmann, K., An Improved Strategy for a Counterfeit Coin Problem, IEEE Trans. Inf. Theory 28(1982)120–122.
90. Withington, L., Another Solution of the 12–Coin Problem, Scripta Math., 11(1945)361–363.
91. Yao, Y. C., Hwang, F. K., A Fundamental Monotonicity in Group Testing, SIAM J. Disc. Math., Vol. 1, 2(1988)256–259.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 17–17

Homotopy in concurrent processes

Stefan Sokolowski

Institute of Computer Science, Polish Academy of Science
Abrahama 18, 81-825 Sopot, Poland
e-mail: S.Sokolowski@ipipan.gda.pl

Abstract. In theories of concurrent processes and in job scheduling there have been many attempts to introduce tools originating from algebraic topology; notably, homotopy or homology groups. Intuitively, fundamental (or first homotopy) group gives an account of the nature of "holes" in a topological space. In the realm of processes, such holes may correspond to forbidden configurations; e.g. where more than one process is within the same critical region.

However, a number of topological properties, technically necessary for the construction of the fundamental group, have no counterparts, or only artificial ones, in concurrent processes. Therefore, the payout from the homotopy considerations is rather limited.

I put forward to settle for homotopy cpo-s, which is much less than homotopy groups. As will be shown, the transition from vectors of processes to their homotopy cpo-s is functorial, preserves information about the "holes" and abstracts from inessential details. This renders the approach a potentially useful tool for investigating concurrency.

In the talk, I will not assume any familiarity of the audience with algebraic topology. I will explain motivations and will support the construction of homotopy cpo-s by examples.

Papers

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 1–8

Prolog-oriented Support of Calculations

Otakar Babka

Faculty of Science and Technology, University of Macau
P.O.Box 3001, Macau (via Hong Kong);
babka@umac.mo

Abstract The presented framework provides some logic programming features in environment of traditional languages. *Knowledge-based* subsystem in *Prolog* is attached to an engineering software system written in a conventional programming language (typically in C). Control of the computational process of the latter component is supported by the knowledge-based subsystem. In an attempt (i) to bring the environment of the computational component together with the logic programming environment and (ii) to increase the intelligence of it, an instantiation of variables is registered also in the computational component. This contributes to an *autonomous control* of the computational process. The extended engineering software system represents now not only a user friendly environment, but also an environment with an active and intelligent support of the user. Several practical applications were developed based on this framework. Two of them are reviewed in the paper.

1 Introduction

A *Prolog-oriented* knowledge-based *support* of engineering computations is studied in the paper. Since most engineering computations are realized in a conventional programming language (typically in C, for instance), it was necessary to solve the problem of how the computational component (written in C, for example) should cooperate with the supporting knowledge-based component, completed in Prolog. The developed framework solves this problem [1]. A collection of built-in tools [7] facilitates bilateral transfers of both data and control, introducing logic programming features to computational component.

The nature of both these components is different. In an effort to decrease these differences, as well as increase the intrinsic intelligence of the computational component, some important *properties of Prolog* were partially *modelled* in the environment of the conventional programming language (e.g., C language). The system can distinguish between *instantiated* and non-instantiated variables, for instance. So, a built-in *selective mechanism* can recognize potentially feasible procedures of the computational component at a given moment. In the next step, this selective mechanism compares all these potential candidates and selects the optimal one according to a given goal and several constraints. Since this decision-making process is substantially supported by the Prolog-oriented knowledge-based component, the decisions can be made mostly autonomously by the system. This is a typical feature of Prolog, and therefore, once it is modelled in the computational component and this component is supported by Prolog-oriented knowledge-based component, it can significantly contribute to a fundamental intelligence of the system.

2 Prolog-oriented support an selective mechanism

The task of the selective mechanism is to find a feasible and preferably *optimum routine* for a given *goal,* under given *criteria* and *constraints*. The principle of the mechanism is based on the inference process of knowledge-based systems. There are two steps used by the selective mechanism: Feasibility test and Selection.

Feasibility test.

Feasibility of the routines is tested and the *feasible routines* are inserted into the *Conflict set* for the time being. During the feasibility test, each input parameter of the tested routines is checked to determine whether an input value is known at that moment. A routine is feasible only if all input values have been defined and hence are instantiated in a given moment of computation.

Selection.

If the number of potential candidates in the Conflict set is two or higher, the optimum feasible routine of the Conflict set is selected according to the given *selective conditions*. These conditions are composed from the criteria and constraints (given by a user or left as default conditions). Maximizing the accuracy, or an upper limit of the machine time, can be chosen as conditions, for instance. During the selection the corresponding attributes of potential routines are compared with the given selective conditions and the best routine is chosen. The learning ability, employed by a case-based reasoning process [4], enables the utilization of experience gained from previous cases.

3 Autonomous control of computing

The Prolog-like selective mechanism can autonomously control a computing process, constructing an appropriate sequence of the called routines and other information sources, according to a given goal. In order to check the feasibility of routines, it is necessary to detect whether variables have already been instantiated; [1]. On the other

hand, it also opens up the possibility to define *invertible* programs integrating routines of different *orientations* [6].

Orientation
Let the routine r be given as $r(q_1, q_2, \ldots, q_n)$. The routine r can communicate with other objects by:

- *explicit* (i.e. formal) *parameters* $q_1, q_2, \ldots, q_n$;
- *implicit parameters* $q_{n+1}, q_{n+2}, \ldots, q_m$, $m \geq n$, which are global variables used as input, output, or input/output variables;
- *function value,* in the case where r is a function; for the purpose of the orientation issue, the name of the function, r, is incorporated among the (output) parameters.

The routine r models the (oriented) relationship of these parameters. The use of a parameter can be specified by its *mode*. There are three modes of parameters: input, output, and input/output. An assignment of the mode to each parameter defines the *orientation* O of this routine.

A Prolog predicate can usually correspond to several orientations. Such a predicate covering more than one orientation is called *invertible*. However, routines in conventional programming languages commonly are not invertible. One way to model some invertible relations is as follows:

(i) Collect several routines, modelling the same relationship among the same (or almost the same) parameters. They mostly (but not necessarily) correspond to different orientations.

(ii) Encapsulate this collection of similar routines in a family. The most significant differences among these routines (also called members of the family) are usually their orientations, but they can differ also in other properties (e.g. accuracy, used algorithm, and others). For a given purpose, the *selective mechanism* can select the appropriate member of the family, according to the needed orientation and other factors. Thus, despite the fact that conventional languages do not support invertibility, the invertible relationships are accomplished with the help of the families (collecting and encapsulating routines of various orientations) which are supported by the selective mechanism.

Two main features were mentioned above:

(a) The sequence of proper routines is composed autonomously. This increases substantially an intrinsic intelligence on a very general, domain-independent level.

(b) Similar routines, corresponding to different orientations, are encapsulated in the family, modeling an invertible relationship. So, this Prolog-like feature is

implanted in the environment of conventional programming languages. In addition, this approach also enables the invertibility of arithmetical computations, which is not usually possible in Prolog, since this is restricted by the built-in-predicate *"is"*.

4 Example

The features mentioned above are illustrated in the following application from mechanical engineering.

Individual girder

Suppose there exists an individual girder (Figure 1), with two defined relations:

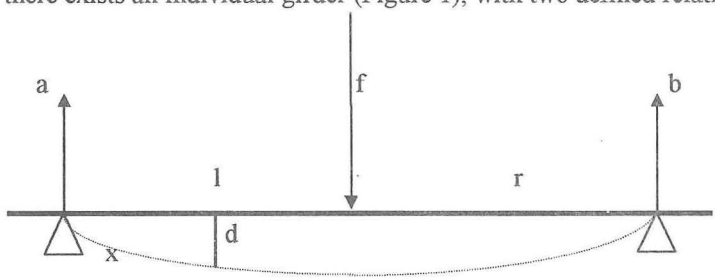

Figure 1: Individual girder

(i) Let there be a *family* of routines, represented by the *template*

force_relation (f, a, b, l, r),

expressing a relationship among force *f*, reactions *a*, and *b*, and distances *l*, and *r* of the individual girder. As mentioned previously, the family collects several routines expressing the same relationship and among the same parameters, however each of these routines can correspond to a different *orientation*. Each needed orientation is represented by one or more member routines. For instance, the call: *force_relation (F, a, b, L, R)* means an orientation, which for a given force *F* and both distances *L*, and *R*, will return both reactions *a* and *b* as results (lower-case identifiers indicate uninstantiated variables, capitals stand for instantiated objects). A different orientation could be represented by the call: *force_relation (f, A, B, L, r)*. It will return force *f* and right-hand distance *r*, for given reactions *A*, and *B*, and for left distance *L*. Since the selective mechanism can choose the appropriate routine among all members of the family (corresponding to various orientations), the above family is *invertible*.

(ii) Besides the relationship of forces and distances, also deformation of the strained girder is defined. Let the

deformation (I, E, G, d, x, F, A, B, S_a, S_b, l, r)

be the routine expressing the girder deformation d in the point defined by distance x, E is Young's modulus of elasticity, G is the shear modulus, I is the section moment of inertia, S_a and S_b are shifts of the supports (it can be caused by a deformation of an underlying girder), other parameters have the same meaning as above.

Set of girders

Both *force_relation* and *deformation* were defined corresponding to the individual beam only. With the help of the selective mechanism, these defined relationships also represent a sufficient apparatus for a set of girders G_1, G_2, G_3, and G_4 in Figure 2. For a given force f, the deformation (depression) of the whole set should be calculated.

The software system uses the routine *deformation* and several orientations of the routine *force_relation* (encapsulated in the family), defining deformation and forces of one individual girder only. The system did not receive any knowledge about sequencing of these routines for sets of girders. However, despite this fact, the presented mechanism can solve the task autonomously and with only minimal domain-specific information.

Calculation

The following sequence was designed by the selective mechanism.

$$[fr(G_1),\ fr(G_2),\ fr(G_3),\ fr(G_4),\ def(G_3),\ def(G_4),\ def(G_2),\ def(G_1)],$$

where, e.g., symbol $fr(G_1)$ means the call:

$$force\_relation(f,\ a_1,\ b_1,\ L_1,\ P_1)$$

and represents the calculation of the reactions a_1 and b_1 for the girder G_1. Similarly, symbol $def(G_3)$ means call of the routine *deformation* for calculation of a local depression on the girder G_3.

This achieved result is consistent with our intuition: it is obvious that (1) *force_relation* firstly should be applied to the upper girder G_1, then to G_2, and only then to the remaining ones, to define all forces and reactions. (2) Only after that, routine

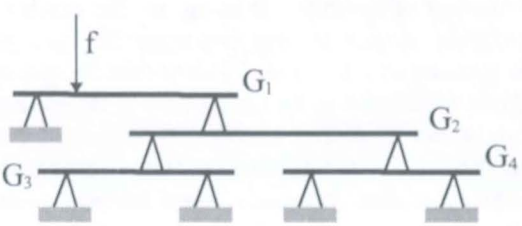

Figure 2: Set of girders

deformation can be applied, but in the reverse sequence, bottom-up, because of superposition of the depressions.

Recalculation
Invertibility can enable a simple recalculation. This recalculation is inevitable for an optimization of the design: Assume that all reactions were calculated, but the real maximum load weight of some actually used girder supports will probably be higher than previously calculated (in order to use only standardized supports, the girders of the same or next higher value will actually be used). Therefore, it would be suitable to recalculate this task and the admissible value of force *f,* as well as new values of deformations under these modified conditions. The modified task would need a considerably *modified sequence* of the calculation. With the help of the selective mechanism, the new *sequence* for recalculation is designed:

$$[\, fr(G_3),\ def(G_3),\ fr(G_4),\ def(G_4),\ fr(G_2),\ def(G_2),\ fr(G_1),\ def(G_1)\ ].$$

Notice that a different orientation of *force_relation* was used for the recalculation. The correct routine was (automatically) chosen inside the *force_relation* family.
Even a task as simple as the presented one can demonstrate that the selective mechanism can support a user considerably. The task would at least require the user's basic knowledge of both mechanics and the software system (furthermore the user's time, energy, concentration, and others, if the user should "manually" call the correct routines, in the correct order, and with the correct parameters). On the other hand, the presented selective mechanism can autonomously control the computing process and *without any specific knowledge about sequencing*. The mechanism needs only the input/output mode of parameters of all routines and other sources of information.

5 Implementation issue

The selective mechanism was implemented and applied in several versions. The choice of programming language was influenced by the language of the application, however a host environment of the C language was used predominantly for the implementation.

Implementing the presented concept, two issues are the most important: (i) how to register whether a variable is instantiated and (ii) how to implement invertible routines.

Instantiated variables
It was mentioned that the test of feasibility is based on the check whether or not all inputs are instantiated. This concept is very important for logic programming, but commonly not used in conventional languages. Notice that the concept of instantiated variables was adopted in a modified way for the purpose of the presented approach: the instantiated variable can arbitrarily change its value. Nevertheless, the instantiated and uninstantiated variables have to be distinguishable. How to recognise the instantiated variables is a considerable problem, especially when the host environment of some programming language is used. Several methods were used in an effort to solve this problem.

- One of the methods is based on the principle that a variable can be instantiated inside a procedure only. If this restriction is accepted, each variable returned by a procedure as an output parameter is instantiated. The list of all variables is maintained by each level of program, recording whether the variable was already instantiated.
- Another method indicates uninstantiated variables with the help of a special value, which was specifically chosen for each type. This value is assigned to a variable during the initialisation phase. The method could seem risky. Nevertheless, for a sensibly chosen value, the risk of confusion is noticeably less then 10^{-9} for the four byte word.
- Also the tagged approach was studied, distinguishing the variable according to an attached tag to the data structure.
- In addition, pre-processing of the source code of routines can be combined with the mentioned methods, especially the last one.

6 Conclusions

The presented concept is based mainly on the *selective mechanism*. Main features can be summarised as follows:

- Routines serving a similar purpose are grouped into *families*. All important properties of routines are stored in the family shell.
- The selective mechanism can choose the *optimum* routine in the family automatically.
- This mechanism can be used for automatic control of the calculation (*sequencing* of routines).
- The system can partially learn from its own history.
- The issue of *invertibility* was introduced, supported by a selective mechanism.
- With the help of the presented concept, a *consistent, autonomous, friendly, supportive and intelligent environment* can be built for a user.
- The concept can also contribute to *reliability* and *robustness* of software, (i) preventing undefined input parameters of routines, and (ii) checking the incorrectly designed output parameters that cannot return values (value parameters).

References

1. Babka, O., Cendelin, J. & Hajsman, V., Automatically Controlled Sequencing, 1995, in: *Proceedings of 2nd International Workshop on Learning in Intelligent Manufacturing Systems,* April 20-21, 1995, Budapest, pp. 549-565.
2. Genesereth, M.R. & Ketchpel, S.P., Software Agents, *Communications of the ACM,* July 1994, **37**, 48-53.
3. Harmon, P. & Hall, C., *Intelligent Software Systems Development,* John Wiley & Sons, 1993.
4. Kolodner, J, *Case-based Reasoning,* Morgan Kaufman Publ., San Mateo, CA, U.S.A., 1993.

5. Ras, Z.W. & Zemankova, M., *Intelligent Systems: State of the Art and Future Directions,* Ellis Horwood Limited, 1990.
6. Sickel, S., *Invertibility of Logic programs.* Technical Report No. 78-8-005. University of California, Santa Cruz, California, 1978.
7. Zahradnik, Z., *Prolog-oriented Knowledge-based Support of Engineering Activities,* Master's Thesis, in Czech, Faculty of Science, University of West Bohemia, Pilsen, 1990.

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 9–18

LASS - A Language for Agent-Oriented Software Specification

Mihal Badjonski, Mirjana Ivanović
Institute of Mathematics, Faculty of Science,
University of Novi Sad,
Trg Dositeja Obradovića 4,
21000 Novi Sad, Yugoslavia,
e-mail: {mihal, mira}@unsim.ns.ac.yu

Abstract: This paper presents a new agent-oriented programming language named *LASS*. *LASS* is aimed for agent-oriented programming in multi-agent systems. It enables the programming using new, agent-oriented concepts. Agents programmed with *LASS* can have deliberative properties (plans, intentions) as well as the reactive ones (behaviors). Agent can execute its plans and/or behaviors simultaneously. It can communicate with other agents. Some agents may be located in the same computer while other agents in the system may be located on the other computers connected with the Internet.

1 Introduction

Multi-agent systems ([21]) are a new and promising area in the field of distributed artificial intelligence (DAI), as well as in the mainstream computer science. These systems are compound of relatively autonomous and intelligent parts, called *agents*.

Agent-oriented programming languages are programming languages developed for the programming of agents. Agent-oriented programming (AOP) can also be seen as a post-object-oriented paradigm.

An advantage of the usage of agents in software development instead of objects stems from the primitives used for programming. AOP introduces new concepts such as mental categories, reactivity, pro-activeness, concurrent execution inside and between agents, communication, meta-level reasoning, etc.

This paper presents an AOP language named *LASS*. Agent programmed with *LASS* possesses *intentions*, *beliefs* and *plans* for its *public* and *internal services*. Besides deliberative properties, agent specified with *LASS* can behave reactively as well. *LASS* introduces the usage of *behaviors* - programming primitives enabling agent to react immediately when it is necessary. *LASS* enables powerful communication between agents which is based on agents' public services. Services are used similarly like remote procedure calls.

Most of the concepts in *LASS* are already seen in other programming languages. However, the usage of all of them in one computer language is unique.

LASS is intended for the programming of static agents in multi-agent system (MAS), but it can be also used as a general-purpose programming language. In [9] the most appropriate domains for AOP are given. Nevertheless, in every nontrivial software system, several crucial components may be identified whose

cooperative or competitive work determines the system's behavior. These components may be viewed as agents and the software system may be viewed as MAS. The examples of such systems are given in [1], [3] and [4]. Once the software system is identified as a MAS, *LASS* can be used for its implementation. Trivial software systems may be also programmed with *LASS,* using only one agent. However, the benefits that *LASS* provides are more evident when it is used for the programming of more complex systems.

This paper is organized as follows. A definition and features of computer agent and multi-agent system are given in the next section. The programming language *LASS* is presented in the third section. The fourth section compares *LASS* with other AOP languages. A conclusion is given in the last section.

2 Agents

Even if we restrict ourselves to computer science, a word 'agent' has many meanings.

In [21], agent is defined as:"... a hardware or (more usually) software based computer system that enjoys the following properties:

- *autonomy*: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
- *social_ability*: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
- *reactivity*: agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
- *pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative."

This definition does not specify the size of agents. They can be as big as expert system is and as small as the part of an application interface is. Agent can be static (permanently located in some computer) or mobile (moving across the computer network, such as Internet). The amount of agent's intelligence is also not specified. A collection of agent's definitions is given in [13].

Multi-agent system (MAS) is a system compound of at least two agents. An attractive feature of MAS is that alternative approach with one central programmable entity that controls and plans every action of every other entity is mostly much harder to use in the system's development. Sometimes, centralized approach is unachievable, while the MAS approach to system's development gives satisfying results. An example that demonstrates this feature is given in [14]. Suppose that all the citizens in some big city are without intelligence and are only able to execute the commands of the only one intelligent person in the city. It is hardly possible for that person to organize the arrival to work of every citizen in the city. If something goes wrong and some important street becomes closed for the traffic, the system will crash down. However, people do come to their work every day in every city. They do not use high-level intelligence to accomplish this task.

3 *LASS*

Every program written in *LASS* is intended for the specification of exactly one agent. If there are *n* agents in MAS, *n* programs in *LASS* will be written.

The main part of the *LASS*'s syntax and the description of the syntax categories are as follows.

```
program =
  'AGENT' agent_name ';'
  [known_agents_decl ';']
  [fact_types_def ';']
  [facts_decl ';']
  [public_services_decl ';']
  [private_services_decl ';']
  [behaviors_decl ';']
  [init_beliefs ';']
  [init_intentions ';']
  'END' agent_name
```

Program in *LASS* consists of eight optional parts. The agents that will communicate with the agent are specified in the first part. The agent can ask services from each of these agents and it can be asked for service by each one of them. If the agent will not communicate, program will not contain this part.

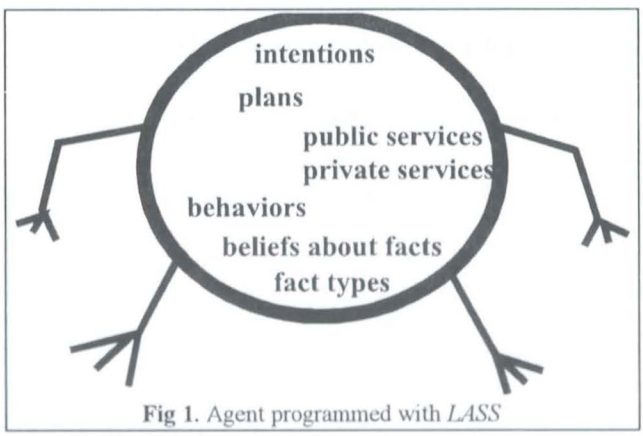

Fig 1. Agent programmed with *LASS*

The facts important to agent have to be declared. Before that, their types have to be defined in the second part of the program.

Besides public services, agent can perform its own, private services as well.

Agent can possess behaviors. They monitor the agent's beliefs and use them to activate or deactivate themselves.

At the beginning of its existence, agent can have initial beliefs about the facts in its environment. If agent does not have belief about some fact, it believes that fact has UNKNOWN value. It can also have initial intentions.

```
known_agents_decl =
  'KNOWN' 'AGENT' agent_decl {';' agent_decl}
```

```
agent_decl = agent_name ':' (internet_adr | 'LOCAL')
```

There are two types of agents from the viewpoint of an agent: the agents located on the same machine where the agent is located and the agents located on the remote machines.

```
fact_types_def = 'FACT' 'TYPE' ftype_def {';' ftype_def}

ftype_def = ftype_name '=' ftype

ftype = prim_type | record_type | array_type
```

Fact can be of a primitive type (standard type), a record or an array.

```
facts_decl = 'FACTS' fdecl {';' fdecl}

fdecl = fact_names ':' ftype_name

fact_names = fname {',' fname}
```

Facts are used as variables in traditional languages. Besides user-defined facts, meta-level facts CURRINT and CURRBEH are also available. They contain information about current intentions and active behaviors.

```
public_services_decl =
  'PUBLIC' 'SERVICES' serv_decl {';' serv_decl}

serv_decl =
  serv_name '(' [ par_decl {';' par_decl} ] ')' ';'
  ('ALWAYS' | 'WHEN' bool_expr ';')
  'PLAN' body
  'END' serv_name
```

Service can contain parameters. Parameter can be of INPUT type or INTPUT-OUTPUT type (VAR parameter). par_decl represents the declaration of parameter(s). It consists of parameters' names optionally preceded with the word VAR and followed by the type of parameter(s). Service will not be performed if bool_exp in WHEN condition is not satisfied. Every service has its plan for its execution.

```
bool_expr =
  'TRUE' |
  'FALSE' |
  'KNOWN' '(' fname ')' |
  '(' bool_expr ')' |
  test_service |
  term relation term |
```

```
  'NOT' bool_expr |
  bool_expr ('AND' | 'OR') bool_expr
```

test_service is a special type of private service that returns logical value TRUE or FALSE after its execution.

```
body = action {';' action}

action =
  communicative_action|
  service_action |
  loop_action |
  cond_action |
  modify_fact_action|
  input_output_action
```

LASS supports standard constructs from procedural programming languages such as loop_action, cond_action, modify_fact_action and input_output_action. In addition *LASS* possesses special communicative primitives characteristic for the AOP languages.

```
communicative_action =   ask_service_wait | ask_service

ask_service_wait =
  'SENDWAIT' serv_name '(' [params] ')'
  'TO' agent_name
  'REPORT' 'IN' rep_fact_name

ask_service =
  'SEND' serv_name '(' [params] ')'
  'TO' agent_name
  'STATUS' 'IN' stat_fact_name
```

Communication is used when some service is asked from a local or remote agent. Agent that asks other agent's service may stop the execution of the actions' sequence while remote service is being performed or it may continue to perform its actions. Agent can have several intentions and/or behaviors active simultaneously. If it uses remote service with wait, only one plan/behavior will be paused, while other will continue to execute. Report can have values: 'DONE' or 'DENIED'. Status of the service can be: 'DENIED', 'EXECUTING' or 'DONE'.

```
service_action = service_wait | service
```

Agent may execute its own service in two ways. The plan or behavior that invoked the service may continue to execute simultaneously with the new service or it may wait until the service finish its execution. Like with remote services, service is accompanied with report (service_wait) or status information (service).

```
private_services_decl =
  'PRIVATE' 'SERVICES' pri_serv_decl {';' pri_serv_decl}

pri_serv_decl = test_serv_decl | serv_decl

behaviors_decl = 'BEHAVIORS' beh_decl {';' beh_decl}

beh_decl =
  beh_name ';'
  [ 'PRIORITY' integer ';' ]
  'ACTIVE' 'WHILE' bool_expr
  'BEGIN' body 'END' beh_name
```

Behavior's activation depends on the truth value of `bool_expr` defined in `'ACTIVE''WHILE'` part of the declaration. When `bool_expr` is true, the behavior will be active. There can be several behaviors active at the same time. However, only the active behaviors with the highest priority (lowest `integer` number) are executed, while other active behaviors are paused. By default, behavior has highest level priority.

```
init_beliefs = 'BELIEFS' 'INITIALIZATION' body
```

`body` should be used for the assignment of values to various facts. Facts that are not initialized have special value: `UNKNOWN`.

```
init_intentions =
  'INITIAL' 'INTENTIONS' intention {';' intention}

intention = serv_name '(' [params] ')'

params = par {',' par}
```

Intentions are the list of services that have to be performed.

All actions specific to all particular problem domains cannot be covered with any AOP language. Therefore *LASS* does not contain actions which would be used in services such as: "open_the_door", "pick_up_the_box" (robot's actions), "send_email_to_boss", "check_the_web_site", "get_the_headlines" (actions of the personal digital assistant), etc.

In order to use *LASS* in every particular MAS domain, domain specific services should be grouped in the special agent named *SYSTEM*. *SYSTEM* should be able to provide its services to every local *LASS* agent. From the viewpoint of any *LASS* agent, it would act like *LASS* agent. However, it would not be programmed with *LASS*. It would provide domain specific actions represented as *LASS* services. It would serve as an effector common to all local agents.

To every hardware component in MAS is attached exactly one *SYSTEM* agent.

4. Related Work

The first AOP language that uses mental categories is AGENT0 ([16]). Agents programmed in AGENT0 have their initial beliefs and capabilities to perform private and communicative actions. The main part of AGENT0 program are the commitment rules. Each commitment rule determines the new commitments and other mental changes that will occur if particular message is received. Types of messages are chosen from the speech acts theory ('inform', 'request', and 'unrequest'). Agents programmed with AGENT0 are synchronized with a common clock. The language is loosely coupled with modal temporal logic. This logic is used for the specification of the language.

Unlike AGENT0, *LASS* is intended for practical usage. AGENT0's purpose was to introduce new concepts in an elegant manner. *LASS* is not bound to any logic. It uses some procedural constructs and its expressive power is greater than it is in AGENT0. Agents' communication in *LASS* is inspired with remote procedure call, while agents in AGENT0 use speech acts. Communication with speech acts reminds on human-like communication, but it is less efficient and less convenient than the communication used in *LASS*. Agents programmed in *LASS* do not use clocks and references to time points to synchronize their actions. `ask_service_wait` can be used for synchronization. While AGENT0 possesses some elements of logic programming, *LASS* is more oriented to procedural constructs.

PLACA ([17], [18]) is the descendant of AGENT0. PLACA introduces planning capabilities of agents. Agent in PLACA uses plans to achieve the desired state of the world.

Agents in *LASS* also use plans, but plans cannot be generated at run-time as they can be in PLACA. Whereas PLACA is descendant of AGENT0, its comparison with *LASS* is similar to the comparison of AGENT0 and *LASS*.

In Concurrent MetateM ([11], [12], [20]) MAS is specified with the logic. The logic is modal and linear temporal. Specification of MAS is directly executed.

Concurrent MetateM is only in experimental stage and so far it has no common features with *LASS*.

A different approach to MAS programming is proposed in [15]. In HOMAGE, the language for agent specification has two levels. The lower level uses objects of Java, Common Lisp and C++ instead of mental categories. Higher level contains constructs for organization of objects from the lower level into agent's program. Agents in HOMAGE communicate and received messages are handled with rules similar to those in AGENT0 and PLACA.

LASS does not allow the use of other languages. However, two types of primitives in *LASS* can be identified. Primitives that are specific for AOP languages (communicative actions, services, plans, intentions, behaviors, etc.) correspond to higher level in HOMAGE. Primitives inherited from procedural languages

(`loop_action`, `cond_action`, `modify_fact_action`, `input_output_action`, ...) correspond to lower level in HOMAGE.

The AOP language with the greatest influence on our research is AgentSpeak ([19]). Creators of AgentSpeak aimed to join object oriented programming and MAS concepts such as: mental categories, reactive and proactive properties, distribution over wide area network, real-time response, communication with speech acts, concurrent execution of plans in and between agents and meta-level reasoning.

LASS is very similar to AgentSpeak. The main difference is in communication. We believe that *LASS* introduces more powerful communicative primitives than those existing in languages enlisted above. Speech acts can be easily implemented in *LASS* using services. Services act like remote procedure calls and enable more efficient and simpler transfer of information.

None of the above languages possess such a powerful construct for agent's reactivity such as behaviors. Behavioral approach to artificial intelligence is developed at MIT. Its creator, R. Brooks ([5], [7], [8]), has developed many simple robots that are able to perform complex tasks. Brooks proposes Subsumption Architecture for the organization of behaviors. Behaviors in *LASS* are organized in the similar manner.

The combination of deliberative agent architecture and behaviors is proposed in [10], for the programming of animated agents in computer animation.

Most of the concepts used in *LASS* are already seen in other programming languages. The significance of *LASS* is in the inclusion of all these concepts into one programming language.

5. Conclusion

This paper presents a new programming language called *LASS*. *LASS* enables programming using agent-oriented concepts. Usage of these concepts in computer programming should make it more convenient and easier for wider population of people.

Besides other concepts, *LASS* provide the usage of behaviors. Behaviors can be used for programming of agent's reactive features.

LASS and/or ideas on which *LASS* is based were conceptually applied in the previous research of the authors. A multi-agent system in agricultural domain is described in [1] and in [3]. Intelligent tutoring system compound of agents is described in [4] and [6]. Personal digital assistant programmed in *LASS* is given in [2].

LASS is not yet available for practical use, but authors are working on its implementation. *LASS* is being implemented in Java. The implementation will be done in three steps. The first step is aimed for the creation of the Java's objects, that will be used for the implementation of some parts of *LASS*. This stage of *LASS*' implementation as well as the use of the created objects in implementation of three MASs might be included in a master theses of the first author. In the second stage all

remaining Java's objects that are necessary for *LASS* implementation will be created. Finally, in the third stage a translator from *LASS* to Java will be made.

LASS is suitable for agent-oriented software engineering. This approach to software engineering has several advantages. It facilitates the usage of divide-and-conquer strategy. It also enables the exploitation of parallelism. Software system can be easily deployed on the wide area network and executed on several machines simultaneously. Whereas agents are encapsulated entities, the system behaves robustly when the addition of new agents or the modification or removal of existing ones occurs.

Acknowledgment

Authors are grateful to professor Hans-Dieter Burkhard for his valuable help.

References

[1] M. Badjonski, M. Ivanović, "An Application of Multi-agent Theory in Agriculture", Proceedings of *IEEE First International Conference on Intelligent Processing Systems (IEEE ICIPS)*, Beijing, China, 1997, to appear.

[2] M. Badjonski, M. Ivanović, "Personal Digital Assistant Programmed in *LASS*", *Proceedings of Panonian Applied Mathematical Meetings (PAMM)*, Balaton, Hungary, 1997, to appear.

[3] M. Badjonski, M. Ivanović, "Multi-agent System for Determination of Optimal Hybrid for Seeding", Proceedings of EFITA '97 - First European Conference for Information Technology in Agriculture, Copenhagen, Denmark, June 15 - 18, 1997, to appear.

[4] M. Badjonski, M. Ivanović, Z. Budimac, "Possibility of using Multi-Agent System in Education", *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, Orlando, Florida, USA, October 12-15, 1997, to appear.

[5] R. A. Brooks, "Intelligence without Reason", *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pp 569-595, Sydney, Australia, 1991.

[6] M. Badjonski, M. Ivanović, Z. Budimac, "Possibility of using Multi-agent System in Education", Proceedings of *IEEE First International Conference on Intelligent Processing Systems (IEEE ICIPS)*, Beijing, China, 1997, to appear.

[7] R. A. Brooks, "Intelligence without Representation", *Artificial Intelligence*, 47:139-159, 1991.

[8] R. A. Brooks, "A robust layered control system for a mobile robot", *IEEE Journal of Robotics and Automation*, 2(1):14-23, 1986.

[9] H. D. Burkhard, "Agent-Oriented Programming for Open Systems", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 291-306.

[10] M. Costa, B. Feijo, "Agents with Emotions in Behavioral Animation", *Computers & Graphics*, Vol. 20, No 3, pp. 377-384, 1996.

[11] M. Fisher, "Representing and Executing Agent-Based Systems", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 307-323.

[12] M. Fisher, "A Survey of Concurrent MetateM - the language and its applications", *Temporal Logic - Proceedings of the First International Conference (LNAI Volume 827)*, pp. 480-505, Springer-Verlag, July 1994.

[13] S. Franklin, A. Graesser, "Is it and Agent, or just a Program?: A Taxonomy for Autonomous Agents", Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages, ECAI '96, Budapest, Hungary, pp. 193-206.

[14] L. Glicoes, R. Staats, M. Huhns, "A Multi-Agent Environment for Department of Defense Distribution", *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, Vol 1042, Springer-Verlag, 1996, pp. 53-84.

[15] A. Poggi, G. Adorni, "A Multi Language Environment to Develop Multi Agent Applications", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages, ECAI '96*, Budapest, Hungary, pp. 249-261.

[16] Y. Shoham, "Agent-Oriented Programming", *Artificial Intelligence*, 60(1):51-92, 1993.

[17] S. R. Thomas, "The PLACA Agent Programming Language", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 356-370.

[18] R. S. Thomas, "PLACA, an Agent Oriented Programming Language", Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA 94305, August 1993. (Available as technical report STAN-CS-93-1487).

[19] D. Weerasooriga, A. Rao, K.Ramamohanarao, "Design of a Concurrent Agent-Oriented Language", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 386-401.

[20] M. Wooldridge, :A Knowledge-Theoretic Semantics for Concurrent MetateM", *Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages, ECAI '96*, Budapest, Hungary, pp. 279-293.

[21] M. Wooldridge, N. R. Jennings, "Agent Theories, Architectures, and Languages: A Survey", *Intelligent Agents*, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, 1994, pp. 1-39.

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE *LIRA '97*
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1–4, 1997, PP. 19–25

THE CONCEPT OF LOGIC PROGRAMMING LANGUAGE BASED ON THE RESOLUTION THEOREM PROVER

Ivana Berković, Ph. D., Assist., Petar Hotomski, Ph. D., Prof.

Abstract - *The systems of automated reasoning are put into the base for development of the descriptive languages for logical programming in this paper. The determinate resolution system for automated theorem proving is especially put into the base of prolog-like language, as the surrogate for the concept of negation as definite failure. This logical complete deductive base is used for building a new descriptive logical programming language. The language enables eliminating the defects of PROLOG-system (the expansion concerning Horn clauses, escaping negation treatment as definite failure), keeping the main properties of PROLOG-language and possibilities of its expansions.*

1. INTRODUCTION

The methods and techniques of automated reasoning have a wide variety of its applications. The computer can be used to assist in various reasoning aspects, as well as it has been used to assist in certain numerical aspects of mathematics. In this sense, "automated reasoning is concerned with the discovery, formulation, and implementation of concepts and procedures that permit the computer to be used as a reasoning assistant", (Wos, [13]).

The developing of automated reasoning results into the developing of logic programming languages. The advantages of changing one system for automated reasoning by the other are described in this paper. The determinate resolution system for automated theorem proving ADT (OL-resolution with marked literals) is especially put into the base of prolog-like language, as the surrogate for the concept of the negation as definite failure (SLDNF resolution) in PROLOG.

2. THE RULE OF ORDERED LINEAR RESOLUTION AS THE FOUNDATION OF AUTOMATIC THEOREM PROVING

The most popular method for automatic theorem proving is the resolution method, which is discovered by J. A. Robinson in 1965 ([1], [5], [14]). Since 1965., many resolution forms and techniques are developed because the pure resolution rule has been unable to handle complex problems, ([7]).

The general automatic method for determining if a theorem (conclusion) **A** follows from a given set of premises (axioms) **F**:

$$\mathbf{F} \vdash \mathbf{A}.$$

Each formula will be transformed to the ***clauses form***. The ***clauses*** have the form: $L_1 \vee L_2 \vee \ldots \vee L_m$, where L_i are literals. The symbol for disjunction is: $\vee$.

The *literals* L_i have the form: $P(t_1,t_2, \dots , t_n)$ or $\neg P(t_1,t_2, \dots , t_n)$, where P is predicate symbol, t_i is term, $\neg$ is negation. The literal $P(t_1,t_2, \dots , t_n)$ is called positive literal, the literal $\neg P(t_1,t_2, \dots , t_n)$ is called negative literal.

Resolution method is a syntactic method of deduction. *Reduction ad absurdum* is in the basis of resolution method:

$$F \vdash A \quad \text{iff} \quad F \cup \{ \neg A \} \vdash \text{contradiction} .$$

Resolution rule will be applied on the set of clauses - axioms which was expanded by negating the desired conclusion in clause form.

Ordered Linear (OL) resolution rule with marked literals ([6]) increases efficiency and doesn't disturb completeness of pure resolution rule.

The generating process of OL-resolvent from central clause (**d1**) and auxiliary clause (**d2**):

1. Redesignate variables (without common variables in the clauses).
2. Determine universal unificator Θ for last literal of **d1** and **k**-literal (**k=1,2,...**) of **d2** (if it exists for some k, else it is impossible to generate OL-resolvent for specification clauses).
3. Create resolvent with marked last literal in $d1\Theta$ and add the rest of clause $d2\Theta$ without **k**-literal ($d1\Theta$ and $d2\Theta$ are clauses, which were formed by universal unificator Θ applied on **d1** and **d2**, respectively).
4. Eliminate identical non-marked literals and tautology examination (tautologies are not memorized).
5. The Shortening Operation (delete all ending marked literals)
6. The Compressing Operation (delete the last non-marked literal, which is complemented in relation to negation, with some marked literal for unificator λ).
7. Repeat steps: 5 and 6 until the empty clause is got, or the Compressing Operation is not applied on the last non-marked literal.

The rule of OL-resolution with marked literals is separated in two parts: ***in-resolution*** and ***pre-resolution***. The steps: 1 - 5 are represented in-resolution. The steps: 6 - 7 are represented pre-resolution. Mid-resolvents are the products of in-resolution and without their memorizing, the completeness of the method can be lost. This modification of Ordered Linear resolution rule is served as the base for development of the system for automatic theorem proving ADT.

3. THE SYSTEM FOR AUTOMATIC THEOREM PROVING ADT

ADT is a system for automatic theorem proving ([2]), which is implemented on PC - computer by Pascal programming language. The system ADT is based on the resolution rule. The rule of Ordered Linear Resolution with marked literals presents the system base. The system is developed at Technical Faculty "Mihajlo Pupin" in

Zrenjanin. ADT is projected for scientific - researching, teaching and practical purpose.

The system ADT disposes three search strategies: breadth-first, depth-first and their combination. The first and the second strategy are common blind search procedures. The third blind search procedure is constructed as their combination. The system ADT permits comparisons of strategies. Numerously experiments with ADT system are shown that depth-first strategy has the most efficiency. In depth-first search, a new node is generated at the next level, from the one current, and the search is continuing deeper and deeper in this way until it is forced to backtracking.

The main characteristics of ADT system:
- This system presents a complete logical deductive base: *the clauses-set is unsatisfied (contradictory) iff the empty clause is generated by finite use of the resolution rule*. So, the proof of conclusion **A** is completed ($\mathbf{F} \vdash \mathbf{A}$) when the empty clause is generated by the resolution from clauses-set $\mathbf{F} \cup \{\neg \mathbf{A}\}$.
- Besides the theoretical completeness of the system, it has the satisfying practical efficiency limited by the space-time computer resources.
- The first-order logic is the form of representation in ADT system (each formula is transformed into the clause form). This deductive base has no restriction in Horn clause (expansions concerning Horn clauses) and it allows the logical treatment of negation (escaping negation treatment as a definite failure).

Therefore, the system of automated reasoning ADT is put into the base for development of the descriptive language for logic programming. This logical complete deductive base is used for building a new descriptive logical programming language.

4. THE CONCEPT OF LOGIC PROGRAMMING LANGUAGE BASED ON ADT SYSTEM

Many logic programming languages have been implemented, but PROLOG is the most popular language and useful for solving many problems.

PROLOG as a logic-oriented language ([4], [8], [9], [11]) contains a resolution-based theorem-prover (PROLOG-system). The theorem-prover in PROLOG appears with the depth-first search approach. It uses the special resolution rule: ***SLDNF*** (Linear resolution with Selection function for Definite clauses and Negation as Failure).

The first-order predicate logic is the form of representation in PROLOG. PROLOG-program is a set of sentences. Every sentence is finished by full stop. Program in PROLOG consists of axioms (rules, facts) and a theorem to be proved (goal). The axioms are restricted in ***Horn clauses*** form.

Horn clauses ([9]), are clauses with at most one positive literal.

The ***rules*** have the form:

G :- T_1, T_2, ... , T_n.

where **G** is positive literal and **T_j (j=1,2,...,n)** are literals (positive or negative). The symbol for conjunction is: , . The element **G** is presented head of the rule. The elements **T_j (j=1,2,...,n)** are presented body of the rule. The separator **:-** corresponds to implication (⇐). The symbol for negation is: **not**.

The ***facts*** have the form:

G.

where **G** is positive literal.

The ***goals*** (questions) have the form:

?- T_1, T_2, ... , T_n.

where **T_i (i=1,2,...,n)** are literals.

Practically, programming in PROLOG is restrictive in a subset of first-order logic. Horn clauses are represented the first defect of PROLOG. The concept of negation as definite failure is represented the second defect of PROLOG.

An other approach to logic programming is implementation a new deductive concept. The determinate system ADT for automated theorem proving is especially put into the base of prolog-like language, as the surrogate for the concept of negation as definite failure. This logical complete deductive base is used for building a new descriptive logic programming language.

The first-order logic is the form of representation in ADT system, too. But, this system has not restriction in Horn clauses. It appears with ***clauses***. The program on logic language based on the ADT system is a set of sentences (clauses). There are three kinds of sentences: rules, facts and goals. Every sentence is finished by full stop.

The ***rules*** have the form:

G_1, G_2, ... , G_m :- T_1, T_2, ... , T_n.

where **G_i (i=1,2,...,m)** and **T_j (j=1,2,...,n)** are literals (positive or negative). The symbol for conjunction is: , . The elements **G_i (i=1,2,...,m)** are presented head of the rule. The elements **T_j (j=1,2,...,n)** are presented body of the rule. The separator **:-** corresponds to implication (⇐). The symbol for negation is: ~ .

The ***facts*** have the form:

G.

where **G** is literal (positive or negative).

The ***goals*** (questions) have the form:

?- T_1, T_2, ... , T_n.

where **Ti (i=1,2,...,n)** are literals (positive or negative).

The rules and facts (axioms) are presented by auxiliary clauses. The goal (central clause) is negating the theorem to be proved. Symbol **?-** in goal is the substitution for negation. The execution procedure is ADT system based on OL-resolution with marked literals. This formulation enables eliminating the defects of PROLOG-system.

The logic programming language PROLOG and the logic programming language based on ADT system are compared.

PROLOG rules and facts do not allow the explicit statement of negative information. But, the declarative syntax of the logic programming language based on ADT system allows the expression of negative information in rules and facts. Also, it is possible to construct the rule with more than one element.

Example 1.

The problem of trying to formulate sentence:

"Alice likes whatever Queen dislikes, and dislikes whatever Queen likes." into PROLOG form, ([10]). The represcntations

```
likes(alice,X1) :- not likes(queen,X1).
not likes(alice,X1) :- likes(queen,X1).
```

are illegal in PROLOG because the second rule has a negation in head (it isn't Horn clause). It is possible to solve the problem by trick - using a modified predicate likes, and expressing the statement as:

```
likes(alice,X1,true) :- likes(queen,X1,false).
likes(alice,X1,false) :- likes(queen,X1,true).
```

The expansion concerning Horn clauses on the logic programming language based on ADT system has the possibilities to express the statement as:

```
likes(alice,X1) :- ~likes(queen,X1).
~likes(alice,X1) :- likes(queen,X1).
```

PROLOG-system has the negation defect, ([7]). This defect is corrected in ADT system. It can be illustrated by the following example.

Example 2.

Program in PROLOG:

```
vegetarian(tom).
vegetarian(ivan).
smoker(tom).
likes(ana,X1) :- not (smoker(X1)), vegetarian(X1).
```

PROLOG-system gives unconnected answers on the following questions:

?- likes(ana,X1).
no
?- likes(ana,ivan).
yes

If the last clause is now:
likes(ana,X1) :- vegetarian(X1), not (smoker(X1)).

PROLOG-system gives wrong answers on the following questions:
?- likes(ana,X1).
X1=ivan
?- likes(ana,ivan).
yes

These answers are incorrect because we have not data about Ivan and smoking. We don't know is Ivan a smoker or not. The correct answer will be: "I don't know".

The program in logic programming language based on ADT system:
vegetarian(tom).
vegetarian(ivan).
smoker(tom).
likes(ana,X1) :- ~ smoker(X1), vegetarian(X1).

ADT-system gives answers on the following questions:
?- likes(ana,X1).
Success=0
The proof isn't completed
?- likes(ana,ivan).
Success=0
The proof isn't completed

When the last clause is:
likes(ana,X1) :- vegetarian(X1), ~smoker(X1).
ADT system also gives the correct answers: **"Success=0, the proof isn't completed"**.

In fact, ADT system generates resolvents, but can not complete the proof with depth-first strategy. In this system is escaped the treatment of negation as definite failure.

The concept of logic programming language based on ADT system allows eliminating of endless branches, recursion using and works with structures and lists, as well as PROLOG. It is presented in some concrete examples, ([3]).

5. CONCLUSION

Completeness and universality of the resolution method, as the base of ADT system, enables it to be applied as the deductive base of prolog-like language. The relationship between programming language based on ADT system and programming language PROLOG are emphasized. The logic programming language based on ADT system enables eliminating the defects of PROLOG-system (the expansion concerning Horn clauses, escaping negation treatment as definite failure, eliminating of endless branches), keeping the main properties of PROLOG-language and possibilities of its expansions.

REFERENCES

[1] Barr A., Cohen P.R., Feigenbaum E.A., *The Handbook of Artificial Intelligence*, Vol.I,II,III, Heuris Tech Press, W. Kaufmann, Inc., California 1982.

[2] Berković I., *Variable Searching Strategies in the Educationally Oriented System for Automatic Theorem Proving*, M.Sc. Thesis, Technical Faculty "Mihajlo Pupin", Zrenjanin, 1994. (in Serbian)

[3] Berković I., *The Deductive Bases for the Development of the Descriptive Languages for the Logical Programming*, Ph.D. Thesis, Technical Faculty "Mihajlo Pupin", Zrenjanin, 1997. (in Serbian)

[4] Bratko I., *PROLOG Programming for Artificial Intelligence*, Addison-Wesley Publ. Comp. 1986.

[5] Gevarter W.B., *Intelligent Machines*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[6] Hotomski P., Pevac I., *Mathematical and Programming Problems of Artificial Intelligence in the Field of Automatic Theorem Proving*, Naučna knjiga, Belgrade, 1991. (in Serbian)

[7] Hotomski P., *Systems of Artificial Intelligence*, Technical Faculty "Mihajlo Pupin" Zrenjanin, 1995. (in Serbian)

[8] Malpas J., *PROLOG: A Relational Language and Its Applications*, Prentice-Hall International Inc., 1987.

[9] Pereira F.C.N., Shieber S.M., *PROLOG and Natural - Language Analysis*, CLSI, Leland Stanford Junior University, 1987.

[10] Subrahmanyam P.A., *The "Software Engineering" of Expert Systems: Is Prolog Appropriate?*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, november 1985., pp. 1391-1400

[11] Tošić D., Protić R., *PROLOG by Examples*, Tehnička knjiga, Belgrade, 1992. (in Serbian)

[12] Van Roy P., Despain, A.M., *High-Performance Logic Programming with the Aquarius Prolog Compiler*, IEEE Computer, january 1992., pp. 54-68

[13] Wos L., *Automated Reasoning*, The American Mathematical Monthly, Vol. 92, No. 2, february 1985., pp. 85-93

[14] Wos L., Overbeek R., Lusk E., Boyle J., *Automated Reasoning: Introduction and Applications*, Prentice-Hall, 1992.

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 27–34

Regular expression star-freeness is PSPACE-complete

László Bernátsky

Department of Computer Science, Attila József University
Aradi vértanúk tere 1., H-6720 Szeged, Hungary
Tel.: 36-62-454-287
e-mail: benny@inf.u-szeged.hu

Abstract. Star-free languages form an important subclass of regular languages: they are the ones that can be obtained from the finite languages by a finite number of applications of the operations of union, complement and product. By Schützenberger's famous theorem [7], a regular language is star-free if and only if its syntactic monoid is aperiodic, or equivalently, if it is recognized by an aperiodic DFA. Jacques Stern [8] proved that the problem of deciding whether a DFA is aperiodic is **Co-NP**-hard and belongs to **PSPACE**. Sang Cho and Dung T. Huynh [2] strengthened Stern's result by showing that this problem is in fact **PSPACE**-complete. Here we prove that the problem of deciding if a regular expression denotes a star-free language is **PSPACE**-complete.

1 Definitions and preliminary facts

The set of nonnegative integers is denoted $\mathbf{N}$, and ω stands for the set of positive integers. For $n \in \mathbf{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$, so that $[0]$ is another name for the empty set $\emptyset$.

Suppose that A and B are sets, $A' \subseteq A$, $B' \subseteq B$, and $\rho \subseteq A \times B$ is a relation from A to B. We write $A'\rho B'$ if there exist $a \in A'$ and $b \in B'$ with $a\rho b$. The image of A' under ρ is denoted $A'\rho$. When $A' = \{a\}$ is a singleton, we write $a\rho$ instead of $A'\rho$.

We denote by A^* the set of all finite words over A including the empty word ϵ, while A^+ stands for $A^* \setminus \{\epsilon\}$. The set A^ω is the collection of all infinite words over A. The length of a finite word $u \in A^*$ is denoted $|u|$, and the ith letter of a finite or infinite word $w \in A^* \cup A^\omega$ is denoted w_i. Thus, any finite word $u \in A^*$ can be written as $u_1 u_2 \ldots u_{|u|}$, where each u_i is an element of A.

1.1 Finite automata

Most of our automata-theoretical notations and definitions are adopted from [3].

A **(nondeterministic) finite automaton** (NFA) is represented as a 5-tuple $\mathcal{A} = (Q, \Sigma, \tau, I, F)$, where

- Q is the finite set of states,
- Σ is the input alphabet,
- $\tau : \Sigma \to \mathrm{P}(Q \times Q)$ is the transition function,
- $I \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of final states.

Note that for each input symbol $\sigma \in \Sigma$, $\tau(\sigma)$ is a binary relation on Q, called the **relation induced by σ in the automaton $\mathcal{A}$**. We prefer the notation $\sigma_{\mathcal{A}}$ to $\tau(\sigma)$. When $u \in \Sigma^*$ is an input word, $u_{\mathcal{A}}$ denotes the **relation induced by u in $\mathcal{A}$**.

The automaton $\mathcal{A}$ can be visualized as a directed graph with vertices Q, and edges labeled by input symbols in Σ. Motivated by this point of view, we shall sometimes denote the relation $u_{\mathcal{A}}$ by $\xrightarrow{u}_{\mathcal{A}}$. Then $q \xrightarrow{u}_{\mathcal{A}} q'$ means that there is a directed u-labeled path from vertex q to vertex q'.

The language $L(\mathcal{A})$ recognized by $\mathcal{A}$ consists of those words $u \in \Sigma^*$ for which there exists a u-labeled path from some initial state to a final state, formally

$$L(\mathcal{A}) = \{u \in \Sigma^* \mid I \xrightarrow{u}_{\mathcal{A}} F\}.$$

When $\mathcal{A}$ is understood, we sometimes omit the subscript in $\xrightarrow{u}_{\mathcal{A}}$ and $u_{\mathcal{A}}$.

We call $\mathcal{A}$ a **deterministic finite automaton** (DFA) if it has at most one initial state, and each relation $\sigma_{\mathcal{A}}$ ($\sigma \in \Sigma$) is a *partial* function $Q \to Q$. A deterministic automaton is called **complete** if it has a unique initial state, and each of its input symbols induces a total function. The automaton $\mathcal{A}$ is called a **reset automaton** if it has at most one initial state and each input symbol $\sigma \in \Sigma$ induces either the identity function or a partial constant function $Q \to Q$. Note that each reset automaton is deterministic.

A state q of $\mathcal{A}$ is called **accessible** (respectively, **coaccessible**) if there exists some input word $u \in \Sigma^*$ with $I \xrightarrow{u}_{\mathcal{A}} \{q\}$ (respectively, $\{q\} \xrightarrow{u}_{\mathcal{A}} F$). Note that each initial state is accessible and each final state is coaccessible. A **biaccessible** state is one which is both accessible and coaccessible. Two states $q, q' \in Q$ are called **equivalent**, denoted $q \approx_{\mathcal{A}} q'$, if

$$\{q\} \xrightarrow{u}_{\mathcal{A}} F \iff \{q'\} \xrightarrow{u}_{\mathcal{A}} F,$$

for all input words $u \in \Sigma^*$. Suppose that $\mathcal{A}$ is a DFA. Then $\mathcal{A}$ is called **minimal** if all of its states are biaccessible, and it has no different equivalent states, and $\mathcal{A}$ is called **aperiodic** if there exists an integer $k \geq 0$ such that $(u^k)_{\mathcal{A}} = (u^{k+1})_{\mathcal{A}}$, for all $u \in \Sigma^*$. Observe that if $\mathcal{A}$ is a reset automaton then $(u^2)_{\mathcal{A}} = (u^3)_{\mathcal{A}}$, and if $\mathcal{A}$ is a complete reset automaton then $u_{\mathcal{A}} = (u^2)_{\mathcal{A}}$, for all words $u \in \Sigma^*$. Thus, each reset automaton is an aperiodic DFA.

Remark. It is well known (see [3]) that a deterministic automaton $\mathcal{A}$ is aperiodic if and only if it satisfies the implication $q \xrightarrow{u^k}_{\mathcal{A}} q \implies q \xrightarrow{u}_{\mathcal{A}} q$, for all states q, (nonempty) input words u and integers $k \geq 2$.

1.2 Turing machines

We assume the reader is familiar with Turing machines and the basic concepts of complexity theory, such as space-complexity and logspace-reducibility (see [1], for example). Nevertheless, we briefly review some basic definitions and notations.

A **deterministic Turing machine** (DTM) with a single one-way infinite tape is a system $\mathcal{M} = (Q, \Gamma, \Sigma, \delta, q_0, q_f)$, where

- Q is the finite set of states,
- Γ is the tape alphabet containing the special "blank" symbol $\flat$,
- $\Sigma \subseteq \Gamma$ is the input alphabet, $\flat \notin \Sigma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, 1\}$ is the partial transition function,
- $q_0 \in Q$ is the initial state,
- $q_f \in Q$ is the final state.

We say the machine $\mathcal{M}$ is in the **configuration** (q, i, u) for a state $q \in Q$, integer $i \in \omega$ and infinite word $u \in \Gamma^\omega$ if in state q it scans the ith tape cell and the content of the tape is u. The **language** $L(\mathcal{M}) \subseteq \Sigma^*$ **recognized by** $\mathcal{M}$ consists of those input words $u \in \Sigma^*$ for which $\mathcal{M}$ reaches the final configuration $(q_f, 1, \flat^\omega)$ when it is started from the initial configuration $(q_0, 1, u\flat^\omega)$.

The language class **PSPACE** consists of those languages which are recognized by some Turing machine $\mathcal{M}$ having space-complexity p, for some polynomial function $p : \mathbf{N} \to \mathbf{N}$. Suppose L and L' are languages. In this paper, $L \leq_{log} L'$ stands for "L is logspace-reducible to L'". The language L is called **PSPACE-hard** with respect to logspace-reductions, written **PSPACE** $\leq_{log} L$, if every language in **PSPACE** is logspace-reducible to L. Lastly, L is called **PSPACE-complete** with respect to logspace-reductions if $L \in$ **PSPACE** and **PSPACE** $\leq_{log} L$.

2 Results

We are interested in the computational complexity of the following decision problems:

1. The intersection problem of reset automata (**IPR**):
   INPUT: A sequence $\mathcal{A}_1, \ldots, \mathcal{A}_n$ $(n \geq 2)$ of reset automata with a common input alphabet.
   QUESTION: Does $\bigcap_{i \in [n]} L(\mathcal{A}_i) \neq \emptyset$ hold?
2. Automaton star-freeness (**ASF**):
   INPUT: A nondeterministic finite automaton $\mathcal{A}$.
   QUESTION: Does $\mathcal{A}$ recognize a star-free language?

3. Regular expression star-freeness (**RSF**):
   INPUT: A regular expression E.
   QUESTION: Does E denote a star-free language?

Assuming some efficient encoding of automata and regular expressions with words over a fixed finite alphabet (see [4]), all these problems can be considered as languages.

Our first lemma shows that for each reset automaton $\mathcal{A}$ there exists a "short" regular expression denoting the complement of the language recognized by $\mathcal{A}$. This fact plays a key role in proving that the problem **RSF** is **PSPACE**-hard.

Lemma 1. *Suppose that $\mathcal{A} = (Q, \Sigma, \tau, I, F)$ is a reset automaton. Then there exists a regular expression E of length $O(|Q| \cdot |\Sigma|)$ such that $L(E) = \overline{L(\mathcal{A})}$.*

Proof. We may assume that $\mathcal{A}$ has a unique initial state q_0. Let

$$\begin{aligned} X_q &= \{\sigma \in \Sigma \,|\, Q\sigma_{\mathcal{A}} = \{q\}\} \\ Y_q &= \{\sigma \in \Sigma \,|\, q\sigma_{\mathcal{A}} = \{q\}\} \\ Z_q &= \{\sigma \in \Sigma \,|\, q\sigma_{\mathcal{A}} = \emptyset\} \\ E_q &= \begin{cases} \Sigma^* X_q Y_q^* & \text{if } q \neq q_0 \\ \Sigma^* X_q Y_q^* \cup Y_q^* & \text{if } q = q_0, \end{cases} \end{aligned}$$

for all $q \in Q$. Lastly, let

$$E = \left(\bigcup_{q \in Q \setminus F} E_q \right) \cup \left(\bigcup_{q \in Q} E_q Z_q \Sigma^* \right).$$

One can show that $u \in L(E_q) \implies q_0 u \subseteq \{q\}$ and $q_0 u = \{q\} \implies u \in L(E_q)$, for all $q \in Q$, $u \in \Sigma^*$. Then $L(E) = \overline{L(\mathcal{A})}$ follows since the definition of E expresses the fact that an input word $u \in \Sigma^*$ is rejected by the automaton $\mathcal{A}$ either if $q_0 u = \{q\}$ for some non-final state q, or $q_0 u = \emptyset$. □

Our main result is the following.

Theorem 2. *The problems* **IPR**, **ASF** *and* **RSF** *are* **PSPACE***-complete with respect to logspace reductions.*

Proof. We show

$$\mathbf{PSPACE} \leq_{log} \overline{\mathbf{IPR}} \leq_{log} \mathbf{RSF} \leq_{log} \mathbf{ASF} \in \mathbf{PSPACE},$$

where $\overline{\mathbf{IPR}}$ is the complement of the problem **IPR**. It is easy to see that $\mathbf{RSF} \leq_{log} \mathbf{ASF}$: given a regular expression E, a logspace-bounded Turing machine can construct a *nondeterministic* automaton $\mathcal{A}$ such that $L(E) = L(\mathcal{A})$.

Proof of $\mathbf{PSPACE} \leq_{log} \overline{\mathbf{IPR}}$: Suppose that $L \subseteq \Sigma^*$ is a language in **PSPACE**. Then also $\overline{L} \in \mathbf{PSPACE}$, and thus there exist a polynomial function $p : \mathbf{N} \to \mathbf{N}$

and a deterministic Turing machine $\mathcal{M} = (Q, \Gamma, \Sigma, \delta, q_0, q_f)$ of space-complexity p with $L(\mathcal{M}) = \overline{L}$. Suppose that $u \in \Sigma^n$ is an input word, for some $n \geq 0$. We construct a sequence $\mathcal{S}, \mathcal{P}, \mathcal{A}_1, \ldots, \mathcal{A}_m$ of reset automata, where $m = \max\{p(n), 1\}$, such that

$$u \in L(\mathcal{M}) \iff L(\mathcal{S}) \cap L(\mathcal{P}) \cap \bigcap_{i \in [m]} L(A_i) \neq \emptyset. \tag{1}$$

Let

$$\mathcal{S} = (Q, A, \tau_{\mathcal{S}}, \{q_0\}, \{q_f\})$$
$$\mathcal{P} = ([m], A, \tau_{\mathcal{P}}, \{1\}, \{1\}),$$

and for each $i \in [m]$,

$$\mathcal{A}_i = (\Gamma, A, \tau_i, \{(ub^\omega)_i\}, \{b\}),$$

where

$$A = \{\langle q, k, \gamma\rangle \,|\, q \in Q,\ k \in [m],\ \gamma \in \Gamma\}$$

and the transition functions $\tau_{\mathcal{S}}, \tau_{\mathcal{P}}, \tau_1, \ldots, \tau_m$ are defined as follows.

Suppose that $a = \langle q, k, \gamma\rangle$ is an element of A. If $\delta(q, \gamma)$ is undefined then

$$\tau_{\mathcal{S}}(a) = \tau_{\mathcal{P}}(a) = \tau_1(a) = \cdots = \tau_m(a) = \emptyset,$$

and if $\delta(q, \gamma)$ is defined, say $\delta(q, \gamma) = (r, \gamma', t)$, then

$$\tau_{\mathcal{S}}(a) = \{(q, r)\}$$
$$\tau_{\mathcal{P}}(a) = \begin{cases} \{(k, k+t)\} & \text{if } k + t \in [m], \\ \emptyset & \text{if } k + t \notin [m], \end{cases}$$
$$\tau_i(a) = \begin{cases} \{(\gamma, \gamma')\} & \text{if } k = i \\ \{(\sigma, \sigma) \,|\, \sigma \in \Gamma\} & \text{if } k \neq i. \end{cases}$$

The intuition is that the automata $\mathcal{S}, \mathcal{P}, \mathcal{A}_1, \ldots, \mathcal{A}_m$ together "simulate" the computation of $\mathcal{M}$ on the input word u, such that $\mathcal{S}$ knows the current state of $\mathcal{M}$, $\mathcal{P}$ knows the position of the read-write head, and each $\mathcal{A}_i$ ($i \in [m]$) knows the content of the ith tape-cell. An input symbol $\langle q, k, \gamma\rangle \in A$ corresponds to the statement "the current state of $\mathcal{M}$ is q, the position of the read-write head is k, and the content of the kth tape-cell is γ".

It is easy to see that each one of $\mathcal{S}, \mathcal{P}, \mathcal{A}_1, \ldots, \mathcal{A}_m$ is a reset automaton. (In fact they are even more restricted: for all input symbols $a \in A$, the relation induced by a in each one of the automata $\mathcal{S}, \mathcal{P}, \mathcal{A}_1, \ldots, \mathcal{A}_m$ is either empty, or a singleton, or the identity function.)

Proof of $\overline{\mathbf{IPR}} \leq_{log} \mathbf{RSF}$: Suppose that $\mathcal{B}_1, \ldots, \mathcal{B}_n$ ($n \geq 2$) are reset automata, say $\mathcal{B}_i = (Q_i, \Sigma, \tau_i, I_i, F_i)$. First we construct a DFA $\mathcal{C}$ such that

$$\bigcap_{i \in [n]} L(\mathcal{B}_i) = \emptyset \iff L(\mathcal{C}) \text{ is star-free.} \tag{2}$$

(The very same construction was used by Cho and Huynh in [2].)

We may assume that each automaton $\mathcal{B}_i$ has a unique initial state s_i, so that $L(B_i) \neq \emptyset$, for all $i \in [n]$. Let p be the least prime number with $p \geq n$. It is well known (see [5]) that $p < 2n$. For integers $i \in \{n+1, n+2, \ldots, p\}$ let $\mathcal{B}_i = (Q_i, \Sigma, \tau_i, \{s_i\}, F_i)$ be the minimal DFA recognizing the language Σ^*. Then let $\mathcal{C}$ be the DFA depicted on Figure 1. See [2] for a proof of (2). Note that

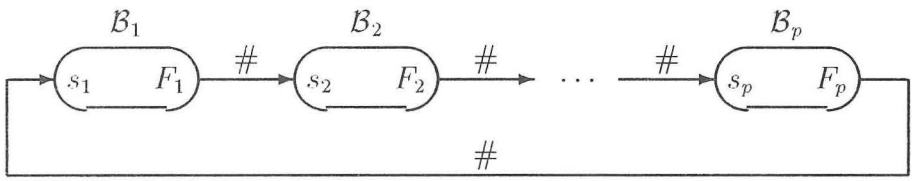

Fig. 1. The automaton $\mathcal{C}$

$\mathcal{C}$ is a DFA with

$$L(\mathcal{C}) = \left(L(\mathcal{B}_1)\#L(\mathcal{B}_2)\#\cdots L(\mathcal{B}_n)\#(\Sigma^*\#)^{p-n}\right)^* .$$

It follows that a word $v = v^{(0)}\#v^{(1)}\#\cdots v^{(k-1)}\#v^{(k)}$ ($k \geq 0$, $v^{(0)}, \ldots, v^{(k)} \in \Sigma^*$) belongs to $L(\mathcal{C})$ if and only if $v^{(k)} = \epsilon$, k is a multiple of p, and $v^{(i)} \in L(\mathcal{B}_{(i \bmod p)+1})$, for all $i < k$ with $i \bmod p < n$. The languages denoted by the regular expressions

$$F_1 = (\Sigma \cup \#)^*\Sigma$$

$$F_2 = ((\Sigma^*\#)^p)^* \left(\bigcup_{i \in [p-1]} (\Sigma^*\#)^i \right) \Sigma^*$$

$$F_3 = ((\Sigma^*\#)^p)^* \left(\bigcup_{i \in [n]} (\Sigma^*\#)^{i-1} E_i \# \right) (\Sigma \cup \#)^*$$

consist of those words $v = v^{(0)}\#v^{(1)}\#\cdots v^{(k-1)}\#v^{(k)}$ for which

1. $v^{(k)} \neq \epsilon$,
2. k is not a multiple of p,
3. $v^{(i)} \notin L(B_{(i \bmod p)+1})$ for some $i < k$ with $i \bmod p < n$,

respectively. It follows that the regular expression $E := F_1 \cup F_2 \cup F_3$ denotes the complement of the language $L(\mathcal{C})$. (Note that E has star-height 2.) Then we see

$$L(E) \text{ is star-free} \iff L(\mathcal{C}) \text{ is star-free} \iff \bigcap_{i \in [n]} L(\mathcal{B}_i) = \emptyset.$$

Proof of **ASF** $\in$ **PSPACE**: Suppose that $\mathcal{A} = (Q, \Sigma, \tau, I, F)$ is an NFA. By Schützenberger's theorem, $L(\mathcal{A})$ is star-free if and only if the minimal DFA recognizing $L(\mathcal{A})$ is aperiodic. Recall that the power automaton of $\mathcal{A}$ is the DFA $\mathrm{P}(\mathcal{A}) = (\mathrm{P}(Q), \Sigma, \tau', \{I\}, F')$, where

$$F' = \{S \in \mathrm{P}(Q) \,|\, S \cap F \neq \emptyset\},$$
$$\tau'(\sigma) = \{(S, S\sigma_{\mathcal{A}}) \,|\, S \in \mathrm{P}(Q)\},$$

for all $\sigma \in \Sigma$. The minimal DFA recognizing $L(\mathcal{A})$ is obtained from $\mathrm{P}(\mathcal{A})$ by deleting those states which are not biaccessible, and then identifying the equivalent states. It follows that $L(\mathcal{A})$ is star-free if and only if there exists some input word $u \in \Sigma^*$, *accessible* state S of $\mathrm{P}(\mathcal{A})$ and integer $k \geq 2$ such that $S \approx_{\mathrm{P}(\mathcal{A})} S(u_{\mathcal{A}})^k$ and $S \not\approx_{\mathrm{P}(\mathcal{A})} Su_{\mathcal{A}}$.

It is easy to give a nondeterministic Turing machine $\mathcal{M}_0$ of linear space-complexity for deciding if $S \not\approx_{\mathrm{P}(\mathcal{A})} S'$ holds for two states S, S' of $\mathrm{P}(\mathcal{A})$: if exactly one of the two sets $S \cap F$ and $S' \cap F$ is empty then $\mathcal{M}_0$ halts, otherwise it guesses an input symbol $\sigma \in \Sigma$, and repeats the previous test for the sets $S := S\sigma_{\mathcal{A}}$ and $S' := S'\sigma_{\mathcal{A}}$. By Savitch's theorem we obtain a deterministic machine $\mathcal{M}_1$ of quadratic space-complexity which decides if two states of $\mathrm{P}(\mathcal{A})$ are equivalent. Using $\mathcal{M}_1$ we can build a nondeterministic Turing machine $\mathcal{M}_2$ of quadratic space-complexity which decides if the language recognized by a nondeterministic automaton $\mathcal{A} = (Q, \Sigma, \tau, I, F)$ is not star-free. $\mathcal{M}_2$ works as follows: first it successively guesses the letters of an input word $u \in \Sigma^*$ and calculates the relation $u_{\mathcal{A}} \subseteq Q \times Q$. (This is done by storing *only* the relation $u_{\mathcal{A}}$ on the tape, which requires quadratic space. If the next letter guessed is σ, the next relation $(u\sigma)_{\mathcal{A}}$ is calculated by taking the composition of the currently stored relation $u_{\mathcal{A}}$ with $\sigma_{\mathcal{A}}$. After each step $\mathcal{M}_2$ may decide to stop or continue guessing the next letter.) Then it guesses an accessible state $S \subseteq Q$ of the power automaton $\mathrm{P}(\mathcal{A})$. (Similarly as before, this is done by successively guessing the letters of a word v with $S = Iv_{\mathcal{A}}$.) In the end the deterministic machine $\mathcal{M}_1$ is used to see if $S \not\approx_{\mathrm{P}(\mathcal{A})} Su_{\mathcal{A}}$ and $S \approx_{\mathrm{P}(\mathcal{A})} S(u_{\mathcal{A}})^k$ hold for some $k \geq 2$.

Then again by Savitch's theorem, we obtain a deterministic Turing machine $\mathcal{M}$ of space-complexity $O(n^4)$ which decides if a nondeterministic automaton accepts a star-free language. □

3 Conclusion and open problems

We have proved that the intersection problem of finite reset automata is complete in **PSPACE**, which is a strengthening of Kozen's original result [6] involving arbitrary deterministic automata. (It is also possible to show that restricting the intersection problem to minimal reset automata does not decrease its space-complexity.) Using this result we showed that the problem of deciding whether a regular expression of star-height 2 denotes a star-free language is **PSPACE**-complete. Similarly as in [2], one can show that the same problem remains **PSPACE**-complete if we restrict it to regular expressions of star-height 2 over the two-element alphabet $\{0, 1\}$.

These results suggest that the following questions may be interesting.

1. What is the complexity of deciding whether $\bigcap_{i\in[n]} L(\mathcal{A}_i) \neq \emptyset$ holds for complete reset automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$?
2. What is the complexity of deciding whether a regular expression of star-height 1 denotes a star-free language?

We conjecture that the answer for the first question is "**NP**-complete". The second question seems to be harder. However, it is our conjecture that restricting the problem **RSF** to regular expressions of star-height 1 substantially decreases its space-complexity. This would prove that, in some sense, our results are tight.

4 Acknowledgement

I would like to thank Zoltán Ésik and Stephen L. Bloom for encouragement and for many useful comments and suggestions.

References

1. D. P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity.* Prentice Hall, 1994.
2. Sang Cho and Dung T. Huynh. Finite-automaton aperiodicity is PSPACE-complete. *Theoretical Computer Science*, 88:99–116, 1991.
3. Samuel Eilenberg. *Automata, Languages and Machines.* Academic Press, New York and London, 1974.
4. Michael R. Garey and David S. Johnson. *Computers and intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.
5. G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, London, 3rd edition, 1954.
6. Dexter Kozen. Lower bounds for natural proof systems. In *Proc. 18th Ann. Symp. on Foundations of Computer Science*, pages 254–266, Long Beach, CA, 1977. IEEE Computer Society.
7. M. P. Schützenberger. On Finite Monoids Having Only Trivial Subgroups. *Information and Control*, 8:190–194, 1965.
8. Jacques Stern. Complexity of Some Problems from the Theory of Automata. *Information and Control*, 66:163–176, 1985.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 35–42

Implementing Discrete Time in Promela and Spin

Dragan Bošnački

Institute of Informatics, Faculty of Natural Sciences and Mathematics
University "St. Cyril and Methodius"
P.O. Box 162, 91 000 Skopje, Macedonia
e-mail: dragan@pmf.ukim.edu.mk

Abstract. An approach is presented to implementation of discrete time in Promela, a high level modelling language for specification of concurrent systems. Discrete time features are introduced on user level, using only the existing language constructs and without any change in Spin, its associated software package for automated analysis and validation. It is shown how the new time framework can be used for simulation and validation of systems which correct functioning depends crucially on timing.

1 Introduction

Promela is a high level modelling language for specification of concurrent systems. The models (programs) written in Promela are used as an input for *Spin* software package for their automated simulation and validation.

Originally, Promela and Spin have been developed for analysis and validation of communication protocols. The language syntax is derived form C, but also uses the denotations for communications from Hoare's CSP and control flow statements based on Dijkstra's guarded commands.

The full presentation of the language and the validator is beyond the scope of this paper and we suggest [5] as a reference to the interested reader.

In Promela, the system components are modeled as *processes* that can communicate via *channels* either by buffered message exchanges or rendez-vous operations, and also trough shared memory represented as global *variables*. The execution of actions is considered asynchronous and interleaved, which means that in every step only one *enabled* action is performed and without any additional assumptions of the relative speed of the process execution.

Given as an input a Promela model Spin can do random or interactive simulations or to generate a C program that performs a validation of the system by scanning the state space.

Xspin is a graphical interface for Spin. It provides an integrated windowing environment for writing Promela models and carrying out virtually all Spin functions in a user friendly manner. The outputs are displayed in various ways, among which are Message Sequence Charts.

The time ordering of the actions in Promela is implicit and is given by the sequential composition (concatenation) of statements, like in the ordinary procedural programming languages. The time relations are only *qualitative*, meaning that we only know that an action will eventually be executed after another, but we do not know the exact time interval that will elapse between the two events.

This can be a significant shortcoming when the systems which correct functioning closely depend on the timing parameters are to be simulated or verified (e.g. communication protocols which have to deal with unreliable transport media, where the duration of timeout intervals could be of utmost importance).

To overcome this problem we extend Promela with the concept of discrete time. In the discrete time concept used in this paper time is divided in slices indexed by natural numbers. The actions are then framed into those slices, obtaining in that way a good *quantitative* estimation for the intervals between the events belonging to different slices. Within a slice however, we can only have the qualitative relation between the events, as in the time free case.

To the best of author's knowledge, there are three other attempts ([9, 7, 2]) to introduce timing in Promela and Spin. The concept of [9] which uses real (dense) time has certain problems with the urgent actions which impose limits to its applications. The ongoing research of [7] is also based on real time, but it is still in its early phase. The common carachteristic of those two approaches is that, unlike our approach, they implement the time features on the level of Spin, by changing its source code. The approach of [2] that uses discrete time and is the most similar to the one presented here, suffered from the very big memory requirements. It implements discrete time on user level too.

2 Implementation of Discrete Time in Promela

For simulation purposes discrete time can be implemented in a straightforward way by introducing one global integer variable and a special process as in the following segment

```
int TMaster;

proctype MasterClock()
{
  do :: timeout->TMaster=TMaster+1; od
}
```

The variable TMaster is acting as a master clock on which all the processes are then synchronized, by forcing them to wait for the next clock ticks to continue their execution.

For instance, we can implement a delay of one tick with the following sequence:

```
T=TMaster; TMaster==T+1 -> other_statements
```

where `T` is a local integer variable. A process containing this sequence will hang on the boolean condition until `TMaster` is incremented by one. In that way the execution of `other_statements` will be postponed for the next time slice.

The value of `TMaster` can be increased only by the process `MasterClock`, which consists of an endless `do` loop. An instantiation of `MasterClock`, started form `init` process, is run concurrently with the other processes of the system.

The key idea of the concept is the usage of `timeout` - a predefined Promela statement that becomes true when no other statement within the system is executable. Its usage is allowed only in `Clocks` process.

By guarding the incrementation of the clock with `timeout` we ensure that no process will proceed with an action from the next time slice until all the other processes have executed all actions from the current time slice. (Within the same time slice the actions can be interleaved in an arbitrary way, of course.) Putting `timeout` at the beginning of each iteration causes `MasterClock` to have the least priority of all processes. Thus, it has to wait for all of them to finish the execution of their actions for the current slice, and then it can increment `TMaster`. This will trigger the hanging processes and in fact cause the system to pass into a new time slice.

It is worth noting that `timeout` was introduced in Promela to make up the lack of quantitative treatment of timing and it would have been useless in a time setting. Instead, in our concept it is given one of the central roles. By using `timeout` we continue to use the built-in Spin process scheduling and synchronization. In that way we avoid the introduction of additional "housekeeping" processes and variables. For instance, elements like process scheduler, semaphore variables or some other synchronization mechanisms, might be needed. It is very likely that we will lose in the clarity of the specifications and growth of the state space if all of them are implemented on user level.

When using Xspin, one can add an appropriate `printf` statement in `Master Clock` so that the changes of `TMaster` can be displayed on the message sequence charts output.

Although it can be improved in several ways, the described straightforward approach would still not be adequate for validation of the discrete time models. Even if we solve the problem of clock overflows, for instance, the virtually unbounded increasing of the master-clock will inevitably lead to a state space explosion.

The remedy is to use multiple clocks instead of just one master-clock. With a careful managing of their operation we will be able to eliminate a lot of unnecessary transitions and shrink the state space significantly. This new approach is given by the following suite of Promela macro definitions:

```
#define clock short
#define OFF (-1)
#define TIMEOUT (0)
#define set(x,y) (x=y)
```

```
#define shut(x) set(x,OFF)
#define on(x) (x!=OFF)
#define tick(x) if :: on(x)->x=x-1; :: else; fi
#define tmout(x) (x==TIMEOUT) /*timeout*/
#define delay(x,y) set(x,y); tmout(x);
```

to which we add the process `Clocks`, which is a modification of `MasterClock` adapted to the new concepts. If we assume only two clocks in the model, `sc` and `rc`, it could look like

```
proctype Clocks()
{
    do
    :: timeout -> atomic{tick(sc); tick(rc);}
                  if
                  :: !on(sc) && !on(rc) -> break;
                  :: else;
                  fi;
    od;
}
```

The first macro defines `clock` as a synonym for the `short` type. We could have as well used any other integer type - the choice of `short` is motivated merely by practical reasons. The special constants `OFF` and `TIMEOUT` determine two special states of the clock - the inactive and timeout state, respectively, that will be described in a while. The next five definitions that follow reflect the work of clocks as countdown timers. All clocks are shut off in the beginning of `init` process by initializing them to `OFF` using `shut`. `shut` is only a variant of `set` trough which a clock can be set to an arbitrary value. If the clock has been shut off, setting it to a positive value will switch it `on`. Although all the clocks can be set to by any process, they can be decreased only by execution of a `tick` in the `Clocks`. Moreover, a clock value can be decreased only if it is not `OFF`, i.e. only if the clock is active. The `tmout` macro indicates a timeout, a moment of timer expiration when it reaches the timeout value. `delay` can be regarded as a "higher level" macro that uses `set` and `tmout` and is used to postpone the execution of a statement for a fixed number of time slices.

There are some new features in `Clocks` that were not present in `MasterClock`. `Clocks` is usually decreasing multiple clocks, instead of one and the clocks are decremented only if they are not shut off. In case all clocks are switched off then the loop is broken and this leads to a deadlock of the system. This feature was necessary in the new time setting, because having `timeout` at the beginning of the endless loop we loose the possibility to detect a deadlock. Now there is a process that is always active - namely the `Clocks` itself! And this is of course artificial, because we have introduced `Clocks` as a mere modelling convenience and it is not a component of the original system. Allowing for loop to be broken, we restore the important capability of deadlock detection.

The usage of `atomic` sequence is to avoid the standard *test and set problem*, i.e. the interference of other processes which otherwise could have been activated by the incrementation of some of the clocks. Such an unwanted awakening of processes between two ticks in the `Clocks` process could lead to inconsistencies in the synchronization.

3 Validation in Promela with Discrete Time

In this section we show how discrete time can be used for specification and validation of systems in Promela and Spin on the example of Parallel Acknowledgment with Retransmission (PAR) protocol [8].

The choice of PAR was motivated by the fact that it is relatively simple protocol for which almost complete model can be given in the paper, but it is yet complex enough that its correct functioning depends on the duration of time intervals in a nontrivial way. Another advantage of PAR is that it occurs often in the literature on specification and verification (e.g. [10, 6]), so it is well-known and also the results obtained with discrete time Promela can be checked against the existing ones.

Informal Description of the Protocol. PAR is one-way (simplex) data-link level protocol intended to be used over unreliable transmission channels which may corrupt or lose data. There are four components in our implementation a *sender*, a *receiver*, *data transmission channel K* and *acknowledgment transmission channel L*.

The sender receives data from the upper level and sends them labelled with a sequence number over the unreliable channel K. The sequence alternates between 0 and 1 - sometimes PAR is classified as a variant of Alternating Bit Protocol. After that it waits for an acknowledgment, which should be received via (ureliable) channel L, from the receiver before a new datum is transmitted. If an acknowledgment does not occur after some period of time, the sender times out and resends the old data. The receiver receives data from the channel K and if the data are undamaged and labelled with the expected sequence number it delivers them to the upper level and sends an acknowledgment trough the channel L to the receiver.

Of crucial importance here is the duration of the time-out period which should be longer than the sum of the delays trough the channels and message processing time by the receiver, otherwise the premature timeout can cause the loss of a frame.

A Model of PAR Protocol in Promela with Discrete Time. Because of the space limitations we give only the incomplete listing of the model of PAR (without Receiver process), which can hopefuly give a flavour of the way the new discrete time framework can be used.

```
/*discrete time macros*/
```

```
#define clock short
/*...the same as in the text above*/
#define delay(x,y) set(x,y); tmout(x);

/*PAR time parameters*/
#define dK 3 /*delay along the channel K*/
#define dL 3 /*delay along the channel L*/
#define dR 1 /*delay along the channel R*/
#define To 9 /*timeout interval*/
#define MAX 8 /*max number of different message contents*/

/*channels*/
chan K = [1] of {byte, bit}
chan L = [1] of {bit}

/*clocking*/
clock sc, rc;

proctype Clocks()
{
/*...the sprocess is given in the text above*/
}

proctype Sender(chan in, out)
{
byte mt; /* message data */
bit  sn=0; /* sequence number*/

  R_h:  /*unbounded start delay - sending of a new message
        can start in any time slice*/
        do
        :: delay(sc,1);
        :: mt = (mt+1)%MAX; break;
        od;
  S_f:  delay(sc,dK); out!mt,sn; /*sand and delay in channel K*/
        set(sc,To-dK);
        /*modelling peculiarity - the delay along the
        channel K should be subtracted from the timeout period*/
  W_s:  do
        :: in?_ ->
          if
          :: atomic{skip; delay(sc, 1); sn=1-sn; goto R_h;};
                                              /*ack is OK*/
          :: atomic{printf("MSC: ACKerr\n");  goto S_f};
          fi;
        :: tmout(sc) -> goto S_f;  /*timeout*/
        od;
}

proctype Receiver(chan in, out) { /*...not given ...*/}
```

```
init { atomic{shut(sc); shut(rc);
       run Clocks();
       run Sender(L, K);
       run Receiver(K, L); } }
```

Given as an input to the Spin validator (version 2.9.4) running on SPARC station 5 with 64 MBytes of memory, the PAR model resulted in a resource consumption shown in the table below.

time parameters				*resource usage*		
dK	dL	dR	To	*transitions*	*memory* [MB]	*CPU time* [sec]
3	3	1	9	764	2.4	0.8
30	30	10	90	2807	2.5	1.0
300	300	100	900	23237	4.4	3.4
3000	3000	1000	9000	227537	21.7	25.9

Spin was verifying the safety properties (like absence of deadlock, and unspecified receptions), and also the assertion that always the expected message was received, by an exhaustive search of the state space.

It is interesting that the resource usage for small timing intervals, like the ones in the first row in the table, is virtually the same as for the time free model. With the growth of the time parameters the state space increases linearly. Unfortunately, it is only a characteristics of simpler protocols like PAR. For more complex protocols one should expect an exponential growth.

When given a PAR program with incorrect time parameters - when $\texttt{To} < \texttt{dK} + \texttt{dR} + \texttt{dL}$ - Spin was able to find the scenario that leads to message loss. This is another useful feature, because, even for the systems for which the complete validation of the model is impossible, it can help in discovering sequences of events that lead to an incorrect system behaviour (if they occur in the early phase of the state space search, of course).

4 Conclusions and Future Work

We presented an implementation of discrete time in Promela and Spin, using integer variables as stop-watches (clocks) to stamp the time slices. The core idea was to use Promela `timeout` predefined statement as a mechanism for process (i.e. clock) synchronization.

The new time language constructs were implemented as Promela macro definitions and their usage was demonstrated on the specification and validation of PAR protocol.

The main future task will certainly be to test the approach and accumulate experience by doing new verifications of systems known in the literature or some newly developed industry cases which depend on timing parameters.

So far we have applied the Discrete Time Promela on several concurrent systems with various complexity among which we would like to emphasize the Bounded Retransmission Protocol [3, 4], a "real-world" industry protocol used by Phillips. Following the lines from [3] we have been able to obtain all the verification results presented there. Moreover, our approach has an advantage that Spin treats data more naturally than Uppaal, a tool for symbolic model-checking of real-time systems, used in [3]. We live the full presentation of the results for a forthcoming paper.

It would be very interesting to incorporate the ideas about data and time abstraction of [2] in the existing models in order to obtain verifications that are independent of the concrete parameter values, and to see how general is their applicability. Another open question is to abstract the number of processes, so that the verifications will not depend on this parameter too.

One of the main goals for further research will be to establish the relation of discrete time extensions of Promela and Spin with "more formal" formal methods, like, for example, some version of timed Büchi automata with discrete time or algebra for communicationg processes with descrete timing [1].

Acknowledgments: The author would like to thank for the help during the work on this paper: Dennis Dams, Stavros Tripakis, Rob Gerth, Jos Baeten, Michel Reniers, Sjouke Mauw and Bart Knaack.

References

1. Baeten, J. C. M., Bergstra, J. A., *Discrete Time Process Algebra*, Formal Aspects of Computing 8 (2), 1991, pp. 142-148
2. Dams, D., Gerth, R., *Bounded Retransmission Protocol Revisited*, to appear in Second International Workshop on Verification of Infinite Systems, Infinity, 1997
3. D'Argenio, P. R., Katoen, J.-P., Ruys, T., Tretmans, J., *The Bounded Retransmission Protocol Must Be on Time!*, TACAS'97, 1997
4. Groote, J. F., van de Pol, J., *A Bounded Retransmission Protocol for Large Data Packets*, in Wirsing, M., Nivat, M., ed., Algebraic Methodology and Software Technology, LCNS 1101, pp. 536-550, Springer-Verlag, 1996
5. Holzmann, G. J., *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`
6. Kluesner, A. S., *Models and Axioms for a Fragment of Real Time Process Algebra*, Ph. D. Thesis, Eindhoven University of Technology, 1993
7. Knaack, B., *Real-time extension of SPIN - Ongoing Research*, Abstracts from SPIN'97, Third SPIN Workshop, Twente University, Enschede, The Netherlands, Also: `http://netlib.bell-labs.com/netlib/spin/ws97/papers.html`
8. Tanenbaum, A., *Computer Networks*, Prentice Hall, 1989
9. Tripakis, S., Courcoubetis, C., *Extending Promela and Spin for Real Time*, TACAS '96, LCNS 1055, Springer Verlaag, 1996
10. Vaandrager, F., *Two Simple Communication Protocols*, in Baeten, J.C.M., ed., Applications of Process Algebra, pp.23-44,Cambridge University Press, 1990

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 43–50

Transformations of evolving algebras

Stephan Diehl

FB 14 - Informatik, Universität des Saarlandes
Postfach 15 11 50, 66041 Saarbrücken, Germany
e-mail: diehl@cs.uni-sb.de

Abstract. We give a precise definition of evolving algebras as nondeterministic, mathematical machines. All proofs in the paper are based on this definition. First we define constant propagation. We extend evolving algebras by macros and define folding and unfolding transformations. Next we introduce a simple transformation to flatten transition rules. Finally a pass separation transformation for evolving algebras is presented. It can be used to derive a compiler and abstract machine from an interpreter. All transformations are proven correct. Finally a comparison to other work is given.

1 Introduction

Evolving algebras (EvAs) have been proposed by Gurevich in [Gur91] and used by Gurevich and others to give the operational semantics of languages like C, Modula-2,Prolog and Occam. Börger and Rosenzweig's proof of the correctness of the Warren Abstract Machine is based on a slight variation of evolving algebras ([BR92]). An evolving algebra may be tailored to the abstraction level necessary for the intended application of the semantics, e.g. we might have a hierarchy of evolving algebras, each being more concrete with respect to certain aspects of the semantics. In this paper we only discuss syntactic-sugar free evolving algebras. As a result reading descriptions of an EvA using this notation is harder than reading descriptions, which make extensive use of syntactic-sugar. The advantage of considering the syntactic-sugar free EvAs is clearly, that we have to deal with less constructs when we define EvAs and a variety of transformations, as well as, when we prove operational equivalence and other properties.

Syntactic-sugar free EvAs For our purposes here, we need a precise definition of what an EvA is, and what a computation of an EvA looks like. An **evolving algebra** Ψ is a quadruple $< \sigma, S, T, \mathcal{I}_0 >$ where [1] σ is a *signature*, i.e. a finite set of function names with associated arity, S is a nonempty set, called the *superuniverse*, T is a finite set of transition rules and $\mathcal{I}_0 : \sigma \rightarrow \bigcup_{n \geq 0}(S^n \rightarrow S)$ is the initial interpretation of functions in σ, i.e. $\mathcal{I}_0$ maps every function name f of arity n to an interpretation function $\mathcal{I}_0(f) : S^n \rightarrow S$.
Transition rules are either **function updates** $\boxed{f(t_1, ..., t_n) := t_0}$, where $f \in \sigma$, $n \geq 0$ is the arity of f and the t_i are terms, or **guarded updates** $\boxed{\textit{if } b \textit{ then } C}$, where b is

[1] We will assume $\{true, false\} \subseteq S$.

a term and C is a set of transition rules. A term t is either of the form $f(t_1, ..., t_n)$, where $f \in \sigma, n \geq 0$ is the arity of f and the t_i are terms, or $t \in S$.
A function update changes the interpretation of a function f for the arguments $t'_1, \ldots, t'_n$ to the value t'_0, where t'_i is the value of the term t_i in the current interpretation. In a guarded update the updates in C are only executed, if the guard b is true in the current interpretation.
We will use the notation $\mathcal{I} \xrightarrow{\Psi} \mathcal{I}'$ to indicate, that $\mathcal{I}'$ is the result of applying the transition rules of Ψ to $\mathcal{I}$. We will call this a *step* of the evolving algebra. Before we can define a step of an EvA, we have to introduce some notation. First we define the value of a term t in an interpretation $\mathcal{I}$ and the evaluated form of a function update:
$eval(f(t_1, ..., t_n), \mathcal{I}) = \quad \mathcal{I}(f)(eval(t_1, \mathcal{I}), \ldots, eval(t_n, \mathcal{I}))$ *for* $n \geq 0$
$eval(f(t_1, ..., t_n) := t_0, \mathcal{I}) = f(eval(t_1, \mathcal{I}), \ldots, eval(t_n, \mathcal{I})) := eval(t_0, \mathcal{I})$ *for* $n \geq 0$
Let T be a set of transition rules and $\mathcal{I}$ be an interpretation, then those function updates occurring in T can be executed, which either depend on guards evaluating to true in the interpretation or on no guard at all. We define $updates(T, \mathcal{I}) = \{eval(u, \mathcal{I}) : u \in T \wedge u$ is a function update$\} \cup updates(U, \mathcal{I})$ where U is the union of all C, such that $\boxed{\textit{if } b \textit{ then } C} \in T$ and $eval(b, \mathcal{I}) = true$.
There can be several conflicting function updates in $updates(T, \mathcal{I})$, i.e. evaluated function updates, which change the interpretation of a function for the same arguments to different values. Let M be a set of evaluated function updates, then $\overline{M}$ denotes the set of all greatest subsets A of M, such that if $\boxed{f(t_1, ..., t_n) := t_0}$ in A then there is no update $\boxed{f(t_1, ..., t_n) := t'_0}$ in A where $t_0 \neq t'_0$. The relation $\xrightarrow{\Psi}$ is defined as follows: $\mathcal{I} \xrightarrow{\Psi} \mathcal{I}' \Leftrightarrow \exists U \in \overline{updates(T, \mathcal{I})} \quad \forall \tilde{a} \in S^*, s \in S, f \in \sigma :$

$$\mathcal{I}'(f)(\tilde{a}) = \begin{cases} s & \textit{if } \boxed{f(\tilde{a}) := s} \in U \\ i & \textit{if } f \textit{ is an external function } (\textit{for some } i \in S) \\ \mathcal{I}(f)(\tilde{a}) & \textit{otherwise} \end{cases}$$

Note, that if $\overline{updates(T, \mathcal{I})}$ is not a singleton, then from every set of conflicting updates only one member is chosen nondeterministically.
A terminating *computation* of an evolving algebra Ψ is a sequence $< \mathcal{I}_0, \mathcal{I}_1, ..., \mathcal{I}_k >$, such that $\mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_1 \xrightarrow{\Psi} ... \xrightarrow{\Psi} \mathcal{I}_k$ and $updates(T, \mathcal{I}_k) = \emptyset$. Sometimes we will use the notation $\mathcal{I}_0 \xrightarrow{\Psi}\!\!\!\!\!\rightarrow \mathcal{I}_k$ to refer to a computation. Furthermore the set $reach(\mathcal{I}_0)$ is defined as $\{\mathcal{I}_m : \exists \mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_1 \xrightarrow{\Psi} \ldots \xrightarrow{\Psi} \mathcal{I}_m\}$.

Proof Method Let Ψ and Ψ' be EvAs and $\mathcal{F}$ be a partial mapping of interpretations in Ψ' to those in Ψ. Then Ψ' is **correct** wrt. Ψ iff $\mathcal{I}_0 = \mathcal{F}(\mathcal{I}'_0)$ and for every terminating computation $\mathcal{I}'_0 \xrightarrow{\Psi'}\!\!\!\!\!\rightarrow \mathcal{I}'_k$ there is a terminating computation $\mathcal{I}_0 \xrightarrow{\Psi}\!\!\!\!\!\rightarrow \mathcal{F}(\mathcal{I}'_k)$. Furthermore Ψ is **complete** wrt. Ψ, iff for every terminating computation $\mathcal{I}_0 \xrightarrow{\Psi}\!\!\!\!\!\rightarrow \mathcal{I}_k$ there is a terminating computation $\mathcal{I}'_0 \xrightarrow{\Psi'}\!\!\!\!\!\rightarrow \mathcal{I}'_m$ such that $I_k = \mathcal{F}(\mathcal{I}'_m)$. If Ψ' is both correct and complete wrt. Ψ, then Ψ' and Ψ are **operational equivalent**. The proof method is discussed in more detail in [BR92].

2 Transformations

Constant Propagation In evolving algebras functions are classified as internal or external. External functions mimic input to the evolving algebra, i.e. how their interpretation changes at each step of the evolving algebra can not be foreseen.

An internal function f is called static, if there is no function update to f in the transition rules. We will extend this classification by allowing external functions to be static or dynamic. We will call an external function static, if we know its value on all arguments a priori. We actually turn an external function into an internal static one. Now we will show, how a given EvA can be partially evaluated with respect to its static functions. First we define the result of constant propagation $\pi(t)$ of a term t. If $t \equiv f(t_1, ..., t_n)$ and f is static then $\pi(t) = \mathcal{I}(f)(\pi(t_1), ..., \pi(t_n))$ else $\pi(t) = f(\pi(t_1), ..., \pi(t_n))$. A term is defined to be static, if it does not contain any dynamic function, i.e. t is static iff $t \in S$ or $t = f(t_1, ..., t_n)$ where $n \geq 0$ and all t_i and the function f are static.

Let C be a set of transition rules. We construct the set $\pi(C)$ of the transition rules after constant propagation by induction. $\pi(C)$ is also called the residual of C. For all $r \in C$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $\boxed{f(\pi(t_1), ..., \pi(t_n)) := \pi(t_0)} \in \pi(C)$. If $r \equiv \boxed{\textit{if } b \textit{ then } D}$ and $\pi(b) \notin \{true, false\}$ then $\boxed{\textit{if } \pi(b) \textit{ then } \pi(D)} \in \pi(C)$. Finally, If $r \equiv \boxed{\textit{if } b \textit{ then } D}$ and $\pi(b) = true$ then $\pi(D) \subseteq \pi(C)$.

Theorem: Let $\Psi =< \sigma, S, T, \mathcal{I}_0 >$ and let $\pi(\Psi)$ denote the residual $< \sigma, S, \pi(T), \mathcal{I}_0 >$ of Ψ. Then $\pi(\Psi)$ is operationally equivalent to Ψ.

Proof: After constant propagation in the resulting algebra the same updates are done as before, we only changed the amount of work which is necessary to evaluate terms. So the inital interpretation and the terminal interpretations are preserved (correctness). Furthermore for every terminating computation in Ψ there is a terminating computation in $\pi(\Psi)$ (completeness). The operational equivalence follows immediately from the correctness and completeness. □

Macro Definitions Readability of an evolving algebra can be increased, if we define functions in terms of other functions. First we might think of macro definitions as simple combinations of functions like $snd = fst \circ rest$ implying $\mathcal{I}(snd) = \mathcal{I}(fst) \circ \mathcal{I}(rest)$. But this is not powerful enough. So we will consider macro definitions of a different form,e.g. $mult\_twice(x, y) = mult(plus(x, x), plus(y, y))$, which is to imply $\forall x, y \in S : \mathcal{I}(mult\_twice)(x, y) = \mathcal{I}(mult)(\mathcal{I}(plus)(x, x), \mathcal{I}(plus)(y, y))$.

Let $\tilde{\sigma}$ be the set of all static functions in σ, $f \in \tilde{\sigma}$ and t_0 be a first-order term consisting of function names in $\tilde{\sigma}$ and $x_1, \ldots, x_n$ distinct variables, then a macro definition is of the form $f(x_1, \ldots, x_n) = t_0$. A macro definition is **valid**, iff $eval(f(s_1, \ldots, s_n), \mathcal{I}) = eval(t_0[x_1 \mapsto s_1, \ldots, x_n \mapsto s_n], \mathcal{I})$ for all $s_i \in S$ and all $\mathcal{I} \in reach(\mathcal{I}_0)$.

We have several choices to restrict macros: no additional restrictions on the macros (1), allow only non-recursive definitions (2) or none of the macros defined, may occur in the right hand side of a macro definition (3). We will address the implications of these restrictions in the next section.

Unfolding Macros Let Δ be a set of macro definitions. First we define the Δ-unfolding of a term t, which we will write as $t \restriction \Delta$. If $t \equiv f(t_1, ..., t_n)$ and $(f(x_1, ..., x_n) = t_0) \in \Delta$ then $t \restriction \Delta = t_0[x_1 \mapsto t_1 \restriction \Delta, ..., x_n \mapsto t_n \restriction \Delta]$ else $t \restriction \Delta = t$

We will denote $\underbrace{t \restriction \Delta ... \restriction \Delta}_{n \text{ times}}$ by $t \restriction^n \Delta$. The above mentioned restrictions on macro definitions have the following implications with respect to the Δ-unfolding of a term: it is possible, that there is no n such that $t \restriction^n \Delta = t \restriction^{n+1} \Delta$, e.g. $\Delta = \{f(x) = f(x)\}$ there is an n such that $t \restriction^n \Delta = t \restriction^{n+1} \Delta$ or $t \restriction \Delta = t \restriction^2 \Delta$.

Now we define the Δ-unfolding of a set of transition rules T, which we will write as $T \restriction \Delta$. Let $r \in T$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $\boxed{f(t_1 \restriction \Delta, ..., t_n \restriction \Delta) := t_0 \restriction \Delta} \in T \restriction \Delta$. If $r \equiv \boxed{\textit{if } b \textit{ then } D}$ and $b \restriction \Delta \notin \{true, false\}$ then $\boxed{\textit{if } b \restriction \Delta \textit{ then } D \restriction \Delta} \in T \restriction \Delta$. Finally, if $r \equiv \boxed{\textit{if } b \textit{ then } D}$ and $b \restriction \Delta = true$ then $D \restriction \Delta \in T \restriction \Delta$

Theorem: Let $\Psi =< \sigma, S, T, \mathcal{I}_0 >$, let Δ be a set of valid macro definitions and let $\Psi \restriction \Delta$ denote the evolving algebra $< \sigma, S, T \restriction \Delta, \mathcal{I}_0 >$. Then $\Psi \restriction \Delta$ is operationally equivalent to Ψ.

Proof: In the unfolded algebra the same updates are done as before, we only changed the structure of the terms, not their interpretation, i.e. the value they evaluate to. The operational equivalence follows by the same argument used for the proof in the previous section. □

Folding Macros As before let Δ be a set of macro definitions. First we define the Δ-folding of a term t, which we will write as $t \downarrow \Delta$. Furthermore we will use $\sqcap$ to denote unification of first-order terms.If $t \equiv f(t_1, ..., t_n)$ and $t_i^* \in t_i \downarrow \Delta$ then $f(t_1^*, ..., t_n^*) \in t \downarrow \Delta$. Furthermore, if $f(t_1, ..., t_n)$ and t_0 are unifiable, i.e. $f(t_1, ..., t_n) \sqcap t_0$ is defined and $g(x_1, ..., x_m) = t_0 \in \Delta$ then $g(\hat{x}_1, ..., \hat{x}_m) \in t \downarrow \Delta$, where the $\hat{x}_i$ are terms, such that $f(t_1, ..., t_n) = t_0[x_1 \mapsto \hat{x}_1, ..., x_m \mapsto \hat{x}_m]$ Note, that in an implementation we do not need an occurs check here, because we always unify a variable free term and a term. Now we define the Δ-folding of a set of transition rules T, which we will write as $T \downarrow \Delta$. Let $r \in T$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $\{\boxed{f(t_1^*, ..., t_n^*) := t_0^*}\} \cup T^* \in T \downarrow \Delta$, where $t_i^* \in t_i \downarrow \Delta$ and $T^* \in T \setminus \{r\} \downarrow \Delta$. If $r \equiv \boxed{\textit{if } b \textit{ then } D}$ and $b \restriction \Delta \notin \{true, false\}$ then $\{\boxed{\textit{if } b^* \textit{ then } D^*}\} \cup T^* \in T \downarrow \Delta$, where $b^* \in b \downarrow \Delta, D^* \in D \downarrow \Delta$ and $T^* \in T \setminus \{r\} \downarrow \Delta$ Note, that $T \downarrow \Delta$ is the set of all possible foldings of the rules in T.

Theorem: Let $\Psi =< \sigma, S, T, \mathcal{I}_0 >$. Let Δ be a set of macro definitions and $T^* \in T \downarrow \Delta$. $< \sigma, S, T^*, \mathcal{I}_0 >$ is operationally equivalent to Ψ.

Proof: In the folded algebra the same updates are done as before, we only changed the structure of the terms, not their interpretation, i.e. the value they evaluate to. The operational equivalence follows by the same argument used for the proofs in the previous sections. □

Clearly, in practice we are interested in one set of folded rules. Thus in an implementation we would have to choose one $T^* \in T \downarrow \Delta$. The choice can be based on heuristics. Both, folding and unfolding transformations did only change the terms occuring in rules. Next we will address transformations, which change the structure of a set of rules.

Flattening Next we consider a simple transformation, which is helpful to prepare a set of rules to apply other transformations. Let C be a set of rules, then we construct the set of flat rules $\mathcal{F}(C)$ as follows. For each $r \in C$ we have: If $r \equiv \boxed{\textit{if } b_1 \textit{ then } D} \in C$ then $\{\boxed{\textit{if } b_1 \textit{ then } u} : u \in D \text{ is function update } \} \cup \{\boxed{\textit{if } b_1 \& b_2 \textit{ then } u} : \boxed{\textit{if } b_2 \textit{ then } u} \in \mathcal{F}(D)\} \subseteq \mathcal{F}(C)$. If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ then $r \in \mathcal{F}(C)$. For this construction to be semantics preserving, the interpretation of & has to be $\forall \tilde{a} \in S : \mathcal{I}(\&)(\tilde{a}) = \begin{cases} true & if\ \tilde{a} = (true, true) \\ false & otherwise \end{cases}$

Note that in the definition of a computation of an EvA, we defined *updates*, such that the rules of a guarded update are only considered, if the condition evaluates

to *true*. Flattening and its inverse transformation ("crushing"), can be used to restructure a set of rules, e.g.: $\{if\ b_1\ then\ \{u_1,\ if\ b_2\ then\ u_2\},\ if\ b_1\ then\ u_3\}$ can be transformed into $\{if\ b_1 \& b_2\ then\ u_2,\ if\ b_1\ then\ \{u_1, u_3\}\}$

Pass Separation Now we will classify dynamic functions as compile-time or run-time functions. The value of a compile-time function is known, before that of a run-time function, e.g. in an interpreter we might consider the program as compile-time data and the input to the program as run-time data. The idea is now to classify the rules: There is one group of rules, which depend only on compile-time functions and the remaining rules depend on compile-time or run-time functions. In practice we consider some of the external functions not to be known before run-time. Since other dynamic functions can depend on these functions, we have to classify these dynamic functions as run-time functions, too. In the literature on partial evaluation (e.g. [JGS93]) this process is called binding-time analysis.

Classification of Functions: Let R be the initial set of run-time functions and $\Psi = < \sigma, S, T, \mathcal{I}_0 >$. Now we classify the functions in S as follows:

1. Let $R' = R$
2. For all $r \in \mathcal{F}(T)$: If $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ and there is a function name $g \in R'$, such that g occurs at least in one of the terms $t_0, ..., t_n$, then $f \in R'$. If $r \equiv \boxed{if\ b\ then\ f(t_1, ..., t_n) := t_0}$ and there is a function name $g \in R'$, such that g occurs at least in one of the terms $b, t_0, ..., t_n$, then $f \in R'$.
3. If $R' = R$ then return R else set $R := R'$ and goto 2

Now the set of all compile-time functions is just $C = \sigma - R$. Note, that all static functions are classified as compile-time functions. The classification of functions terminates in time $O(|\sigma|)$, because in each iteration the $|R'|$ decreases and $|R'| < |\sigma|$.

Classification of Rules: Next we have to classify rules as compile- or run-time rules: $r \in T$ is a run-time rule, if $r \equiv \boxed{f(t_1, ..., t_n) := t_0}$ and there occurs at least one run-time function in one of the terms $t_0, ..., t_n$, or if $r \equiv \boxed{if\ b\ then\ D}$ and there occurs at least one run-time function in b or there is a run-time rule in D. Otherwise r is a compile-time rule. This classification of rules terminates in time $O(|T|)$.

For the pass separation transformation, we require that the top-level conditions in the run-time rules are *mutually exclusive*, i.e. if $\{\boxed{if\ b_1\ then\ u_1}, \ldots, \boxed{if\ b_n\ then\ u_n}\}$ is the set of all run-time rules in T, then we require: for all interpretations $\mathcal{I} \in reach(\mathcal{I}_0) : eval(b_k, \mathcal{I}) = true \Rightarrow$ for all $i \neq k : eval(b_i, \mathcal{I}) = false$

An evolving algebras is *separable*, if the top-level conditions of the run-time rules are mutually exclusive and consist of compile-time functions only, and if there occurs no term $f(t_1, ..., t_n)$ in any of the run-time rules, where f is a dynamic compile-time function and a run-time function occurs in at least on of the t_i.

Now we construct two evolving algebras: one which generates a program, and one which executes this program. In the following we assume, that the usual non-destructive list functions $(cons, fst, rest, reverse, nth, islist)$ are static functions in the evolving algebra and that it is separable. For each run-time rule $\boxed{if\ b\ then\ D}$ in T let $i \in S$ be a new instruction and add the following rules to T_c and T_e:

compilation: $\boxed{if\ b\ then\ D^C \cup \{prg := cons(cons(i, args), prg)\}} \in T_c$

execution: $\boxed{if\ islist(prg) \& fst(fst(prg)) = i\ then\ \hat{D}^K} \in T_e$

where D^C is the set of compile-time rules in D, D^R is the set of run-time rules in D and $args = [a_1, ..., a_m]$ is the list of all maximal subterms occurring in D^R, which only consist of compile-time functions. $\tilde{D}^R$ is obtained from D^R by replacing every occurrence of a_i by $nth(i+1, fst(prg))$. Furthermore the *islist* function yields true, if its argument is a non-empty list. Finally we have $\boxed{if\ islist(prg)\ then\ prg := rest(prg)} \in T_e$ and all compile-time rules are elements of T_c. Obviously splitting the rule set T can be done in time $O(|T|)$. Now we define the following evolving algebras [2] : $\Psi_c =< \sigma \cup \{prg\}, S, T_c, \mathcal{I}_0^c >$ where $\mathcal{I}_0^c|_\sigma = \mathcal{I}_0$ and $\mathcal{I}_0^c(prg) = nil$ and $\Psi_{\mathcal{I}_m^c} =< \sigma \cup \{prg\}, S, T_e, \mathcal{I}_0^e >$ where $\mathcal{I}_0^e|_\sigma = \mathcal{I}_m^c|_\sigma$ and $\mathcal{I}_0^e(prg)() = \mathcal{I}_m^c(reverse)(\mathcal{I}_m^c(prg)())$. We call the algebra executing the program $\Psi_{\mathcal{I}_m^c}$ to make explicit, that it depends on the terminal state of the compiling algebra. Taking the time complexities of all phases of the pass separation into account, the transformation needs time $O(max(|\sigma|, |T|))$

Theorem: If $\mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_m$ is a computation in Ψ, then in the compiling algebra Ψ_c there exists a computation $\mathcal{I}_0^c \xrightarrow{\Psi_c} \mathcal{I}_m^c$ and in the executing algebra $\Psi_{\mathcal{I}_m^c}$ there exists a computation $\mathcal{I}_0^e \xrightarrow{\Psi_{\mathcal{I}_m^c}} \mathcal{I}_q^e$, where $q \leq m$. Furthermore we have $\mathcal{I}_q^e|_\sigma = \mathcal{I}_m$.

Proof: First we note, that no dynamic compile-time functions occur in T_e. Let C be the set of compile-time rules in T and R be the set of run-time rules. We will prove the five stronger properties

Lemma: The following properties hold: (1) $\forall j \in \{0, ..., m\} : \mathcal{I}_j^c|_R = \mathcal{I}_0|_R$ and (2) $\forall j \in \{0, ..., m\} : \mathcal{I}_j^c|_C = \mathcal{I}_j|_C$ and (3) $\forall j \in \{0, ..., q\} : \mathcal{I}_j^e|_C = \mathcal{I}_m|_C$ and (4) $\exists i_0, ..., i_q \leq m, i_k < i_{k+1} : \forall j \in \{0, ..., q\} : \mathcal{I}_j^e|_R = \mathcal{I}_{i_j}|_R$ and (5) $\mathcal{I}_q^e|_R = \mathcal{I}_m|_R$

(1): Clearly $\mathcal{I}_j^c|_R = \mathcal{I}_0|_R$, because there is no update to a run-time function in any of the rules in T_c.

(2): This part follows by induction on the steps of the computations:

$j = 0$: by definition we have: $\mathcal{I}_0^c|_\sigma = \mathcal{I}_0$ and as a consequence $\mathcal{I}_0^c|_C = \mathcal{I}_0|_C$

$j + 1$: In T_c are only updates to compile-time functions, because any rule containing an update to a run-time function is considered a run-time rule. As a consequence, for all updates u to compile-time functions we have $u \in updates(T, \mathcal{I}_j) \Leftrightarrow u \in updates(T_c, \mathcal{I}_j^c)$, because the conditions,which have to be true for adding u to $updates(T_c, \mathcal{I}_j^c)$ contain only compile-time functions, for which we know, that $\mathcal{I}_j|_C = \mathcal{I}_j^c|_C$ by the induction hypothesis. By the definition of a computation step it follows, that $\mathcal{I}_{j+1}|_C = \mathcal{I}_{j+1}^c|_C$.

(*): Furthermore we know, that only the guard of one run-time rule can be true (mutually exclusive rules). In this case prg is updated:

$\mathcal{I}_{j+1}^c(prg)() = \mathcal{I}_j^c(cons)([i, eval(a_1, \mathcal{I}_j^c), ..., eval(a_n, \mathcal{I}_j^c)], \mathcal{I}_j^c(prg))$.

By the induction hypothesis it follows, that $eval(a_k, \mathcal{I}_j^c) = eval(a_k, \mathcal{I}_j)$

(3): Since T_e does not contain an update to a compile-time function, we have $\mathcal{I}_j^e|_C = \mathcal{I}_m^c|_C$ and by (1) we have $\mathcal{I}_m^c|_C = \mathcal{I}_m|_C$.

(4): This part follows by induction on the steps of the computations:

$j = 0$: By definition we have: $\mathcal{I}_0^e|_\sigma = \mathcal{I}_m^c|_\sigma$ and by (1) $\mathcal{I}_m^c|_R = \mathcal{I}_0|_R$. Thus it follows, that $\mathcal{I}_0^e|_R = \mathcal{I}_0|_R$ and $i_0 = 0$. $j + 1$:

case 1: There is a computation step $\mathcal{I}_{i_{j+1}-1} \xrightarrow{\Psi} \mathcal{I}_{i_{j+1}}$, where $i_j < i_{j+1}$ and a top-level condition of a run-time rule evaluates to *true*. Then this guard also evaluates to *true* in the step $\mathcal{I}_{i_{j+1}-1}^c \xrightarrow{\Psi_c} \mathcal{I}_{i_{j+1}}^c$ and $[i, \tilde{a_1}, ..., \tilde{a_k}]$ is cons'ed to prg. The rules involved

[2] The restriction of a function f to a set A is defined as $f|_A = \{(a, f(a)) : a \in A\}$.

are: $\boxed{\textit{if } b \textit{ then } D} \in T$, $\boxed{\textit{if } b \textit{ then } D^c \cup \{prg := cons(cons(i, args), prg)\}} \in T_c$
and $\boxed{\textit{if } islist(prg) \& fst(fst(prg)) = i \textit{ then } \tilde{D}^R} \in T_r$
Since in $\Psi_{\mathcal{I}^c_m}$ the value of prg has been reversed and at each step $prg := rest(prg)$ is executed, it is easy to see, that $[i, \tilde{a_1}, ..., \tilde{a_k}]$ is the first element of prg in $\mathcal{I}^c_j$. As a consequence we have: $updates(T_c, \mathcal{I}^c_j) = updates(\tilde{D}^R, \mathcal{I}^c_j) \cup \{eval(prg := rest(prg))\}$
Since there is no other update to a run-time function in an intermediate step, we have $\mathcal{I}_{i_j}|_R = \mathcal{I}_{i_{j+1}-1}|_R$ and by the induction hypothesis, $\mathcal{I}^c_j|_R = I_{i_j}|_R$. Now it follows, that $updates(\tilde{D}^R, I^c_j) = updates(\tilde{D}^R, \mathcal{I}_{i_{j+1}-1} \cup \mathcal{I}^c_j|_{\{prg\}})$ and by (*) we know that $\tilde{a_k} = eval(a_k, I^c_{i_{j+1}-1}) = eval(a_k, I_{i_{j+1}-1})$ and thus $updates(\tilde{D}^R, \mathcal{I}_{i_{j+1}-1} \cup \mathcal{I}^c_j|_{\{prg\}}) = updates(D, \mathcal{I}_{i_{j+1}-1})$. And by the definition of a computation step: $\mathcal{I}^c_{j+1}|_R = \mathcal{I}_{i_{j+1}}|_R$.
case 2: There is no such computation step. Then $j = q$ and we conclude, that $\mathcal{I}_q|_R = \mathcal{I}_m|_R$ and by the induction hypothesis $\mathcal{I}^c_q|_R = \mathcal{I}_q|_R$ and thus $\mathcal{I}^c_q|_R = \mathcal{I}_m|_R$, which is point (5) of the above lemma. □

An Example Next we will apply pass separation to an interpreter for simple arithmetic expressions (E — $VAR \mid INT \mid (E\ OP\ E)$). We assume, that `in` is a list of symbols representing an expression, e.g. `in` = ["(", X, +, "(", 7, *, 3, ")", ")"]. Furthermore `env` maps variable names to values, e.g. `env(X)` = 3.

```
if islist(in) then
 { if fst(in)="(" then in:=rest(in),
   if isop(fst(in)) { opstack:=cons(fst(in),opstack), in:=rest(in) },
   if isint(fst(in)) then { estack:=cons(fst(in),estack), in:=rest(in) },
   if isvar(fst(in)) then { estack:=cons(env(fst(in)),estack), in:=rest(in) },
   if fst(in)=")" then { opstack:=rest(opstack),
                         estack:=cons(apply(fst(opstack),snd(estack),
                         fst(estack)), rest(rest(estack))), in:=rest(in) } }
```

Using flattening the above transition rule can be converted into a set of transition rules, which is more suitable for applying the pass separation transformation:

```
if islist(in) then in:=rest(in),
if islist(in) & isop(fst(in)) then opstack:=cons(fst(in),opstack),
if islist(in) & isint(fst(in)) then estack:=cons(fst(in),estack),
if islist(in) & isvar(fst(in)) then estack:=cons(env(fst(in)),estack),
if islist(in) & (fst(in)="(") then  opstack:=rest(opstack),
if islist(in) & (fst(in)=")") then estack:=cons(apply(fst(opstack),snd(estack),
                                        fst(estack)), rest(rest(estack)))
```

We assume, that `in` is known at compile-time and `env` not before run-time and classify functions and rules as described above. Now we can apply the pass separation transformation[3] to generate a simple compiler

```
if islist(in) then
 { in:=rest(in),
   if isop(fst(in)) then opstack:=cons(fst(in),opstack),
   if isint(fst(in)) then prg:=cons(cons("pushint",fst(in)),prg),
   if isvar(fst(in)) then prg:=cons(cons("pushvar",fst(in)),prg),
   if fst(in)=")" then opstack:=rest(opstack),
   if fst(in)=")" then prg:=cons(cons("app",fst(opstack)),prg) }
```

[3] To increase readability we applied the "crushing" transformation, see the conditions `islist(in)` and `islist(prg)`.

and an abstract target machine

```
if islist(prg) then
 { if fst(fst(prg))="pushint" then estack:=cons(rest(fst(prg)),estack),
   if fst(fst(prg))="pushvar" then estack:=cons(env(rest(fst(prg))),estack),
   if fst(fst(prg))="app" then estack:=cons(apply(rest(fst(prg)),
                              snd(estack),fst(estack)), rest(rest(estack))),
   prg := rest(prg) }
```

For example given the value `in` = ["(", X, +, "(", 7, *, 3, ")", ")"] at compile time, the compiler will generate the abstract machine program: `prg` =[`(pushvar X)`, `(pushint 7)`, `(pushint 3)`, `(app *)`, `(app +)`]. The above example shows, that pass separation can be used for semantics-directed compiler generation.

Implementation All transformations in this paper can be automated, but testing the mutual exclusion of run-time rules is not even decidable. Nevertheless heuristics can be used to decide, whether the conditions are mutually exclusive. Even checking mutual exclusion at run-time is co-NP complete ([Gur91]).

3 Other Work

In [JS86] the authors use pass separation to generate a compiler and an abstract machine for a functional language from a specification of an abstract interpreter. The transformations are very sophisticated, but they are neither formally defined, nor is it likely that they can be automated. In [Han91] John Hannan defines a pass separation transformation of a very restricted class of term rewriting systems. From an interpreter for a simple functional language, which he calls the CLS machine, he derives a compiler and an abstract machine similar to the CAM ([CCM85]).

4 Conclusions

We defined evolving algebras in automata theoretic terms and used this definition as a basis to define some transformations on evolving algebras and prove some essential properties of these. The pass separation transformation can be used to split simple interpreters into compilers and abstract machines.

References

[BR92] Egon Börger and Dean Rosenzweig. The WAM – Definition and Compiler Correctness. Technical Report TR-14/92, Universita Degli Studi Di Pisa, Pisa, Italy, 1992.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorial Abstract Machine. In *Proceedings of FPCA '85*. Springer, LNCS 201, 1985.

[Gur91] Yuri Gurevich. Evolving Algebras: a tutorial introduction. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–284, 1991.

[Han91] J. Hannan. Staging Transformations for Abstract Machines. In *Partial Evaluation and Semantics-Based Program Manipulation*. SigPlan Notices, vol. 26(9), 1991.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[JS86] U. Jørring and W.L. Scherlis. Compilers and Staging Transformations. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 51–57

Relative Completeness

Rade Doroslovački[1], Jovanka Pantović[1], Ratko Tošić[2], Gradimir Vojvodić[2]

[1] Institute of Mathematics, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
[2] Faculty of Engineering, University of Novi Sad
Trg D. Obradovića 3, 21000 Novi Sad, Yugoslavia

Abstract. This paper presents the relative completeness with respect to some specified clones that we find to be of particular interest, considered in [4,5,9].

1 Notation and Preliminaries

To support the systematic design of biomolecular computing systems, an algebraic system called set logic is developed. As it is interesting to consider the question: If one has all Boolean functions, how much does one need to have functional completeness? So also it is natural extension to ask, if one has a specified subset of multiple-valued logic functions, how much does one need to have functional completeness?

By $\mathbf{N}$ we denote the set of positive integers: $\{1, 2, \ldots\}$. For $k, n \in \mathbf{N}$ let $E_k = \{0, 1, \ldots, k-1\}$, let $P_k^{(n)}$ denote the set of all maps $E_k^n \to E_k$, and let $P_k = \bigcup_{n \in \mathbf{N}} P_k^{(n)}$. We say that f is an *i-th projection of arity n* $(1 \leq i \leq n)$ iff $f \in P_k^{(n)}$ and f satisfies the identity $f(x_1, \ldots, x_n) \approx x_i$. Let π_i^n denote the i-th projection of arity n. Let Π_k denote the set of all the projections over E_k.

$F \subseteq P_k$ is *clone of operations on set* E_k (or *clone* for short) iff $\Pi_k \subseteq F$ and F is closed with respect to superposition. All the clones on E_k form a lattice which shall be referred to as $\mathcal{L}_k$. For $F \subseteq P_k$, $\langle F \rangle_{\mathrm{CL}}$ stands for the clone generated by F. $F \subseteq P_k$ is complete iff $\langle F \rangle_{\mathrm{CL}} = P_k$.

Let $\varrho \subseteq E_k^h$ be a h-ary relation and $f \in P_k^{(n)}$. We say that f preserves ϱ iff for each n h-tuples $(a_{11}, \ldots, a_{1h}), \ldots, (a_{n1}, \ldots, a_{nh})$ from ϱ we have

$$(f(a_{11}, \ldots, a_{n1}), \ldots, f(a_{1h}, \ldots, a_{nh})) \in \varrho.$$

$\mathrm{Pol}\,\varrho$ is the set of all $f \in P_k$ that preserve ϱ. For $F \subseteq P_k$, $\mathrm{Inv}\,F$ denotes the set of all the relations being preserved by each $f \in F$.

Definition 1.1 *Let $C \in \mathcal{L}_k$ be a clone and $F \subseteq P_k$. F is complete relative to C (or C-complete) iff $\langle F \cup C \rangle_{\mathrm{CL}} = P_k$.*

Thus, relative completeness is a generalization of weak completeness, introduced in [7]. On the other hand, relative completeness is a generalization of (usual) completeness, because every complete set F is Π_k-complete.

The following easy theorem gives a necessary and sufficient condition for F to be C-complete. It is analogous to the Post's completeness criterion.

Theorem 1.1 *Let C be a clone and $\{M_1, \ldots, M_s\}$ be the set of all the maximal clones containing C:*

$$\{M_1, \ldots, M_s\} = \{D \in \mathcal{L}_k : D \text{ is maximal and } D \supseteq C\}.$$

$F \subseteq P_k$ is complete relative to C iff $F \setminus M_i \neq \emptyset$ for all $i \in \{1, \ldots, s\}$. □

Therefore, the problem of determining whether a set F is relatively complete reduces to determining all the maximal clones that contain F.

We shall introduce some special sets of relations:

R_1 the set of all bounded partial orders on E_k
R_2 the set of selfdual relations, i.e. relations of the form $\{(x, s(x)) : x \in E_k\}$, where s is a fixed point free permutation of prime order (i.e. $s^p = \mathrm{id}$ for some prime p)
R_3 the set of affine relations, i.e. relations of the form $\{(a, b, c, d) \in E_k^4 : a * b = c * d\}$, where $(E_k, *)$ is a p-elementary Abelian group (p prime)
R_4 the set of all nontrivial equivalence relations on E_k
R_5 the set of all central relations on E_k
R_6 the set of all h-regular relations on E_k ($h \geq 3$)

Theorem 1.2 [6] *A clone M is maximal iff there is a $\varrho \in R_1 \cup \ldots \cup R_6$ such that $M = \mathrm{Pol}\, \varrho$* □

2 Relative Completeness With Respect to Minimum and Complement

Let $\min(x, y)$ denote binary operation whose result is the least of x and y (where $x, y \in E_k$), and $\overline{x} = (k-1) - x$. Let $K = \langle \min, \overline{x} \rangle_{\mathrm{CL}}$. It is important to observe that $\max(x, y) = \overline{\min(\overline{x}, \overline{y})}$, which implies that $\max \in K$.

Lemma 2.1 *$\overline{x}$ preserves no $\varrho \in R_1$.*

Corollary 2.1 *$K \setminus \mathrm{Pol}\, \varrho \neq \emptyset$ for all $\varrho \in R_1$.* □

Lemma 2.2 min *preserves no $\varrho \in R_2$.*

Corollary 2.2 *$K \setminus \mathrm{Pol}\, \varrho \neq \emptyset$ for all $\varrho \in R_2$.* □

Lemma 2.3 min *preserves no* $\varrho \in R_3$.

Corollary 2.3 $K \setminus \mathrm{Pol}\,\varrho \neq \emptyset$ *for all* $\varrho \in R_3$. □

Lemma 2.4 *Let* $\varrho \in \mathrm{Inv}\,K \cap R_4$, *let* $(a,b) \in \varrho$ *and* $a < b$ *(where "<" is usual linear order on* E_k*:* $0 < 1 < \ldots < k-1$*). For each* $c \in E_k$, *if* $a < c < b$ *then* $(a,c) \in \varrho$.

Lemma 2.5 *Let* $\varrho \in \mathrm{Inv}\,K \cap R_4$. *Equivalence classes of* ϱ *are intervals with respect to usual ordering on* E_k.

Lemma 2.6 *If* $[a,b]$ *is an equivalence class of some* $\varrho \in \mathrm{Inv}\,K \cap R_4$, *then* $\left[\overline{b},\overline{a}\right]$ *is also an equivalence class of* ϱ.

Corollary 2.4 *Let* $\varrho \in \mathrm{Inv}\,K \cap R_4$. *Equivalence classes of* ϱ *are disjoint intervals covering* E_k *arranged in such a way that* $\frac{k-1}{2}$ *is the center of symmetry of the figure formed by the intervals.*□

Lemma 2.7 *If* $\varrho \in R_4$ *is an equivalence relation such that equivalence classes of* ϱ *are disjoint intervals which cover* E_k *and which are arranged in such a way that* $\frac{k-1}{2}$ *is the center of symmetry of the figure formed by the intervals, then* $\varrho \in \mathrm{Inv}\,K$.□

Corollary 2.5 $\mathrm{card}(\mathrm{Inv}\,K \cap R_4) = 2^{\lfloor \frac{k}{2} \rfloor} - 2$

Lemma 2.8 *Consider* $\varrho \in R_5^{(1)}$. $\varrho \in \mathrm{Inv}\,K$ *iff* $(\forall x \in E_k)(x \in \varrho \Rightarrow \overline{x} \in \varrho)$

Corollary 2.6 $\mathrm{card}(\mathrm{Inv}\,K \cap R_5^{(1)}) = 2^{\lceil \frac{k}{2} \rceil} - 2$

Corollary 2.7 *If* $\varrho \in R_5^{(1)}$ *and* $t = \mathrm{card}(\varrho)$, *then* $\mathrm{card}(P_k^{(n)} \cap \mathrm{Pol}\,\varrho) = t^{t^n} k^{k^n - t^n}$

Lemma 2.9 *If* $\varrho \in R_5^{(h)}$ *and* $h > 2$ *then* $K \setminus \mathrm{Pol}\,\varrho \neq \emptyset$.

Lemma 2.10 *Let* $\varrho \in \mathrm{Inv}\,K \cap R_5^{(2)}$.

(*a*) *If* c *is a central element of* ϱ, *then* $\overline{c}$ *too is central element of* ϱ
(*b*) *If* a *and* b *are central element of* ϱ *and* $a < b$ *then every element from the interval* $[a,b]$ *is a central element of* ϱ

Corollary 2.8 *Let* $\varrho \in \mathrm{Inv}\,K \cap R_5^{(2)}$. *The set of all the central elements of* ϱ *is an interval of the form* $[c,\overline{c}]$ *which is a strict subset of* E_k. □

Lemma 2.11 $\mathrm{card}(\mathrm{Inv}\,K \cap R_5^{(2)}) = 2^{\lfloor \frac{k-1}{2} \rfloor} - 1$

Lemma 2.12 max *preserves no* $\varrho \in R_6$.

Corollary 2.9 $K \setminus \mathrm{Pol}\,\varrho \neq \emptyset$ *for all* $\varrho \in R_6$

3 Relative Completeness With Respect to Two Negations

Consider the following two negations:

$$x^{\sim} = x + 1 \pmod{k}$$
$$x^{(k-1)} = \begin{cases} k-1, & x = k-1 \\ 0, & x \neq k-1 \end{cases}$$

and the clone generated by these negations: $U = \langle x^{\sim}, x^{(k-1)} \rangle_{\mathrm{CL}}$.

Lemma 3.1 *$x^{\sim}$ preserves no $\varrho \in R_1$.*

Corollary 3.1 *$U \setminus \operatorname{Pol} \varrho \neq \emptyset$, for all $\varrho \in R_1$.* □

Lemma 3.2 *$x^{(k-1)}$ preserves no $\varrho \in R_2$.*

Corollary 3.2 *$U \setminus \operatorname{Pol} \varrho \neq \emptyset$, for all $\varrho \in R_2$.* □

Lemma 3.3 *$x^{(k-1)}$ preserves no $\varrho \in R_3$.*

Corollary 3.3 *$U \setminus \operatorname{Pol} \varrho \neq \emptyset$, for all $\varrho \in R_3$.* □

Lemma 3.4 *For each $\varrho \in R_4$ there is $f \in \{x^{\sim}, x^{(k-1)}\}$ such that f does not preserve ϱ.*

Corollary 3.4 *$U \setminus \operatorname{Pol} \varrho \neq \emptyset$ for all $\varrho \in R_4$.* □

Lemma 3.5 *$x^{\sim}$ preserves no $\varrho \in R_5$.*

Corollary 3.5 *$U \setminus \operatorname{Pol} \varrho \neq \emptyset$, for all $\varrho \in R_5$.* □

Lemma 3.6 *Consider an one-element h-regular family $\tau = \{\Theta\}$ and denote by ϱ the regular relation determined by τ. $x^{\sim}$ preserves ϱ iff Θ is an equivalence relation whose blocks are*

$$\{\{0, h, \ldots, (r-1)h\}, \{1, h+1, \ldots, (r-1)h+1\}, \ldots, \{h-1, 2h-1, \ldots, rh-1\}\}$$

where $r = k/h$.

Corollary 3.6 *There exist at least* $\operatorname{card}(\{h : h|k\} \setminus \{1, 2\})$ *maximal clones which contain the clone U.* □

Theorem 3.1 *If $k \leq 8$ then there exist exactly* $\operatorname{card}(\{h : h|k\} \setminus \{1, 2\})$ *maximal clones which contain the clone U.* □

We conclude with an open problem that could clarify the obscurity of case R_6: Is there a $\varrho \in R_6$ such that $\operatorname{ar}(\varrho) > 1$ and $x^{\sim}$ preserves ϱ?

4 Relative Completeness With Respect to Transpositions

Consider the following transpositions ([2],*theorem* $8, p.54$) on E_k:

$$g_i(x) = \begin{cases} i \ , x = 0 \\ 0 \ , x = i \\ x \ , otherwise. \end{cases}$$

and the clone generated by them:

$$\mathcal{C} = \langle \bigcup_{i=1}^{k-1} g_i \rangle_{\mathrm{CL}}.$$

Lemma 4.1 *For each $\varrho \in R_1$ there is $f \in \mathcal{C}$ such that f does not preserve ϱ.*

Corollary 4.1 *$\mathcal{C} \setminus \mathrm{Pol}\,\varrho \neq \emptyset$, for all $\varrho \in R_1$.*

Lemma 4.2 *For each $\varrho \in R_2$ there is $f \in \mathcal{C}$ such that f does not preserve ϱ.*

Corollary 4.2 *$\mathcal{C} \setminus \mathrm{Pol}\,\varrho \neq \emptyset$, for all $\varrho \in R_2$.*

Lemma 4.3 *(a) If $k > 4$ then for each $\varrho \in R_4$ there is $f \in \mathcal{C}$ such that f does not preserve ϱ.*
(b) If $k \in \{3, 4\}$ then g_i preserves ϱ for each $i \in \{1, ..., k-1\}$.

Corollary 4.3 *(a) If $k > 4$ then $\mathcal{C} \setminus \mathrm{Pol}\,\varrho \neq \emptyset$, for all $\varrho \in R_3$.*
(b) If $k \in \{3, 4\}$ then $\mathcal{C} \subseteq Pol\varrho$, for $\varrho \in R_3$.

Lemma 4.4 *For each $\varrho \in R_4$ there is $f \in \mathcal{C}$ such that f does not preserve ϱ.*

Corollary 4.4 *$\mathcal{C} \setminus \mathrm{Pol}\,\varrho \neq \emptyset$ for all $\varrho \in R_4$.*

Lemma 4.5 *For each $\varrho \in R_5$ there is $f \in \mathcal{C}$ such that f does not preserve ϱ.*

Corollary 4.5 *$\mathcal{C} \setminus \mathrm{Pol}\,\varrho \neq \emptyset$, for all $\varrho \in R_5$.*

Lemma 4.6 *(a) For each $\varrho \in R_6, 2 < h < k$ there is $f \in \mathcal{C}$ such that f does not preserve ϱ.*
(b) The k-ary relation $\varrho \in R_6$ is preserved by g_i for each $i \in \{1, ..., k-1\}$.

Corollary 4.6 *(a) $\mathcal{C} \setminus \mathrm{Pol}\,\varrho \neq \emptyset$, for all $\varrho \in R_6, 2 < h < k$.*
(b) If $\varrho = E_k^k - P_{01\ldots(k-1)}$ then $\mathcal{C} \subseteq Pol\varrho$.

Theorem 4.1 *(a) If $k > 4$ then there is exactly 1 relative maximal clone with respect to $\mathcal{C}$.*
(b) If $k \in \{3, 4\}$ then there are exactly 2 relative maximal clones with respect to $\mathcal{C}$.

Corollary 4.7 *If $k > 4$ then $\mathcal{F}$ is relative complete with respect to $\mathcal{C}$ iff it contains an essential function.*

5 Relative Completeness With Respect to Two Unary Functions

Consider the following operations ([2],*theorem* $9, p.54$), on E_k:

$$g(x) = \begin{cases} x & , 0 \leq x \leq k-3 \\ k-1 & , x = k-1 \\ k-2 & , x = k-2. \end{cases},$$

$$f(x) = x - 1 (mod\ k)$$

and the clone generated by them: $\mathcal{C} = \langle \{g, f\} \rangle_{CL}$.

Lemma 5.1 *f preserves no $\varrho \in R_1$.*

Corollary 5.1 *$\mathcal{C} \setminus \mathrm{Pol}\, \varrho \neq \emptyset$. for all $\varrho \in R_1$.* □

Lemma 5.2 *For each $\varrho \in R_2$ there is $h \in \mathcal{C}$ such that h does not preserve ϱ.*

Corollary 5.2 *$\mathcal{C} \setminus \mathrm{Pol}\, \varrho \neq \emptyset$, for all $\varrho \in R_2$.* □

Lemma 5.3 *(a) If $k > 4$ then g preserves no $\varrho \in R_3$.*
(b) If $k \in \{3, 4\}$ then $\{f, g\}$ preserve $\varrho \in R_3$.

Corollary 5.3 *(a) If $k > 4$ then $\mathcal{C} \setminus \mathrm{Pol}\, \varrho \neq \emptyset$, for all $\varrho \in R_3$.*
(b) If $k \in \{3, 4\}$ then $\mathcal{C} \subseteq Pol\varrho$, for $\varrho \in R_3$.
□

Lemma 5.4 *For each $\varrho \in R_4$ there is $h \in \{f, g\}$ such that h does not preserve ϱ.*

Corollary 5.4 *$\mathcal{C} \setminus \mathrm{Pol}\, \varrho \neq \emptyset$ for all $\varrho \in R_4$.* □

Lemma 5.5 *f preserves no $\varrho \in R_5$.*

Corollary 5.5 *$\mathcal{C} \setminus \mathrm{Pol}\, \varrho \neq \emptyset$, for all $\varrho \in R_5$.* □

Lemma 5.6 *(a) For each $\varrho \in R_6, 2 < h < k$ there is $h \in \{f, g\}$ such that h does not preserve ϱ.*
(b) The k-ary relation $\varrho \in R_6$ is preserved by $\{f, g\}$.

Corollary 5.6 *(a) $\mathcal{C} \setminus \mathrm{Pol}\, \varrho \neq \emptyset$, for all $\varrho \in R_6, 2 < h < k$.*
(b) If $\varrho = E_k^k - P_{01\ldots(k-1)}$ then $\mathcal{C} \subseteq Pol\varrho$.

Theorem 5.1 *(a) If $k > 4$ then there is exactly 1 relative maximal clone with respect to $\mathcal{C}$.*
(b) If $k \in \{3, 4\}$ then there are exactly 2 relative maximal clones with respect to $\mathcal{C}$.

Corollary 5.7 *If $k > 4$ then $\mathcal{F}$ is relative complete with respect to $\mathcal{C}$ iff it contains an essential function.*

References

1. Demetrovics J., Reischer C., Simovici D., Stojmenović I., *Enumeration of Function and Bases of Three-Valued Set Logic under Compositions with Boolean Functions*, Proc. of the 24th Int. Symp. on Multiple-Valued Logic, 1994, 164–171
2. Jablonskiĭ S.V., *Introduction to Discrete Mathematics (Russian)*, Nauka, Moskva, 1979.
3. Pogosyan G.,Nozaki A.,Miyakawa M.,Rosenberg I.G., *Hereditary Clones of Multiple Valued Logic Algebra*, Proc.of the 24th Int.Symp.on Multiple-Valued Logic,1994,306-313.
4. Vojvodić G., Pantović J., Tošić R., *Relative Completeness with Respect to Transpositions* (unpublished).
5. Pantović J., Tošić R., Vojvodić G., *Relative Completeness with Respect to Two Unary Functions* (unpublished).
6. Rosenberg I.G., *Completeness Properties of Multiple-Valued Logic Algebra* , In D.C. Rhine Ed. Computer Science and Multiple-Valued Logic: Theory and Application, North-Holland, p.144-186.
7. Simovici D., Stojmenović I., Tošić R., *Functional Completeness and Weak Completeness in Set Logic*, Proceedings of 23rd International Symposium on Multiple-Valued Logic, Sacramento, 1993, pp. 251–256
8. Stojmenović I.,*Completeness Criteria in Many-Valued Set Logic Under Compositions with Boolean Functions*,Proc.of the 24th Int. Symp.on Multiple-Valued Logic,1994,177-183.
9. Tošić R.,Vojvodić G., Mašulović D., Doroslovački R., Rosić J., *Two Examples of Relative Completeness*, Multi.Val.Logic, Vol.2, pp.67-78.

This article was processed using the LaTeX macro package with LLNCS style

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '97
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1–4, 1997, PP. 59–66

Deductive database development with optimization

Mr Dragana Djurić, project-programmer, Soko-Štark, Beograd
Dr Gordana Pavlović-Lažetić, associate professor, Faculty of mathematics, Beograd

Abstract

In this paper an approach to deductive database development with optimization is proposed. A logical (Prolog-like) interface to a relational database is designed and implemented. Query evaluation is performed in two steps: first, materialization of derived relations present in the query, by the most efficient method from a given set of methods; second, translation of a Prolog-like query into an SQL statement over base relations, and its execution by an SQL processor. Decision about the most efficient method for materialization is made according to three criteria – number of successful inferences, input/output cost and the number of simple steps in a corresponding algorithm. A programming system is implemented in Visual Basic environment that includes three compiling methods for derived relation materialization – Naive, Semi-Naive and Henschen-Naqvi, as well as a component for their efficiency estimation. A classification of derived relations is proposed according to efficiency of specific methods. This syntactic characterization of derived relations enables estimation, without prior materialization, of the most efficient evaluation method for four significant classes of derived relations.

1 Introduction

Deductive databases provide for new facts to be deduced from other, explicitely given facts. Such a deduction becomes necessary because of limitations of relational model formalisms – relational algebra and relational calculus, in formulating, by a single expression, different sets of data from a database.

For example, a result of a transitive closure operation of a relation cannot be expressed by a single expression of the classical relational algebra or relational calculus. Different approaches to increasing expressive power of manipulative formalisms exist. Beside solutions offering nesting of a relational formalism into a host language [2], a significant approach is based on extending structural – and consequently manipulative parts of the model itself. This leads to a deductive model (and deductive database systems).

From the proof theory as a first order theory point of view, a deductive database consists of two components:

- Theory T whose axioms, besides general axioms that hold in the corresponding theory for classical relational databases, include the following two groups of axioms:

  (1) Elementary facts, i.e., a set of clauses of the form $P(c_1, \ldots, c_n)$, corresponding to a base table P in a relational database, and representing an EDB (Extensional DataBase) part of a deductive database.

(2) Deductive rules, i.e., a set of clauses of the form

$$R : -P_1, P_2, \ldots, P_n,$$

representing definition of the predicate R in terms of predicates P_1, P_2, ..., P_n. Deductive rules make an IDB (Intentional DataBase) part of a deductive database.

- Integrity constraints set of formulas, IC.

An answer to a query $W(x_1, \ldots, x_n)$, where $x_1, \ldots x_n$ are free variables in W, is a set of tuples $(c_{i_1}, \ldots, c_{i_n})$ such that $T \models W(c_{i_1}, \ldots, c_{i_n})$. Deductive database satisfies integrity constraints of IC iff for every formula $\Phi \in$ IC, $T \models \Phi$.

Relations defined by both deductive rules and elementary facts are known as derived relations. Recursive relations in deductive databases may be defined by two deductive rules.

Problems and implementation of deductive databases have been principal research interests for many researchers for last two decades [1], [3]. Basic research tasks are design of efficient algorithms for recursive query evaluation, development of logical interfaces to relational databases, efficiency analysis of algorithms for query evaluation, etc.

This paper deals with a deductive database development through a Prolog-like interface to a relational database system. Compiling methods are chosen for evaluation and materialization of derived relations. A logical language is designed for defining an IDB part of a deductive database, and the corresponding language processor is implemented that analyses and translates derived relations and queris, from the logical language into SQL. Optimization component based on materialization and classification of derived relations is developed.

2 Recursive query evaluation methods and their efficiency

A recursive query evaluation strategy is defined by a class of rules it is applicable to, and by an algorithm for evaluation queries over such set of rules. If a strategy does not change neither IDB nor a query, it is called a method. If a strategy, prior to query evaluation by a method, performs a transformation of rules in order to optimize query evaluation, it is called a rewriting rules system.

In this paper we will consider compiling methods only, i.e., Naive, Semi-Naive and Henschen-Naqvi methods, applicable to linearly recursive range restricted rules (ones whose right side contains all the variables from the left side). Naive method is the simplest, and its algorithm, for a relation R and a query Q defined by the following clauses (S is a base relation, c is a constant)

$R(X,Y) : -S(X,Y).$
$R(X,Y) : -S(X,Z), R(Z,Y).$
$Q(X) : -R(c,X).$

keeps generating intermediate results representing unions of joins of the relation S and a previous intermediate result, until it reaches its "fixpoint".

Semi-Naive method is an improvement of the Naive method in that it calculates, in every iteration, just new tuples, and avoids repeated calculations of already calculated tuples.

Henschen-Naqvi method is quite complex, and in a special case represented by the following clauses defining "same level" relation and a query

$R(X,Y) : -u(X,XU), R(XU,YU), d(YU,Y).$
$R(X,Y) : -S(X,Y).$
$Q(X) : -R(c,X).$

it generates the answer to the query by calculating expression
$\{a\}.R + \{a\}.u.R.d + \{a\}.u.u.R.d.d + \ldots + \{a\}.u^n.R.d^n + \ldots,$
thus calculating relation degrees ($\{a\}.R$ is a set of all the y-s such that $R(a,y)$ holds).

Question of efficiency of a recursive query evaluation strategy is still an open question. An approach to comparing strategies is presented in [1] and is based on the following criteria: size of application domain, performace, and ease of implementation. Performance is measured by the number of successful inferences.

In this paper three compiling methods are compared according to different cost functions presented in section 4.

3 Derived relations and query definition language

Logical interface to a relational database is defined by the following syntax:

(1) <derived_relation> ::= <relation1> ":-" <relation2>{,<relation2>}^{2}
(2) <query> ::= <relstion3>{,<relation3>}^{2}
(3) <relation1> ::= <relation_name> (<argument1>{,<argument1>}^{1})
(4) <argument1> ::= <variable>[:<attribute_name>]
(5) <attribute_name> ::= <letter>{<letter> | <digit>}
(6) <relation_name> ::= <letter>{<letter> | <digit>}
(7) <variable> ::= <letter>
(8) <relation2> ::= <relation_name> (<argument2>{,<argument2>}^{1})
(9) <argument2> ::= <variable> | '<letter>{<letter>}'
(10) <relation3> ::= <relation_name> (<argument3>{,<argument3>}^{1})
(11) <argument3> ::= <variable> | <variable>* | '<letter>{<letter>}'

Language processor translates derived relation or query definitions into SQL. For example, the following definition of a derived relation R over a base relation $S(at1, at2)$

$R(X : at3, Y : at4) : -S(X,Y).$
$R(X : at3, Y : at4) : -S(X,Z), R(Z,Y).$

will be translated into two SELECT statements:

SELECT $s1.at1$ as $at3$, $s1.at2$ as $at4$
FROM S $s1$

SELECT $s1.at1$ as $at3$, $r2.at4$ as $at4$
FROM S $s1$, R $r2$
WHERE $s1.at2 = r2.at3$

Expressive power of the language is analogue to SQL SELECT-block with SELECT, FROM and WHERE clauses, without nested subqueries or aggregate functions.

4 Implementation

The programming system implemented in Visual Basic environment evaluates a query against the deductive database by materializing derived relations participating in it. When a derived relation definition is submitted, the system first analyses it, and, if correct, translates it into an SQL statement. Then it applies every method implemented to evaluate and materialize the relation and stores costs of materialization for every method in a sistem catalog. The costs are calculated according to defined criteria. Then materializad relations are dropped.

When a query definition is submitted, the system translates it into an SQL statement and decides, based on the costs from the catalog, which method to apply for materialization of every derived relation participating in the query. After materialization has been done, the SQL statement obtained is submitted to the Visual Basic SQL processor for execution.

Thus the system implemented basically consists of three components:

1. derived relations and query evaluation methods – Naive (1.a.), Semi-Naive (1.b.) and Henschen-Naqvi (1.c.), with cost functions evaluated;
2. interface language compiler and translator;
3. procedure for deciding which method is to be used for materialization of a specific derived relation.

Cost functions for the methods implemented are based on the number of successful inferences, number of input/output (I/O) pages from permanent storage, and the number of Visual Basic operations. The I/O cost is considered as the principal criterion, the other two being corrective criteria.

Input/output (I/O) cost function first takes into account the number of I/O operations performed in a single join operation.

Let us consider a derived relation definition

$$R :- R_1, R_2, \ldots, R_n.$$

Evaluation I/O cost for the relation R is the sum of I/O costs of sequence of joins of the relations $R_1, \ldots, R_n$.

By analyzing the process of translating the derived relation R into an SQL statement, the overall I/O cost for evaluation of the SQL statement corresponding to the derived relation R is given by the expression:

$$\Sigma_{i=1}^{n} number\_of\_records(R_i) * sR_i/sp,$$

where sR_i and sp are sizes in bytes of records of relations R_i and a disk page, respectively.

Different methods will produce different I/O costs since they generate intermediate tables of different sizes.

Example Let EDB and IDB consist of the following relations:

parts(code, name) (base table)
structure(code1, code2) (base table)
$samelevel(X, X) : -parts(X, Y).$
$samelevel(X, Y) : -structure(P, X), samelevel(P, Q), structure(Q, Y).$

(code is an identifier of a part and name is its name, while code1 is an identifier of a part directly containing another part whose identifier is code2).

Figures 1–5 present system's processing of the derived relation *samelevel* and a query over it.

Queries against base relations alone are evaluated without any materialization (figure 8).

5 Syntax characterization of some classes of derived relations

By analyzing implementation of methods for derived relations evaluation, conclusions may be drawn about the most efficient methods for some specific classes of derived relations, from relations' syntax alone. It means that, for those classes of derived relations, optimization may be achieved at compile-time, without having to materialize relations by all the methods implemented (in order to estimate the best one).

Let us consider derived relations p defined by the following two clauses:

$p : -r_1, r_2, r_3$
$p : -p_1, p, p_2.$

Relations r_1, r_2, r_3, p_1, p_2 may be derived, and classification of relations p is based on the number of joins of the relation p with p_1 and p_2.

1. First class: non-recursive derived relations (the second clause is absent). Figure 7 presents an example of such a derived relation –

   $names(X, Y) : -parts(P, X), structure(P, Q), parts(Q, Y).$

2. Second class: *simply recursive* derived relations, i.e., relations p whose definition does not contain one of the relations p_1, p_2. This class includes relations defining transitive closure of other relations. Figure 6 presents an example of a derived relation of this class:

   $contains(X, Y) : -structure(X, Y).$
   $contains(X, Y) : -structure(X, Z), contains(Z, Y).$

3. Third class: derived relations p whose definition involves only one join operation (with p_1 or p_2) in the second clause.

4. Fourth class: derived relations p defined by both joins with p_1 and p_2 in the second clause, providing p_1 and p_2 are different.

The most efficient method for the first class of derived relations is Naive method. Namely, in case of such relations, materialization is done by initialization step which is identically implemented in all the methods. This step is followed, in Semi-Naive and Henschen-Naqvi methods, by further conditional steps.

The most efficient method for classes 2–4 is Semi-Naive method. Naive method is eliminated because it has the highest I/O cost. By analyzing I/O costs of intermediate results of iterations we prove that Semi-Naive method has better performance over Henschen-Naqvi.

6 Conclusion

In this paper we presented a programming system for deductive database development with optimization. Deductive database is developed by interfacing a relational database with a logical (Prolog-like) language. A query is evaluated by first materializing all the derived relations participating in it, then by translating it into SQL and finally by submitting the SQL query obtained to an SQL processor for execution. Optimization component consists in choosing one of the three implemented derived relation evaluation methods – Naive, Semi-Naive and Henschen-Naqvi, based on efficiency estimation for specific forms of derived relations.

A classification scheme for a broad class of derived relations is proposed, allowing for the choice of the most efficient method to be made on the basis of relation syntax alone. For all other derived relations, estimation of the most efficient method is done by materializing it using all the three implemented methods. This excessive amount of work is justified by the realistic assumption that relations (base and derived) represent a static part of the system which is rarely updated, while queries represent a dynamic part of the system, allowing for many queries to be formulated over the same set of relations.

Results obtained for examples from various areas of application are in favor of the chosen approach to optimization in deductive databases.

References

[1] Bancilhon,F, Ramakrishnan,R. "An Amateur's Introduction to Recursive Query Processing Strategies", in M.Stonebraker (ed.), *Readings in Database Systems*, Morgan Kauffman, 1988.

[2] Date,C.J. "A Note on the Parts Explosion Problem", in Date,C.J. (ed.) *Relational Database Writings 1985-1989*, Addison Wesley Publ. Inc., 1990.

[3] Minker,J. *Foundations of Deductive Databases and Logic Programming*, Morgan Kauffman, 1988.

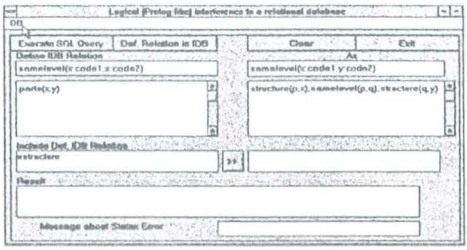

Figure 1

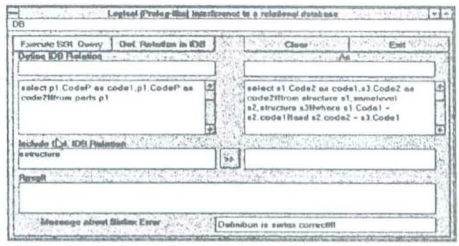

Figure 2

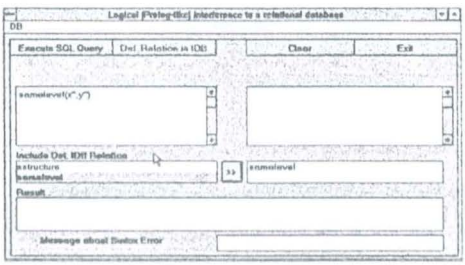

Figure 3

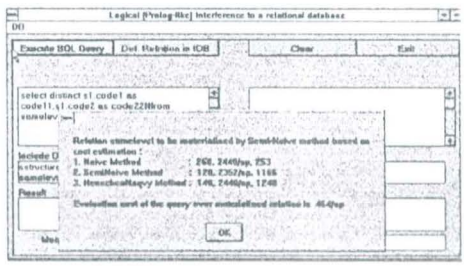

Figure 4

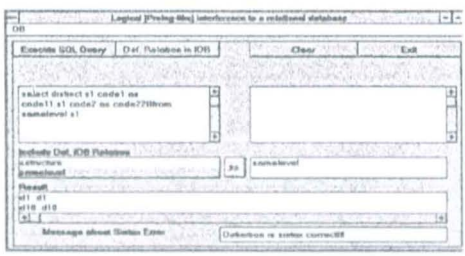

Figure 5

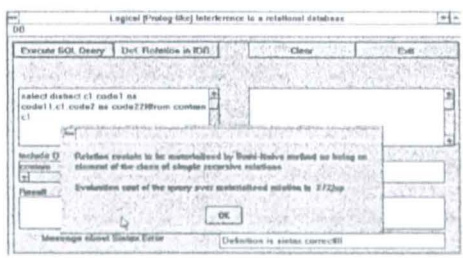

Figure 6

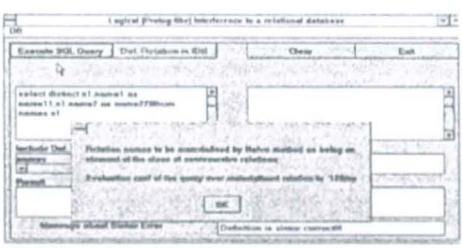

Figure 7

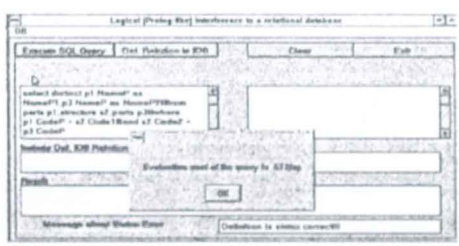

Figure 8

Proceedings of the VIII Conference on
Logic and Computer Science *LIRA '97*
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 67–75

An algorithm for localization of polynomial zeros

Djordje Herceg

Institute of Mathematics, University of Novi Sad,
Trg Dositeja Obradovića 4, Novi Sad 21000, Yugoslavia

Abstract. An algorithm for localization of polynomial zeros is given, which is based on a mesh-like division of a complex plane. The complex plane is divided recursively in order to narrow the regions which contain zeros. After only a few iterations which require modest computation time, a good starting approximation for an iterative zero-finding algorithm can be obtained. The programs are written in *Mathematica*.

1 Introduction

In order to find zeros of the polynomial

$$P(z) = p_0 + p_1 z + p_2 z^2 + \cdots + p_n z^n. \tag{1}$$

using an iterative method, starting approximations to these zeros must be found. There are many methods which can be used to find a region containing all the zeros of a polynomial, as well as methods which give regions containing only one zero or a cluster of zeros. In this paper a method for localization of polynomial zeros is given, which is based on Theorems 1 and 3 from [1]. The program is written in *Mathematica*.

2 Algorithm

Theorem 1 gives a disk which contains all the roots of the polynomial P, which we use as a starting region for the localization of zeros.

Theorem 1. *Let $\lambda_1, \ldots, \lambda_n$ be positive numbers such that*

$$\lambda_1 + \cdots + \lambda_n < 1 \ [\leq 1]$$

and let

$$R := \max_{1 \leq k \leq N} \lambda_k^{-1/k} \left| \frac{a_{N-k}}{a_N} \right|^{1/k}.$$

Then R is an inclusion radius for P.

In particular, by taking $\lambda_k = 1/2^k$, it follows from Theorem 1 that the disk centered at the origin with the radius

$$R = 2 \max_{1 \le k \le N} \left| \frac{a_{N-k}}{a_N} \right|^{1/k} \tag{2}$$

contains all the zeros of the polynomial P.

Definition 2. Let S_i be a disk with center z_i and radius r_i. The interval extension $\mathcal{P}_T(S_i)$ of P over S_i, called the Taylor circular centered form, is given by the disk with center

$$\text{mid}(\mathcal{P}_T(S_i)) = P(z_i)$$

and radius

$$\text{rad}(\mathcal{P}_T(S_i)) = \sum_{k=1}^{n} \frac{|p^{(k)}(z_i)|\, r_i^k}{k!}.$$

Here, $p^{(k)}(z_i)$ is the k–th derivative of $P(z)$ evaluated at point z_i. Using the above exclusion test ($0 \in \mathcal{P}(S_i)$) and Taylor's extension, Gargantini has established the following concrete exclusion test:

Theorem 3. *Let z_i and r_i denote the center and the radius of a disk S_i. A disk S_i is free of zeros if*

$$|P(z_i)| > \sum_{k=1}^{N} \frac{|P^{(k)}(z_i)|\, r_i^k}{k!}.$$

Theorem 3 makes it possible to test an arbitrary disk in the complex plane for presence of polynomial zeros. If the test evaluates to true, then the disk contains no zeros. If the test evaluates to false, then the disk may contain one or more zeros of P.

If a region in the complex plane is covered with a rectangular mesh, and each cell is covered with a disk, then we can use this test to tell which cells do not contain any zeros, and which may contain zeros of P.

The cells for which the test evaluates to true are discarded and never tested again. The cells in which the test evaluates to false are divided in four smaller cells.

The algorithm has three sections.

1. Find a starting region which contains all zeros of the polynomial. Cover the region with a rectangular mesh.
2. Mark the cells of the mesh which do not contain any zeros.
3. Discard marked cells. Cover unmarked cells (which may contain zeros) with a finer mesh. Use this as a starting cover and repeat step 2.

Theorem 1 is used to find a starting region which contains all zeros of P.

2.1 The Algorithm

We shall consider rectangles in the complex plane, which have edges parallel to the axes, and length of each edge kh, where $k \in \mathbf{N}$ and $h \in \mathbf{R}$. The number h is called *step*.

Each pair of complex numbers (z_0, z_1) which satisfies

$$\operatorname{Re} z_0 < \operatorname{Re} z_1, \quad \operatorname{Im} z_0 < \operatorname{Im} z_1, \tag{3}$$

determines a unique rectangle and vice versa, each rectangle determines such a pair. If we write

$$x_0 = \operatorname{Re} z_0, \quad x_1 = \operatorname{Re} z_1, \quad y_0 = \operatorname{Im} z_0, \quad y_1 = \operatorname{Im} z_1.$$

Then it holds

$$z_0 = x_0 + Iy_0 \quad \text{and} \quad z_1 = x_1 + Iy_1, \quad I = \sqrt{-1}.$$

Let

$$n_x = \frac{x_1 - x_0}{h}, \quad n_y = \frac{y_1 - y_0}{h},$$

and

$$\begin{aligned} q_{ij}^0 &= x_0 + (i-1)\,h + I\,(y_0 + (j-1)\,h)\,, \\ q_{ij}^1 &= x_0 + ih + I\,(y_0 + jh)\,, \end{aligned} \qquad i = 1, 2, \ldots, n_y, j = 1, 2, \ldots, n_x$$

It is obvious that n_x, $n_y \in \mathbf{N}$. For each $i = 1, 2, \ldots, n_y$, $j = 1, 2, \ldots, n_x$ a pair of complex numbers (q_{ij}^0, q_{ij}^1) defines a square as shown in Figure 1. We shall mark these squares with k_{ij}. In this way *a mesh* is formed which consists of $n_y \times n_x$ squares.

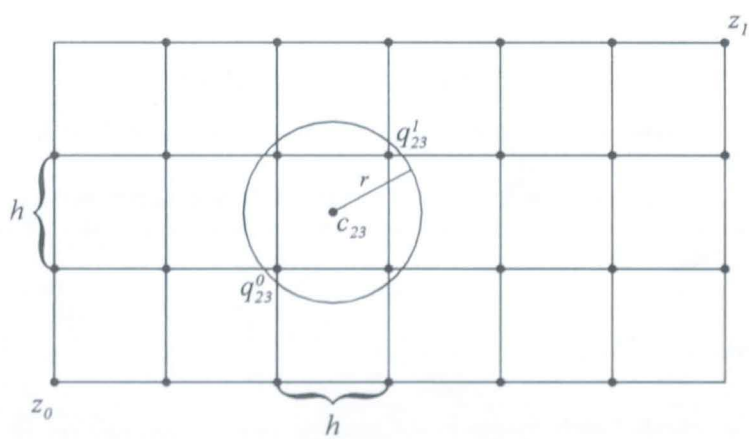

Figure 1.

A $n_y \times n_x$ matrix

$$M = [m_{ij}],$$

is formed, where the element m_{ij} of the matrix represents the square $k_{n_y-i+1,j}$ of the mesh. Elements of M can take values "valid" and "invalid".

If a square does not contain a zero of P, it is marked invalid. Otherwise it is marked valid. If the square is invalid, the corresponding element of M is set to "invalid". If the square is valid, the corresponding element of M is set to "valid". We used the disk from Theorem 3 with center

$$c_{ij} = q_{ij}^0 + \frac{h}{2}(1+I)$$

and radius

$$r = \frac{h}{1.375}$$

to check whether the square k_{ij}, $i = 1, 2, \ldots, n_y$, $j = 1, 2, \ldots, n_x$ contains zeros of P. We choose this radius instead of

$$r = \frac{h}{\sqrt{2}}$$

in order to reduce computation error, since 1.375 is a machine number, and $\sqrt{2}$ is not, and it holds $\sqrt{2} > 1.375$.

For a chosen rectangle, step h and a polynomial P with corresponding values of z_0, z_1, n_y, n_x and M, we form

$$\Omega = (z_0, z_1, h, n_y, n_x, P, M),$$

which we shall call *a region*.

The basic idea of a program for automatic determination of polynomial zeros in the complex plane is presented below.

1. Calculate inclusion radius R for given P, as given in 2.
2. Calculate z_0, z_1 as

   $$z_0 = -R(1+I), \quad z_1 = R(1+I),$$

   which determine smallest rectangle which contains the disk with center at the origin and radius R.
3. Set $n_x = n_y = 16$, which implies the matrix M is a square matrix with 16 rows and columns. The rectangle is covered with a mesh of 16×16 squares. Calculate

   $$h = \frac{x_1 - x_0}{n_x},$$

   where

   $$x_0 = -R, \quad x_1 = R.$$

4. Check whether each square in the mesh is valid or not, and set the corresponding elements of M to "valid" or "invalid" respectively. Now we have the starting region Ω.

5. From the starting region

$$\Omega = (z_0, z_1, h, n_y, n_x, P, M)$$

calculate a new region

$$\Omega^* = \left(z_0, z_1, h^*, n_y^*, n_x^*, P, M^*\right),$$

where

$$h^* = \frac{h}{2}, \quad n_y^* = 2n_y, \quad n_x^* = 2n_x,$$

and the elements of the matrix

$$M^* = \left[m_{ij}^*\right],$$

of dimension $n_y^* \times n_x^*$ are determined as follows.

- if the value of the element m_{ij} is "invalid" then the elements

$$\begin{matrix} m_{2i-1,2j-1}^* & m_{2i-1,2j}^* \\ m_{2i,2j-1}^* & m_{2i,2j}^* \end{matrix}$$

  are set to "invalid".
- if not, values of these elements are calculated in the test from Theorem 3, with appropriate squares

$$\begin{matrix} k_{n_y^*-2i,2j-1}^* & k_{n_y^*-2i,2j}^* \\ k_{n_y^*-2i+1,2j-1}^* & k_{n_y^*-2i+1,2j}^* \end{matrix}.$$

6. Now find all the disjoint *groups* of valid squares in the region Ω^*. Assign a *minimal rectangle* to each group, that is, the smallest rectangle that contains the group and has edges parallel to the axes. Mesh squares contained in the minimal rectangle then form a new region. The corresponding matrix of that new region is the appropriate submatrix of the matrix M^* from the region Ω^*.
7. Each new region from the previous step is formed by taking a submatrix of M^*, which corresponds to the minimal rectangle containing that region, that is, it corresponds to the subset of squares from Ω^*. Other parameters of the new region are calculated accordingly. The regions obtained in this way are now called *starting regions*, and the algorithm is then applied to each of them, from step 5.
8. The algorithm terminates when all the starting regions are smaller than some previously given ε.

Disjoint groups of valid squares, which were mentioned in step 6, appear because the test on Ω^* is more subtle than the test on Ω, because value of step h^* is half the value of h.

Usually after only a couple of iterations we get exactly the same number of disjoint groups as there are different zeros of the polynomial, and one starting region is quickly divided in a few smaller ones, which saves a lot of computer time and memory needed for the matrix M.

The term *group of squares*, which was used here, is defined as follows.

Definition 4. Two mesh squares x and y are said to be *neighbors*, which is written as $x \diamond y$, if and only if they have one common edge.

Definition 5. Two squares x and y are said to be *connected* if they are neighbors or if there exists an array of squares $a_1, a_2, \ldots, a_k$ $k \geq 1$, such that

$$x \diamond a_1 \diamond a_2 \diamond \cdots \diamond a_k \diamond y.$$

Definition 6. A set of squares is a *group of squares* if and only if every two squares from that set are connected.

2.2 Program

A region Ω is represented by a new data type in *Mathematica*:

```
Region[{z0 ,z1 ,h ,ny ,nx ,p ,x}, M]
```

Element x is the variable of P.
The most important functions in the program are:

- `Size[b]` calculates size of the longer side of the region `b`;
- `FindZeros[p, x, eps]` main function for finding polynomial zeros. The iterative process is terminated when size of all the regions is smaller than `eps`.
- `Radius[p, x]` is inclusive radius for P.
- `Cover[p, {x, z0, z1}, opts]` covers the region given by `z0` i `z1` with a mesh of 16×16 squares, and tests the polynomial $P(x)$ on these squares. This funkction is used only once in the main function to generate the starting region, but it can be used to obtain a region with arbitrary z_0, z_1 and number of mesh squares. The parameter `opts` allows the number of mesh squares to be set with the option `MeshPoints`. Default value is 16. The region obtained through this function can be covered with a mesh twice as dense by using the function `Refine[b]`;
- `SplitGroups[b]` extracts all the disjoint groups of valid squares from `b` as a list of regions.

3 Implementation

```
Size[b_$\QTR{tt}{Region}$]:=
  With[{x0=Re[b[[1,1]]],y0=Im[b[[1,1]]],x1=Re[b[[1,2]]],
        y1=Im[b[[1,2]]]},
    Max[x1-x0,y1-y0]]

FindZeros[p_, x_, eps_] :=
  Module[{W, k, R, l, L, H, h},
    R = Radius[p, x];
    W = {Cover[p, {x, -R*(1 + I), R*(1 + I)}]};
```

```
    L = {};
    While[W != {},
      W = Flatten[(SplitGroups[Refine[#1]] & ) /@ W];
      l = Flatten[Position[(Size[#1] <= eps)& /@ W, True]];
      L = Join[L, W[[l]]];
      W = W[[Complement[Range[Length[W]], l]]];
    ];
    L
  ]

Format[$\QTR{tt}{Region}$[t_,b_] ]:=''-Oblast-''

Radius[p_, x_] :=
  Module[{n = Exponent[p, x], c},
    c = Table[Coefficient[p, x, k], {k, 0, n}];
    2 Max[Table[Abs[c[[n - k + 1]]/c[[n + 1]]]^(1/k), {k, n}]]
  ]

MeshPoints::usage=
  "MeshPoints is an option for Cover which specifies how many
  mesh points to use.";

Cover[p_, {x_, z0x_, z1x_}, opts___] :=
  Module[{z0 = z0x, z1 = z1x, dx, dy, rad, h, n = Exponent[p, x],
          test, z, k, e, pts, nx, ny, dp, suma, a, b},
    pts = If[opts === Null, 16, (If[NumberQ[#1], #1, 16] & )
            [MeshPoints /. opts]];
    dx = Re[z1 - z0]; dy = Im[z1 - z0];
    If[dx > dy, h = N[dx/pts]; nx = pts; ny = Ceiling[dy/h];
                {z0, z1} += I*(ny*h - dy)/2,
                h = N[dy/pts]; ny = pts; nx = Ceiling[dx/h];
                {z0, z1} += (nx*h - dx)/2];
    rad = h/1.375;
    dp = Table[Simplify[D[p, {x, k}]], {k, n}];
    suma = Simplify[Sum[(Abs[dp[[k]]]*rad^k)/k!, {k, 1, n}]];
    test[z_] := If[Abs[p /. x -> z] > suma /. x -> z, "#", 0];
    $\QTR{tt}{Region}$[{z0, z1, h, ny, nx, p, x},
      Table[test[z0 + i*h + h/2 + I*(j*h + h/2)], {j, 0, ny - 1},
        {i, 0, nx - 1}]]
  ]

Refine[b_$\QTR{tt}{Region}$] :=
  Module[{c, Dbl, z, i, j, k, x0 = Re[b[[1,1]]], y0 = Im[b[[1,1]]],
    x1 = Re[b[[1,2]]], y1 = Im[b[[1,2]]], h = b[[1,3]]/2, ny = 2*b[[1,4]],
    nx = 2*b[[1,5]], p = b[[1,6]], x = b[[1,7]], test, dp, n, rad, h2},
    n = Exponent[p, x];
    Dbl[h_] := Flatten[({#1, #1} & ) /@ h, 1];
    c = Dbl[Dbl /@ b[[2]]];
    rad = h/1.375;
    dp = Table[Simplify[D[p, {x, k}]], {k, n}];
```

```
    suma = Simplify[Sum[(Abs[dp[[k]]]*rad^k)/k!, {k, 1, n}]];
    test[z_] := If[Abs[p /. x -> z] > suma /. x -> z, "#", 0];
    $\QTR{tt}{Region}$[{b[[1,1]], b[[1,2]], h, ny, nx, p, x},
      Table[If[c[[j,i]] == "#", "#",
        test[x0 + (i - 1)*h + h/2 + I*(y0 + (j - 1)*h + h/2)]],
        {j, ny}, {i, nx}]]
  ]

Neighbor[{x0_, y0_}, {x1_, y1_}] :=
  True /; y0 == y1 && (x0 == x1 + 1 || x0 + 1 == x1) ||
          x0 == x1 && (y0 == y1 + 1 || y0 + 1 == y1);
  Neighbor[{_, _}, {_, _}] := False;

CountGroups[b0_Oblast]:=
  Module[{b=b0[[2]],p,h,nov,br=0},
    p=Position[b,0];
    While[p != {},
      h={First[p]};
      br++;
      nov={};
      While[h\[NotEqual]{},
        h=Join[h,Select[p,Neighbor[First[h],#]&]];
        p=Complement[p,h];
        nov=Append[nov,First[h]];
        h=Rest[h];
      ];
      b=ReplacePart[b,br,nov];
    ];
    {br,$\QTR{tt}{Region}$[b0[[1]],b]}
  ]

ExtractGroup[b0_$\QTR{tt}{Region}$, n_] :=
  Module[{x0, y0, x1, y1, z0, z1, h, b = b0[[2]], p},
    {z0, z1, h} = Take[b0[[1]], {1, 3}]; p = Position[b, n];
    If[p != {}, p = Transpose[p];
      {x0, y0, x1, y1} = {Min[p[[2]]], Min[p[[1]]], Max[p[[2]]],
      Max[p[[1]]]};
      $\QTR{tt}{Region}$[Join[{z0 + (x0 - 1)*h + I*(y0 - 1)*h,
        z0 + x1*h + I*y1*h, h, y1 - y0 + 1, x1 - x0 + 1},
        Take[b0[[1]], {6, 7}]],
        Take[(Take[#1, {x0, x1}] & ) /@ b, {y0, y1}]],
    Null
  ]

SplitGroups[b0_$\QTR{tt}{Region}$]:=
  Module[{n,l},
    {n,l}=CountGroups[b0];
    Table[ExtractGroup[l,i],{i,n}]
  ]
```

4 Example

We have tested the program on a polynomial of degree 7, which has all roots of degree 1. In Figure 2 the inclusion disk is shown, and only one group of valid squares is found. However, when this region is covered with a mesh twice as dense, seven disjoint groups of squares show up, which give us approximate positions of the roots.

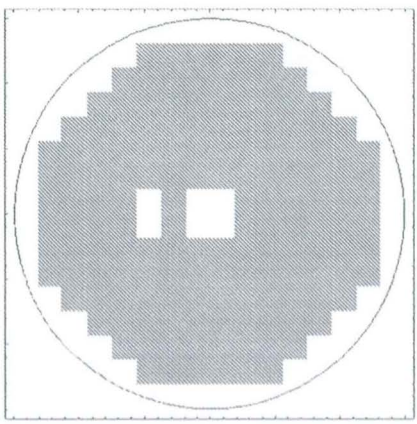

Figure 2.

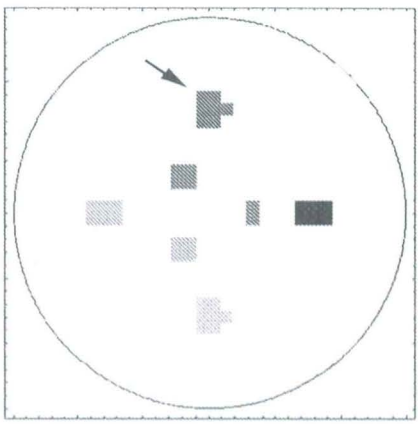

Figure 3.

The region marked with an arrow in Figure 3 is divided further in figures 4–6. The original is shown in Figure 4. When refined, it "shrinks" to the region painted darker in Figure 5. This region is taken as a starting region for the next iteration (Figure 6).

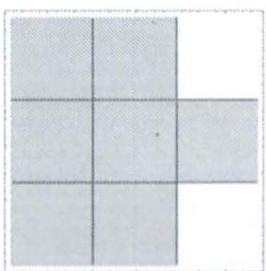

Figure 4.

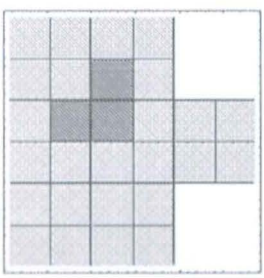

Figure 5.

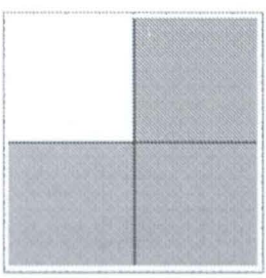

Figure 6.

References

1. M. Petković, Iterative methods for Simultaneous Inclusion of Polynomial Zeros, Springer-Verlag, 1989.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 77–84

On ν_i-products of tree automata

Balázs Imreh

Dept. of Informatics, József Attila University
Árpád tér 2, H-6720 Szeged, Hungary
e-mail: imreh@inf.u-szeged.hu

Abstract. In the present work, the isomorphically complete systems of tree automata with respect to the ν_i-products and the star-product are characterized. The descriptions shows that the ν_i-product is equivalent to the star-product regarding the isomorphically complete systems of tree automata, for all positive integer i. Moreover, it is proved that the ν_i-products constitute a proper hierarchy for the tree automata.

1 Introduction

In general, a composition of automata can be considered as a network of automata where the input sign of each automaton of this network depends on the input sign of the network and the actual states of the network neighbours of the automaton considered. If the number of the neighbours is bounded by a positive integer i, then we obtain the notion of the ν_i-product. This notion is introduced in [1] where the isomorphic representation and simulation regarding the ν_i-product are studied. Further studies on the ν_i-products can be found in the works [2], [3], [4], [6], [7], [9], [11], [13]. Here, we generalize the notion of the ν_i- product to tree automata and characterize the isomorphically complete systems of tree automata with respect to the ν_i-product, for all i, $i = 1, 2, \ldots$. Furthermore, it is shown that the hierarchy of the ν_i-products is proper. Another kind of compositions is the star-product (see [5], [10], [14], [15]) where the network contains a central automaton and each further automaton has the central automaton as its neighbour and conversely. We generalize this notion to tree automata and characterize the isomorphically complete systems of tree automata with respect to the star-product. This characterization shows that the star-product is equivalent to the ν_i-product with respect to the isomorphically complete systems of tree automata, for all positive integer i.

This work has been supported by the Hungarian National Foundation for Scientific Research, Grant 014888, and by the Ministry of Culture and Education of Hungary, Grant FKFP 0704/1997.

2 Preliminaries

First of all, we recall some notions from [12]. By a set of *operational symbols*, we mean a nonempty union $\Sigma = \Sigma_0 \bigcup \Sigma_1 \bigcup \ldots$, where Σ_m, $m = 0, 1, \ldots$, are pairwise disjoint sets of symbols. For any $m \geq 0$, the set Σ_m is called the set of *m-ary operational symbols*. It is said that the *rank* or *arity* of a symbol $\sigma \in \Sigma$ is m if $\sigma \in \Sigma_m$. Now, let a set Σ of operational symbols and a set R of nonnegative integers be given. R is called the *rank-type* of Σ if $\Sigma_m \neq \emptyset$ if and only if $m \in R$, for every integer $m \geq 0$. Let Σ be a set of operational symbols with rank-type R. By a *Σ-algebra* $\mathbf{A}$, we mean a pair consisting of a nonempty set A and a mapping that assigns to every operational symbol $\sigma \in \Sigma$ an m-ary operation $\sigma^{\mathbf{A}} : A^m \to A$ where the arity of σ is m. The set A is called the *set of elements of* $\mathbf{A}$ and $\sigma^{\mathbf{A}}$ is the *realization of* σ *in* $\mathbf{A}$. The mapping $\sigma \to \sigma^{\mathbf{A}}$ will not be mentioned explicitly, but we write $\mathbf{A} = (A, \Sigma)$. It is said that a Σ-algebra $\mathbf{A}$ is *finite* if A is finite, and it is of *finite type* if Σ is finite. By *a tree automaton*, we mean a finite algebra of finite type. Finally, it is said that the *rank-type of a tree automaton* $\mathbf{A} = (A, \Sigma)$ is R if the rank-type of Σ is R.

Now, let us denote the class of all tree automata with rank-type R by $\mathcal{U}_R$. In what follows, we need a sequence of special tree automata in $\mathcal{U}_R$. For this purpose, let k be an arbitrary positive integer. For every $m \in R$, let us assign one and only one symbol to each m-ary operation over $\{1, \ldots, k\}$. Let $\Theta_m^{(k)}$ denote the set of these symbols and let $\Theta^{(k)} = \cup\{\Theta_m^{(k)} : m \in R\}$. Define the tree automaton $\mathbf{B}_k = (\{1, \ldots, k\}, \Theta^{(k)})$ such that, for every $m \in R$ and $\sigma \in \Theta_m^{(k)}$, $\sigma^{\mathbf{B}_k}$ is the m-ary operation assigned to σ above.

3 ν_i-products

In this section, the ν_i-products of tree automata are studied. In order to define the ν_i-product, let i be an arbitrarily fixed positive integer. Let $\mathbf{A} = (A, \Sigma) \in \mathcal{U}_R$ and $\mathbf{A}_j = (A_j, \Sigma^{(j)}) \in \mathcal{U}_R$, $j = 1, \ldots, n$. Let the neighbours be given by the function $\gamma : \{1, \ldots, n\} \to P(\{1, \ldots, n\})$ satisfying $|\gamma(j)| \leq i$, for all j, $j = 1, \ldots, n$, where $P(\{1, \ldots, n\})$ denotes the power-set of $\{1, \ldots, n\}$. Furthermore, take a family φ of mappings defined by

$$\varphi_{0j} : \Sigma_0 \to \Sigma_0^{(j)} \text{ provided that } 0 \in R \text{ , and}$$
$$\varphi_{mj} : (A_1 \times \ldots \times A_n)^m \times \Sigma_m \to \Sigma_m^{(j)}, \quad 0 \neq m \in R, \quad 1 \leq j \leq n.$$

It is said that the tree automaton $\mathbf{A}$ is a *ν_i-product* of $\mathbf{A}_j$, $j = 1, \ldots, n$, with respect to φ and γ if the following conditions are satisfied:

(i) $$A = \prod_{j=1}^{n} A_j,$$

(ii) if $0 \in R$, then for every $\sigma \in \Sigma_0$, $\sigma^{\mathbf{A}} = (\sigma_1^{\mathbf{A}_1}, \ldots, \sigma_n^{\mathbf{A}_n})$ is valid where $\sigma_j = \varphi_{0j}(\sigma)$, for all j, $j = 1, \ldots, n$,

(iii) for every $0 \neq m \in R$, $j \in \{1, \ldots, n\}$ and

$$((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn})) \in A^m,$$

φ_{mj} is independent of the elements a_{ts}, $t = 1, \ldots, m$, if $s \notin \gamma(j)$,

(iv) for every $m \in R$, $\sigma \in \Sigma_m$, $((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn})) \in A^m$,

$$\sigma^{\mathbf{A}}((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn})) =$$
$$(\sigma_1^{\mathbf{A}_1}(a_{11}, \ldots, a_{m1}), \ldots, \sigma_n^{\mathbf{A}_n}(a_{1n}, \ldots, a_{mn}))$$

where $\sigma_j = \varphi_{mj}((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn}), \sigma)$, $j = 1, \ldots, n$.

We shall use the notation $\prod_{j=1}^{n} \mathbf{A}_j(\Sigma, \varphi, \gamma)$ for the product introduced above, and sometimes we shall indicate only those variables of φ_{mj} on which it may depend.

Let $\mathcal{B}$ be a system of tree automata from $\mathcal{U}_R$. It is said that $\mathcal{B}$ is *isomorphically complete for $\mathcal{U}_R$ with respect to the ν_i-product* if any tree automaton from $\mathcal{U}_R$ can be embedded isomorphically into a ν_i-product of tree automata from $\mathcal{B}$.

Now, we are ready to characterize the isomorphically complete systems of tree automata with respect to the ν_i-product. The following statement is obvious if $R = \{0\}$.

Theorem 1. *A system $\mathcal{B}(\subseteq \mathcal{U}_{\{0\}})$ of tree automata is isomorphically complete for $\mathcal{U}_{\{0\}}$ with respect to the ν_i-product if and only if there exists an $\mathbf{A} \in \mathcal{B}$ such that $\mathbf{B}_2$ can be embedded isomorphically into a ν_i-product of $\mathbf{A}$ with a single factor.*

Let us suppose that $R \neq \{0\}$. Then, the following statement is valid.

Theorem 2. *A system $\mathcal{B}(\subseteq \mathcal{U}_R)$ of tree automata is isomorphically complete for $\mathcal{U}_R$ with respect to the ν_i-product if and only if, for any positive integer k, there exists an $\mathbf{A} \in \mathcal{B}$ such that $\mathbf{B}_k$ can be embedded isomorphically into a ν_i-product of $\mathbf{A}$ with a single factor.*

Proof. The sufficiency of the condition follows from the fact that any tree automaton of k states from $\mathcal{U}_R$ can be embedded isomorphically into a ν_i-product of $\mathbf{B}_k$ with a single factor.

If $k = 1$, then the necessity is obvious. Now, let $k \geq 2$ be an arbitrary integer, and let us denote k^{i+1} by w. First, we prove the following statement. If $\mathbf{B}_w$ can be embedded isomorphically into a ν_i-product $\mathbf{A} = \prod_{j=1}^{n} \mathbf{A}_j(\Theta^{(w)}, \varphi, \gamma)$ of tree automata $\mathbf{A}_j \in \mathcal{U}_R$, $j = 1, \ldots, n$, then $\mathbf{B}_k$ can be embedded isomorphically into a ν_i-product of $\mathbf{A}_j$ with a single factor for some $j \in \{1, \ldots, n\}$. Let μ denote an isomorphism of $\mathbf{B}_w$ into $\mathbf{A}$, and let us denote by $(a_{t1}, \ldots, a_{tn})$ the image of t under μ, for all t, $t = 1, \ldots, w$. Without loss of generality, it can be assumed that $a_{r1} \neq a_{s1}$ for some $r \neq s \in \{1, \ldots, w\}$. Indeed, in the opposite case, $\mathbf{B}_w$ can be embedded isomorphically into a ν_i-product of $\mathbf{A}_2, \ldots, \mathbf{A}_n$. By the definition of the ν_i-product, its ordering is rearrangable. Therefore, we may assume that $\gamma(1) \cup \{1\} = \{1, \ldots, l\}$ where $l \leq i + 1$. Now, we prove that the elements $(a_{t1}, \ldots, a_{tl})$, $t = 1, \ldots, w$, are pairwise different. Contrary, let us suppose that there are $p \neq q \in \{1, \ldots, w\}$ such that $a_{pv} = a_{qv}$, $v = 1, \ldots, l$. Let $m \neq 0$ be an arbitrarily fixed

element of R. For every $t \in \{1, \ldots, w\}$, let us denote by σ_{pt} an m-ary operational symbol of $\mathbf{B}_w$ satisfying $\sigma_{pt}^{\mathbf{B}_w}(1, \ldots, 1, p) = t$ and $\sigma_{pt}^{\mathbf{B}_w}(1, \ldots, 1, q) = q$. Since $R \neq \{0\}$ such an operational symbol exists. Then, for every $t \in \{1, \ldots, w\}$,

$$(a_{t1}, \ldots, a_{tn}) = \mu(t) = \mu(\sigma_{pt}^{\mathbf{B}_w}(1, \ldots, 1, p)) = \sigma_{pt}^{\mathbf{A}}(\mu(1), \ldots, \mu(1), \mu(p)) =$$
$$(\sigma_1^{\mathbf{A}_1}(a_{11}, \ldots, a_{11}, a_{p1}), \ldots, \sigma_n^{\mathbf{A}_n}(a_{1n}, \ldots, a_{1n}, a_{pn}))$$

is valid, and thus, $a_{t1} = \sigma_1^{\mathbf{A}_1}(a_{11}, \ldots, a_{11}, a_{p1})$ where

$\sigma_1 = \varphi_{m1}((a_{11}, \ldots, a_{1n}), \ldots, (a_{11}, \ldots, a_{1n}), (a_{p1}, \ldots, a_{pn}), \sigma_{pt}) =$
$\varphi_{m1}(a_{11}, \ldots, a_{1l}, a_{p1}, \ldots, a_{pl}, \sigma_{pt})$.

In the same way as above, we get that $a_{q1} = \bar{\sigma}_1^{\mathbf{A}_1}(a_{11}, \ldots, a_{11}, a_{q1})$ where $\bar{\sigma}_1 = \varphi_{m1}(a_{11}, \ldots, a_{1l}, a_{q1}, \ldots, a_{ql}, \sigma_{pt})$. Then, by the equalities $a_{pv} = a_{qv}$, $v = 1, \ldots, l$, we obtain that $\sigma_1 = \bar{\sigma}_1$. This implies $a_{t1} = a_{q1}$, $t = 1, \ldots, w$, which contradicts the assumption $a_{r1} \neq a_{s1}$. Consequently, the elements $(a_{t1}, \ldots, a_{tl})$, $t = 1, \ldots, w$, are pairwise different.

Now, we prove that $\mathbf{B}_w$ can be embedded isomorphically into a ν_l-product of $\mathbf{A}_1, \ldots, \mathbf{A}_l$. Since $(a_{t1}, \ldots, a_{tl})$, $t = 1, \ldots, w$, are pairwise different, the mapping $\rho : (a_{t1}, \ldots, a_{tl}) \to (a_{t1}, \ldots, a_{tn})$, $t = 1, \ldots, w$, is a one-to-one mapping. Let us form the ν_l-product, $\prod_{j=1}^{l} \mathbf{A}_j(\Theta^{(w)}, \bar{\varphi}, \bar{\gamma})$ as follows. Let $\bar{\gamma}(r) = \{1, \ldots, l\}$, for all r, $r = 1, \ldots, l$. Let us denote the set $\{(a_{t1}, \ldots, a_{tl}) : t = 1, \ldots, w\}$ by C. Obviously, $C \subseteq \prod_{j=1}^{l} A_j$. Define the mappings $\bar{\varphi}_{mj}$, $m \in R$, $j \in \{1, \ldots, l\}$, in the following way. If $0 \in R$, then let $\bar{\varphi}_{0j}(\sigma) = \varphi_{0j}(\sigma)$, for all $j \in \{1, \ldots, l\}$ and $\sigma \in \Theta_0^{(w)}$. Furthermore, for every $0 \neq m \in R$, $j \in \{1, \ldots, l\}$, $\sigma \in \Theta_m^{(w)}$, $\mathbf{b}_t = (b_{t1}, \ldots, b_{tl}) \in \prod_{r=1}^{l} A_r$, $t = 1, \ldots, m$, let

$$\bar{\varphi}_{mj}((\mathbf{b}_1, \ldots, \mathbf{b}_m), \sigma) =$$

$$\begin{cases} \varphi_{mj}(\rho(\mathbf{b}_1), \ldots, \rho(\mathbf{b}_m), \sigma) & \text{if } \mathbf{b}_t \in C, \ t = 1, \ldots, m, \\ \text{a fixed operational symbol from } \Sigma_m^{(j)} & \text{otherwise.} \end{cases}$$

Obviously, $\bar{\varphi}_{mj}$ is well defined. Let us define the mapping β by $\beta(t) = (a_{t1}, \ldots, a_{tl})$, $t = 1, \ldots, w$. Then, it is easy to see that β is an isomorphism of $\mathbf{B}_w$ into $\prod_{j=1}^{l} \mathbf{A}_j(\Theta^{(w)}, \bar{\varphi}, \bar{\gamma})$.

As the next step, we show that $\mathbf{B}_k$ can be embedded isomorphically into a ν_i-product of $\mathbf{A}_j$ with a single factor for some $j \in \{1, \ldots, l\}$. Since $l \leq i+1$ and $w = k^{i+1}$, there exists a $j \in \{1, \ldots, l\}$ such that the number of the pairwise different elements among $a_{1j}, \ldots, a_{wj}$ is not less than k. For the sake of simplicity, let us suppose that $a_{1j}, \ldots, a_{kj}$ are pairwise different. Then, $\tau : a_{tj} \to (a_{t1}, \ldots, a_{tl})$ is a one-to-one mapping. For every $m \in R$, $\sigma \in \Theta_m^{(k)}$, let us denote by $\bar{\sigma}$ an operational symbol from $\Theta_m^{(w)}$ satisfying $\bar{\sigma}_{|\{1,\ldots,k\}^m}^{\mathbf{B}_w} = \sigma^{\mathbf{B}_k}$. Now, form the ν_i-product $\mathbf{A}_j(\Theta^{(k)}, \varphi^*, \gamma^*)$ with a single factor as follows. Let $\gamma^*(1) = \{1\}$. If $0 \in R$, then let $\varphi_0^*(\sigma) = \bar{\varphi}_{0j}(\bar{\sigma})$, for all $\sigma \in \Theta_0^{(k)}$. Furthermore, for any $0 \neq m \in R$, $\sigma \in \Theta_m^{(k)}$, $a_{r_1 j}, \ldots, a_{r_m j} \in A_j$, let

$$\varphi_m^*(a_{r_1 j}, \ldots, a_{r_m j}, \sigma) =$$

$$\begin{cases} \bar{\varphi}_{mj}(\tau(a_{r_1 j}), \ldots, \tau(a_{r_m j}), \bar{\sigma}) & \text{if } r_t \in \{1, \ldots, k\}, \ t = 1, \ldots, m, \\ \text{a fixed operational symbol from } \Sigma_m^{(j)} & \text{otherwise.} \end{cases}$$

By the construction, it can be easily proved that the mapping δ defined by $\delta(s) = a_{sj}$, $s = 1, \ldots, k$, is an isomorphism of $\mathbf{B}_k$ into $\mathbf{A}_j(\Theta^{(k)}, \varphi^*, \gamma^*)$.

Now, the necessity can be readily proved. Indeed, let $k \geq 2$ be an arbitrary integer. Let us consider the tree automaton $\mathbf{B}_w$ where $w = k^{i+1}$. Since $\mathcal{B}$ is isomorphically complete with respect to the ν_i-product, $\mathbf{B}_w$ can be embedded isomorphically into a ν_i-product $\mathbf{A} = \prod_{j=1}^{n} \mathbf{A}_j(\Theta^{(w)}, \varphi, \gamma)$ of tree automata from $\mathcal{B}$. Then, by the proof above, $\mathbf{B}_k$ can be embedded isomorphically into a ν_i-product of $\mathbf{A}_j$ with a single factor for some $j \in \{1, \ldots, n\}$ which completes the proof.

By the theorems above, the following corollary is obvious.

Corollary 1. For every rank-type R, the ν_i-product is equivalent to the ν_j-product with respect to the isomorphically complete systems of tree automata where i and j are arbitrary positive integers.

Remark. Let us observe that $\mathcal{U}_R$ is the class of the traditional automata if $R = \{1\}$. In this case, $\mathbf{B}_k$ denotes such an automaton which has k states and the input signs induce the all transformations over $\{1, \ldots, k\}$. Therefore, we obtain the characterization of the isomorphically complete systems of automata with respect to the ν_i- product presented in [1] as a special case of Theorem 2.

For studying the hierarchy of the ν_i-products, let us introduce the following notion. For every nonempty set $M \subseteq \mathcal{U}_R$ and positive integer k, let $\nu_k(M)$ denote the class of all tree automata from $\mathcal{U}_R$ which can be embedded isomorphically into a ν_k-product of tree automata from M. Now, let $i \neq j$ be arbitrary integers. It is said that the ν_i-product is *isomorphically more general* than the ν_j-product if $\nu_j(M) \subseteq \nu_i(M)$ is valid, for any set $\emptyset \neq M \subseteq \mathcal{U}_R$, furthermore, there exists at least one set $\emptyset \neq \bar{M} \subseteq \mathcal{U}_R$ such that $\nu_j(\bar{M}) \subset \nu_i(\bar{M})$. We note that this notion was originally introduced in [8] for the α_i-products.

The following statement presents that the hierarchy of the ν_i-products is proper if $R \neq \{0\}$.

Theorem 3. *For every pairs of positive integers u, v, the ν_v-product is isomorphically more general than the ν_u-product if $u < v$.*

Proof. Let $u < v$ be arbitrarily fixed integers. By the definition of the ν_i-products, it is obvious that $\nu_u(M) \subseteq \nu_v(M)$ is valid, for all nonempty set $M \subseteq \mathcal{U}_R$. Now, let $\bar{M} = \{\mathbf{B}_2\}$ and i be an arbitrary positive integer.

As a first step, the validity of $\mathbf{B}_{2^{i+1}} \in \nu_i(\bar{M})$ is proved. Let us denote 2^{i+1} by w, and let μ be a one-to-one mapping of $\{1, \ldots, w\}$ onto $\{1, 2\}^{i+1}$. It is shown that, for every $0 \neq m \in R$, $\sigma \in \Theta_m^{(w)}$, $\mathbf{a}_t = (a_{t1}, \ldots, a_{ti}) \in \{1, 2\}^i$, $t = 1, \ldots, m$, an m-ary operation σ' over $\{1, 2\}$ can be determined. For this purpose, let $x_t \in \{1, 2\}$, $t = 1, \ldots, m$, and let $\mathbf{a}'_t = (a_{t1}, \ldots, a_{ti}, x_t)$. Then,

$\mathbf{a}'_t \in \{1,2\}^{i+1}$, $t = 1,\ldots,m$, and thus, there is one and only one $i_t \in \{1,\ldots,w\}$ such that $\mu(i_t) = \mathbf{a}'_t$, for all t, $t = 1,\ldots,m$. Now, let $\sigma^{\mathbf{B}_w}(i_1,\ldots,i_m) = s$ and $\mu(s) = (b_1,\ldots,b_i,u)$. Let us define σ' by $\sigma'(x_1,\ldots,x_m) = u$. Obviously, σ' is an m-ary operation over $\{1,2\}$. We recall σ' as the m-ary *operation induced by the elements* $\sigma, \mathbf{a}_1,\ldots,\mathbf{a}_m$.

Now, let us form the ν_i-power $\mathbf{A} = \mathbf{B}_2^{i+1}(\Theta^{(w)},\varphi,\gamma)$ as follows. Let $\gamma(j) = \{1,\ldots,i+1\}\setminus\{j\}$, for all j, $j = 1,\ldots,i+1$. In this case, for every $0 \neq m \in R$, and $j \in \{1,\ldots,i+1\}$, the mapping φ_{mj} has the following form:

$$\varphi_{mj} : (\{1,2\}^i)^m \times \Theta_m^{(w)} \to \Theta_m^{(2)}.$$

Let us define the family φ of mappings in the following way. If $0 \in R$, then let us denote by σ_{i0} the operational symbol of $\Theta_0^{(2)}$ which has the realization $\sigma_{i0}^{\mathbf{B}_2} = i$ for $i = 1,2$. For every $\sigma \in \Theta_0^{(w)}$ and $j \in \{1,\ldots,i+1\}$, let $\varphi_{0j}(\sigma) = \sigma_{i0}$ where i is the j-th component of $\mu(\sigma^{\mathbf{B}_w})$. Now, let $0 \neq m \in R$, $\sigma \in \Theta_m^{(w)}$, $\mathbf{a}_t = (a_{t1},\ldots,a_{ti}) \in \{1,2\}^i$, $t = 1,\ldots,m$. Then, φ_{mj} is defined by

$$\varphi_{mj}(\mathbf{a}_1,\ldots,\mathbf{a}_m,\sigma) = \sigma^*$$

where the realization of σ^* is the m-ary operation induced by the elements σ, $\mathbf{a}_1,\ldots\mathbf{a}_m$. By the definition of $\mathbf{B}_2$, such an operational symbol exists.

Now, we prove that μ is an isomorphism of $\mathbf{B}^{(w)}$ onto $\mathbf{A}$. For this purpose, let us denote $\mu(s)$ by $(a_{s1},\ldots,a_{s,i+1})$, for all s, $s = 1,\ldots,w$. If $0 \in R$ and $\sigma \in \Theta_0^{(w)}$, then the equality $\mu(\sigma^{\mathbf{B}_w}) = \sigma^{\mathbf{A}}$ follows from the definition of the mappings φ_{0j}, $j = 1,\ldots,i+1$. Now, let $0 \neq m \in R$, $\sigma \in \Theta_m^{(w)}$, $k_1,\ldots,k_m \in \{1,\ldots,w\}$ be arbitrary elements. We have to prove that

$$\mu(\sigma^{\mathbf{B}_w}(k_1,\ldots,k_m)) = \sigma^{\mathbf{A}}(\mu(k_1),\ldots,\mu(k_m)).$$

Let $\sigma^{\mathbf{B}_w}(k_1,\ldots,k_m) = k$. By the definition of $\mathbf{A}$,

$$\sigma^{\mathbf{A}}(\mu(k_1),\ldots,\mu(k_m)) = (\sigma_1^{\mathbf{B}_2}(a_{k_1 1},\ldots,a_{k_m 1}),\ldots,\sigma_{i+1}^{\mathbf{B}_2}(a_{k_1,i+1},\ldots,a_{k_m,i+1}))$$

where $\sigma_j = \varphi_{mj}(\mathbf{a}_1^{(j)},\ldots,\mathbf{a}_m^{(j)},\sigma)$ and

$$\mathbf{a}_t^{(j)} = (a_{k_t 1},\ldots,a_{k_t,j-1},a_{k_t,j+1},\ldots,a_{k_t,i+1}),\quad t = 1,\ldots,m.$$

Consequently, it is enough to show that $a_{kj} = \sigma_j^{\mathbf{B}_2}(a_{k_1 j},\ldots,a_{k_m j})$ is valid, for all j, $j = 1,\ldots,i+1$. But this is obvious, since $\sigma_j^{\mathbf{B}_2}$ is the m-ary operation induced by the elements σ, $\mathbf{a}_1^{(j)},\ldots,\mathbf{a}_m^{(j)}$, for all j, $j = 1,\ldots,i+1$. Therefore, $\mathbf{B}^{(w)} \in \nu_i(\mathbf{B}_2)$.

To complete our proof, we show that $\mathbf{B}_{2^{i+1}} \notin \nu_{i-1}(\bar{M})$ provided that $i > 1$. For verifying this statement, the same idea can be used as in the proof of Theorem 2. Namely, if $\mathbf{B}_{2^{i+1}}$ can be embedded isomorphically into a ν_{i-1}-product $\prod_{j=1}^{n}\mathbf{A}_j$ of tree automata from $\bar{M}$, then $\mathbf{B}_{2^{i+1}}$ can be embeded isomorphically into a ν_i-product of tree automata from $\bar{M}$ with at most i factors. But the number of the elements of the latter product is not greater than 2^i which is a contradiction. Consequently, $\nu_{i-1}(\bar{M}) \subset \nu_i(\bar{M})$ is valid for all $i \geq 2$ which completes the proof of Theorem 3.

4 Star-product

This composition can be visualised in such a way that there is a central tree automaton and its behaviour depends on itself and the other ones, while each further tree automatan depends on itself and on the central one.

For giving the formal definition, let $\mathbf{A} = (A, \Sigma) \in \mathcal{U}_R$ and $\mathbf{A}_j = (A_j, \Sigma^{(j)}) \in \mathcal{U}_R$, $j = 1, \ldots, n$. Furthermore, take a family φ of mappings defined by

$$\varphi_{0j} : \Sigma_0 \to \Sigma_0^{(j)} \text{ provided that } 0 \in R,$$

$$\varphi_{m1} : (A_1 \times \ldots \times A_n)^m \times \Sigma_m \to \Sigma_m^{(1)}, \quad 0 \neq m \in R,$$

$$\varphi_{mj} : (A_1 \times A_j)^m \times \Sigma_m \to \Sigma_m^{(j)}, \quad 0 \neq m \in R, \ 2 \leq j \leq n.$$

It is said that the tree automaton $\mathbf{A}$ is a *star-product* of $\mathbf{A}_j$, $j = 1, \ldots, n$, with respect to φ if the following conditions are satisfied:

(a) $$A = \prod_{j=1}^{n} A_j \ ,$$

(b) if $0 \in R$, then for every $\sigma \in \Sigma_0$, $\sigma^{\mathbf{A}} = (\sigma_1^{\mathbf{A}_1}, \ldots, \sigma_n^{\mathbf{A}_n})$ holds where $\sigma_j = \varphi_{0j}(\sigma)$, $j = 1, \ldots, n$,

(c) for any $m \in R$, $\sigma \in \Sigma_m$, $((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn})) \in A^m$,

$$\sigma^{\mathbf{A}}((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn})) =$$
$$(\sigma_1^{\mathbf{A}_1}(a_{11}, \ldots, a_{m1}), \ldots, \sigma_n^{\mathbf{A}_n}(a_{1n}, \ldots, a_{mn}))$$

where $\sigma_1 = \varphi_{m1}((a_{11}, \ldots, a_{1n}), \ldots, (a_{m1}, \ldots, a_{mn}), \sigma)$ and

$$\sigma_j = \varphi_{mj}((a_{11}, a_{1j}), \ldots, (a_{m1}, a_{mj}), \sigma), \quad j = 2, \ldots, n.$$

We shall use the notation $\prod_{j=1}^{n} \mathbf{A}_j(\Sigma, \varphi)$ for the product introduced.

Let $\mathcal{B}$ be a system of tree automata from $\mathcal{U}_R$. It is said that $\mathcal{B}$ is *isomorphically complete for $\mathcal{U}_R$ with respect to the star-product* if any tree automaton from $\mathcal{U}_R$ can be embedded isomorphically into a star-product of tree automata from $\mathcal{B}$.

As far as the isomorphically complete systems of tree automata with respect to the star-product are concerned, we have the following statements.

Theorem 4. *A system $\mathcal{B}(\subseteq \mathcal{U}_{\{0\}})$ of tree automata is isomorphically complete for $\mathcal{U}_{\{0\}}$ with respect to the star-product if and only if there exists an $\mathbf{A} \in \mathcal{B}$ such that $\mathbf{B}_2$ can be embedded isomorphically into a star-product of $\mathbf{A}$ with a single factor.*

Now, let us suppose that $R \neq \{0\}$. Then, the following statement is valid.

Theorem 5. *A system $\mathcal{B}(\subseteq \mathcal{U}_R)$ of tree automata is isomorphically complete for $\mathcal{U}_R$ with respect to the star-product if and only if, for any positive integer k, there exists an $\mathbf{A} \in \mathcal{B}$ such that $\mathbf{B}_k$ can be embedded isomorphically into a star-product of $\mathbf{A}$ with a single factor.*

Proof. The sufficiency is based on the fact that any tree automaton of k states from $\mathcal{U}_R$ can be embedded isomorphically into a star-product of $\mathbf{B}_k$ with a single factor. The necessity can be proved in a similar way as that of Theorem 2. Namely, if $\mathbf{B}_{k^2}$ can be embedded isomorphically into a star-product $\prod_{j=1}^{n} \mathbf{A}_j(\Theta^{(k^2)}, \varphi)$, then $\mathbf{B}_k$ can be embedded isomorphically into a star-product $\mathbf{A}_j(\Theta^{(k)}, \varphi')$ with a single factor for some $j \in \{1, \ldots, n\}$.

Theorems 1, 2, 4, and 5 imply the following observation.

Corollary 2. For every rank-type R and every positive integer i, the star-product is equivalent to the ν_i-product with respect to the isomorphically complete systems of tree automata.

References

1. Dömösi, P., B. Imreh, On ν_i-products of finite automata, *Acta Cybernetica* **6** (1983), 149-162.
2. Dömösi, P., Z. Ésik, On the hierarchy of ν_i-products of automata, *Acta Cybernetica* **8** (1988), 315-323.
3. Dömösi, P., Z. Ésik, B. Imreh, On product Hierarchies of automata, Lecture Notes in Computer Science 380, FCT'89, (ed. J. Csirik, J. Demetrovics, F. Gécseg), Springer- Verlag, Proc.,137-145.
4. Dömösi, P., F. Gécseg, Simulation and representation by ν_i^*-products of automata, *Publicationes Mathematicae* (Debrecen) **40** (1992), 75-83.
5. Ésik, Z., A note on the isomorphic simulation of automata by networks of two-state automata, *Discrete Applied Mathematics* **30** (1991), 77-82.
6. Gécseg, F., On ν_i-product of commutative automata, *Acta Cybernetica* **7** (1985), 55-59.
7. Gécseg, F., Metric representation by ν_i-products, *Acta Cybernetica* **7** (1985), 203-209.
8. Gécseg, F., Composition of automata, Proceedings of the 2nd Colloquium on Automata, Languages and Programming, Saarbrücken, 1974, Springer Lecture Notes in Computer Science **14**, 351-363.
9. Gécseg, F., B. Imreh, A comparison of α_i-products and ν_i-products, *Foundations of Control Eng.* **12** (1987), 1-9.
10. Gécseg, F., B. Imreh, On star-product of automata, *Acta Cybernetica*, **9** (1989), 43-46.
11. Gécseg, F., H. Jürgensen, On α_0-ν_1-product, *Theoretical Computer Science* **80** (1991), 35-51.
12. Gécseg, F., M. Steinby, *Tree automata*, Akadémiai Kiadó, Budapest, 1984.
13. Imreh, B., A note on the ν_1-product, Acta Cybernetica **8** (1988), 242-252.
14. Tchuente, M., Computation on binary tree-networks, Discrete Applied Mathematics **14** (1986), 295-310.
15. Tchuente, M., Computations on finite networks of automata, Lecture Notes in Computer Science **316** (1988), 53-67.

This article was processed using the LaTeX macro package with LLNCS style

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE *LIRA '97*
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1–4, 1997, PP. 85–90

Algorithm for PP-reduction a PC Formula to the Clause Form

Mirjana Isaković-Ilić[1], Nebojša Bosiočić[2]
[1]*Šumarski fakultet, Kneza Višeslava 1, 11000 Beograd*
[2]*Savezno ministarstvo za odbranu, Kneza Miloša 33, 11000 Beograd*

Abstract - *Algorithm for the PP-reduction a Propositional Calculus formula to the clause form, which can be used for every binary operator, is described in the paper. The consequence of its application is that disjunction of each two clauses is a satisfied clause. Furthermore, the inferring Resolution Rule may be replaced by the given "summation" rule.*

Key words: Automated theorem proving, Resolution, Principle of Contradiction

1. Introduction

In the Resolution theorem proving system, the theorem of the Propositional Calculus is proved by transforming the negation of the formula to the conjunctive normal form, i. e. to the set of the clauses.

The system of the new rules for constructing the set of the clauses of the Propositional Calculus formula, is introduced in [1]. Clause term has been "raised" so that the elements of the clauses are signed subformulas of the negation of the given formula, not necessary the atomic ones, as in the classic case. If the elements of the clauses are atomic, the clause is considered as a finished one, in the sense that further application of the rules is not possible any more.

Crucial point of the new system is the introduction of the Contradiction Principle:

$$\text{PP:}\quad \frac{\{\Delta\}}{\begin{array}{l}\{\Delta, \mathrm{T}\,A\}\\ \{\Delta, \mathrm{F}\,A\}\end{array}}$$

where the reduction rules are generated from:

$$P_{PP1}:\quad \frac{\{\Delta, \mathrm{F}\,A\wedge B\}}{\{\Delta, \mathrm{F}\,A, \mathrm{F}\,B\}} \qquad \frac{\{\Delta, \mathrm{T}\,A\vee B\}}{\{\Delta, \mathrm{T}\,A, \mathrm{T}\,B\}} \qquad \frac{\{\Delta, \mathrm{T}\,A\to B\}}{\{\Delta, \mathrm{F}\,A, \mathrm{T}\,B\}}$$

$$P_{PP2}:\quad \frac{\{\Delta, \mathrm{T}\,A\wedge B\}}{\begin{array}{l}\{\Delta, \mathrm{T}\,A\}\\ \{\Delta, \mathrm{F}\,A, \mathrm{T}\,B\}\end{array}} \qquad \frac{\{\Delta, \mathrm{F}\,A\vee B\}}{\begin{array}{l}\{\Delta, \mathrm{F}\,A\}\\ \{\Delta, \mathrm{T}\,A, \mathrm{F}\,B\}\end{array}} \qquad \frac{\{\Delta, \mathrm{F}\,A\to B\}}{\begin{array}{l}\{\Delta, \mathrm{T}\,A,\}\\ \{\Delta, \mathrm{F}\,A, \mathrm{F}\,B\}\end{array}}$$

Including this principle, *semantic rules* [1] need to be added. They can be divided into three groups, according to the way they reduce the clauses: $\{\Delta, Z_1\ A*B, Z_2\ C\}$ (where: Δ = disjunction of some signed subformulas of the starting formula; $*$ = binary logic operation $\wedge$, $\vee$ or $\rightarrow$; C = one of the formulas A or B; Z_i = sign of the formula, and may be 'T' or 'F'):

$$P_0: \frac{\{\Delta, Z_1\ A*B, Z_2\ C\}}{-} \qquad P_1: \frac{\{\Delta, Z_1\ A*B, Z_2\ C\}}{\{\Delta, Z_3\ C\}} \qquad P_2: \frac{\{\Delta, Z_1\ A*B, Z_2\ C\}}{\{\Delta, Z_3\ A, Z_4\ B\}}$$

In spite of the fact that semantic rules were cited only for the clauses of the form:

$$\{\Delta, Z_1\ A*B, Z_2\ A\},$$

they can also be used on the clauses:

$$\{\Delta, Z_1\ \neg A*B, Z_2\ A\}$$

after their equivalent transforming to:

$\{\Delta, Z_1\ \neg A*B, Z_2'\ \neg A\}$, Z_2' is the opposite sign to Z_2.

Reduction follows as: first clause is the negation of the given formula. New clause is generated by applying one of the rules. The procedure is being repeated until one of the rules can be applied, namely, while there exists a clause with at least one non atomic clause.

2. Semantic of the rules

The rules P_0, P_1 and P_2 are the "compressing" ones: either they delete the clause (P_0), or decrease the number of formulas (i. e. disjuncts of the conjunctive normal form) inside (P_1) or simplify its structure (P_2). The rules P_{PP} are the only "expanding" ones: either they increase the number of disjuncts within the clause or increase the number of clauses in the deducing. The sequence of their application is naturally imposed: if it is possible, the rules P_0 are applied first, then the rules P_1, P_2 and finally P_{PP}. As the consequence, the application of the rules P_{PP} are avoided, if for that disjunct, the application of one of the "compressing" rules is possible. As there are disjuncts permitting the application of the both, P_{PP} and one of the P_0, P_1 or P_2 rules, the question of signification of the application the P_{PP} rules, for that clause, may be asked. In that sense, for each clause, the set of the *significant rules* for reduction, may be chosen.

Definition 1.

i) *The application of the P_i rule $i \in \{0,1,2\}$, is significant.*

ii) *The application of the P_{PP} rule is significant,* only if it is related to the disjunct for which the application of one of the P_i, $i \in \{0,1,2\}$, rules is not possible in the clause.

iii) For the clause, the *set of the significant rules* is built by all rules whose application is significant.

As the rules are applied to the clause, but effectively related to one or two disjuncts (always the formula and one of its main subformulas corresponds to them) the next types of the disjuncts may be established:
- the pair of disjuncts $\{d_1,d_2\}$ is of the type i, if the P_i rule, $i\in\{0,1,2\}$, may be applied;
- the disjunct is of the type PP (PP-disjunct) if the P_{PP} rule (P_{PP1} or P_{PP2}), may be applied; it should be noticed that *all non atomic disjuncts in the clause*, are PP-disjuncts.

The next notion of the *i-pair*, needs to be introduced:

Definition 2. If the pair of disjuncts $\{d_1,d_2\}$ is of the type i, $i\in\{0,1,2\}$, the ordered pair (d_1,d_2) is *i-pair*, if d_2 corresponds to the main (one of the existing two) subformula of the formula of the d_1. For the PP-disjunct d, (d,0) is *PP-pair*.

The presence of the pair of some type, for the clause, means the possibility of application the corresponding rule. In that sense, as the disjunct d may be paired both in the i-pair, $i\in\{0,1,2\}$ and in the PP-pair, the *significance of the pair* may be considered.

Definition 3. *Significant pair* is every i-pair, $i\in\{0,1,2\}$ and only that PP-pair (d,0), whose disjunct d is not paired within some i-pair, $i\in\{0,1,2\}$. All significant pairs of the clause build its *set of the significant pairs* (SZP).

Reduction algorithm is the algorithm for the application of the rules, i. e. the algorithm of forming the clauses and their sets of the significant pairs. As the clause and its SZP are always observed together, it is natural to include the next definition:

Definition 4. *Overclause* is ordered pair (S,P), where S is clause, and P is its set of significant pairs.

It is possible to prove the next Lemma:

Lemma 1. For each binary operation * (one of the 16), next two statements are valid:

i) $(A*X)\vee A$ may be equally transformed to one of the formulas: A, $X\vee A$, $\neg X\vee A$, True

ii) $(A*X)\vee\neg A$ may be equally transformed to one of the formulas $\neg A$, $X\vee\neg A$, $\neg X\vee\neg A$, True.

Corollary 1: Developing the clause $\{\Delta, Z_1\ A*X, Z_2\ A\}$ or $\{\Delta, Z_1\ \neg A*X, Z_2\ A\}$, one of the clauses: $\{\Delta, Z_2\ A\}$, $\{\Delta, Z_1\ X, Z_2\ A\}$, $\{\Delta, Z_1'\ X, Z_2\ A\}$ is always obtained, or it is deleted from the set of the clauses, as satisfied.

Corollary 2: Applying the rule to overclause (S,P), means:

1. rule P_1 for 1-pair (d_1,d_2):
   - delete d_1 from the clause S;
   - delete all pairs participating d_1, from SZP; the disjuncts which are paired only with d_1, become PP-disjuncts, so complete SZP, with corresponding PP-pairs.
2. rule P_2 for 2-pair (d_1,d_2):
   - delete d_1 from the clause S; add the disjunct related to the other main subformula of the formula of d_1 (as d_2 is related to one main subformula of the formula of d_1,) with corresponding sign, to the clause, only if that disjunct or its conjugate one is not inside the clause; if the conjugate of the disjunct is present, delete the overclause, because the clause is satisfied one.
   - delete all pairs participating d_1, from SZP; the disjuncts which are paired only with d_1, become PP-disjuncts, so complete SZP, with corresponding PP-pairs; if the disjunct has been added in the clause, establish the pairs it makes with the other disjuncts in the clause and complete the SZP.
3. rule P_{PP} for PP-pair (d,0):
   - decide about the type of the P_{PP} rule;

3.1 rule P_{PP1} for PP-pair (d,0):
   - delete d from the clause S; add to the clause two new disjuncts, related to main subformulas of the formula of d, signed according to the corresponding rule;
   - delete pair (d,0) from SZP and, for added disjuncts, establish the pairs they make with the other disjuncts in the clause and complete the SZP.

3.2 rule P_{PP2} for PP-pair (d,0):
   - make another overclause $(S_1,P_1) = (S,P)$;
   - delete d from the clause S; add to the clause two new disjuncts, related to main subformulas of the formula of d, signed according to the corresponding rule;
   - delete pair (d,0) from P and for added disjuncts, establish the pairs they make with the other disjuncts in the clause and complete the SZP P;
   - delete d from the clause S_1; add to the clause the disjunct, related to the one main subformula of the formula of d, signed according the corresponding rule;
   - delete pair (d,0) from P_1 and for added disjunct, establish the pairs it makes with the other disjuncts in the clause and complete the SZP P_1.

3. Reduction algorithm

Step 1. For the formula X, first overclause is ({F X},{(F X,0}).

Step 2. Next overclause is deduced from the overclause with non empty SZP as:

2.1 If there is 0-pair, in the set of significant pairs, apply P_0 rule and go to step 3. If not, go to 2.2.

2.2 If there is 1-pair, in the set of significant pairs, apply P_1 rule and go to step 3. If not, go to 2.3.

2.3 If there is 2-pair, in the set of significant pairs, apply P_2 rule and go to step 3. If not, go to 2.4.

2.4 If there is PP-pair, in the set of significant pairs, apply PP rule and go to step 3.

Step 3. Find overclause with non empty SZP and go to step 1. If such an overclause doesn't exist, go to step 4.

Step 4. If the empty set of the clauses is generated, formula is not the theorem. In the opposite case, conclude the proof including the Resolution Rule.

4. Algorithm application corollaries

After the algorithm application, the sequence of the clauses S_1, S_2, ..., S_m is generated, with the next properties:

Corollary 1. For each of the two clauses S_i and S_j it is valid: $S_i \vee S_j = 1$.

Corollary 2. Clauses are distinct.

Corollary 3. Two clauses, written in the canonical conjunctive normal form, cannot have the same elementary canonical disjunctions.

Let q_1, q_2, ..., q_n be all propositional letters present in the generated clauses. Then it may be introduced the next presentation form for the clauses:

$$\bigwedge_{i=1}^{m} S_i = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} q_j^{\alpha_{ij}}$$

where: if q_j is in the i-th clause with sign 'T', $\alpha_{ij} = 1$
if q_j is in the i-th clause with sign 'F', $\alpha_{ij} = -1$
if q_j is not in the i-th clause, $\alpha_{ij} = 0$

The *matrix A of signs* may be consequently generated for each formula:

$$A = \begin{bmatrix} \alpha_{11} \cdots \alpha_{1n} \\ \vdots \quad\quad \vdots \\ \alpha_{m1} \cdots \alpha_{mn} \end{bmatrix}$$

By summing the column elements, the difference between the number of the propositional letter and the number of its negation, is obtained. The rows containing zero(s), are related to the clauses which are the minimal form of more elementary canonical disjunctions (2, 4, 8,...). Through this fact, the validity of the formula can be decided:

Theorem 1. If the next statement is true for the matrix of signs of the formula $\neg X$:

$$\sum_{i=1}^{m} \alpha_{ij} 2^{k_i} = 0, \text{ for every } j = 1, 2, ..., n; \text{ where } k_i \text{ is number of zeroes in i-th row}$$

then the formula is a theorem. If there is an index j for which the conditions fail, then the formula is not a theorem.

5. Conclusion

In the Propositional Calculus theorem proving systems, based on resolution:
- multiple generating of the same clause,
- deducing of satisfied clauses,
- generating more clauses which semantically correspond to only one clause

is possible. These problems are solved by heuristics, which are based on the search over the set of the clauses. Another problem is the application of the Resolution Rule, which in every step generates one clause more, expanding the set of available clauses.

The establishment of the algorithm for generating final clauses, which have next properties:
- there aren't the same disjuncts within the clause,
- each clause is not satisfied,
- disjunct of each two clauses is a satisfied clause,
- each two clauses are distinct

becomes possible after introducing the Contradiction Principle and the corresponding semantic rules in the reduction system. As a consequence, it is possible to conclude the proof with the clauses generated by the algorithm, without application of the Resolution Rule, due to the given "summation" rule.

References

[1] Isaković, M. and Bosiočić, N. "*Jedno poboljšanje iskazne rezolucije*", Zbornik radova YU INFO '96.

[2] Fitting, M. *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, New York, 1990.

[3] D'Agostino, M. and Mondadori, M. *"An Improvement of Analytic Tableaux"*, 1992.

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 91–101

A comparison of decision procedures in Presburger arithmetic

Predrag Janičić, Ian Green and Alan Bundy

Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, Scotland
e-mail: janicic@opennet.org, {I.Green, A.Bundy}@ed.ac.uk

Abstract. It is part of the tradition and folklore of automated reasoning that the intractability of Cooper's decision procedure for Presburger integer arithmetic makes is too expensive for practical use. More than 25 years of work has resulted in numerous approximate procedures via rational arithmetic, all of which are incomplete and restricted to the quantifier-free fragment. In this paper we report on an experiment which strongly questions this tradition. We measured the performance of procedures due to Hodes, Cooper (and heuristic variants thereof which detect counterexamples), across a corpus of 10 000 randomly generated quantifier-free Presburger formulae. The results are startling: a variant of Cooper's procedure outperforms Hodes' procedure, and is fast enough for practical use. These results contradict much perceived wisdom that decision procedures for integer arithmetic are too expensive to use in practice.

1 Introduction

A *decision procedure* for some theory is an algorithm which for every formula tells whether it is valid or not. The role of decision procedures is critical in many areas, including theorem proving. As Boyer and Moore wrote [2, §1]

> "It is generally agreed that when practical theorem provers are finally available they will contain both heuristic components and many decision procedures."

The first author is supported by The British Scholarship Trust. The other authors are supported in part by grants EPSRC GR/L/11724, and British Council ROM/889/95/70.

Indeed, even (generally) inefficient decision procedures could reduce the search space of heuristic components of a prover and increase its abilities: a decision procedure can both close a branch in a proof, and reject non-theorems. Decision procedures can also have a important role in other areas such as geometry and type checking, for example.

A core part of automatic theorem proving involves reasoning with integer and natural numbers. Since the whole of integer arithmetic is undecidable, we are forced to look for 'useful', decidable sub-theories; there is a trade-off between usefulness and the complexity of associated decision procedures. In this paper, we take Presburger arithmetic: it is useful and there are a number of decision procedures.

In this paper we want to compare two decision procedures (and some simple variations thereof): that due to Hodes for Presburger rational arithmetic and that due to Cooper, for Presburger integer arithmetic. It is part of the tradition and folklore of automated reasoning that Cooper's decision procedure for Presburger integer arithmetic is too expensive to be of practical use. More than 25 years of work has resulted in numerous approximate procedures via rational arithmetic, all of which are incomplete and restricted to the quantifier-free fragment. It is known that the (worst-case) time complexity of Cooper's procedure is $2^{2^{2^n}}$ in the size of the formula,[1] and moreover, this is much worse than Hodes' procedure. Boyer and Moore state [2, §3]

> "... integer decision procedures are quite complicated compared to the many well-known decision procedures for linear inequalities over the rationals [...]. Therefore, following the tradition in program verification, we adopted a rational-based procedure..."

It is this 'tradition' of work in the rationals [1, 9, 10, 2] that we question in this paper. Anecdotal testimonies to this tradition abound in the literature but we are not aware of any experimental comparison of procedures.

Here we report on an experimental comparison of procedures on 10 000 randomly generated formulae. However, the results are very surprising—broadly, they show that a simple variant of Cooper's procedure *outperforms* Hodes' procedureon our sample corpus!

These results cast some doubt on the perceived wisdom in the automated reasoning community that full decision procedures for integer arithmetic are too expensive to use in practice.

Overview of paper. §2.1 defines Presburger arithmetic, procedures and notation. §3 describes the test corpus and the experiments we made on it; §4 shows the results. §5 and §6 discusses further work and draws conclusions.

[1] Shostak [9] attributes this result to Oppen.

2 Background

We write $\models_T f$ ($\not\models_T f$) to mean f is valid (invalid) in theory T. A *decision procedure* for theory T is a total function d from formulae to the set $\{\texttt{yes}, \texttt{no}\}$, having for any f the properties of soundness $d(f) = \texttt{yes}$ implies $T \models f$, and completeness, $T \models f$ implies $d(f) = \texttt{yes}$. An *incomplete decision procedure* is sound but not complete.

2.1 Presburger arithmetic

Presburger arithmetic is (roughly speaking) a theory built up from the constant 0, variables, binary $+$, unary s, relations $<$, $>$, $=$, $\leq$, $\geq$ and the standard connectives and quantifiers of first-order predicate calculus.[2] The notions of term, atomic formula and formula are formally defined in the usual way (a grammar is given in figure 1 for the quantifier-free part). In *Presburger integer arithmetic* (PIA), variables range over the integers. It was Presburger who first showed that PIA is decidable [8]. *Presburger rational arithmetic* (PRA) is defined analogously and is also decidable [7].

The restriction of Presburger integer arithmetic to Peano numbers (i.e., to non-negative integers) we call *Presburger natural arithmetic* (PNA).[3]

2.2 Related work

Some decision procedures for Presburger arithmetic are based on the idea of quantifier elimination described by Kreisel and Krivine [7]—these are Hodes' procedure for Presburger rational arithmetic [6] and Cooper's procedure for Presburger integer arithmetic [5]. There is also the Sup-Inf family of procedures due to Bledsoe [1] and latterly improved by Shostak [9, 10].

The rational-based procedures are an attempt to overcome the complexity of integer-based procedures. It can be easily seen that there are formulae valid in rational arithmetic and invalid in natural arithmetic and vice versa. For instance, $\exists x.2x = 3$ is valid over the rationals, but not over the naturals. Also, $\forall x.x \leq 1 \vee x \geq 2$ is valid over the naturals, but not over the rationals. Therefore, we cannot use a decision procedure for one of these two theories in the other, not even as an incomplete decision procedure.

However, if some universally quantified PNA formula is decided valid by Hodes' procedure (i.e., taking it to be a formula of PRA), then it must be valid

[2] Multiplication of a variable by a constant is also Presburger: nx is treated as $x + \cdots + x$, where x appears n times. We shall write 1 instead of $s(0)$, etc.

[3] For the same theory and some related theories there are a few different terms used. For instance, Hodes calls Presburger rational arithmetic a *theory EAR* ("the elementary theory of addition on the reals"). Boyer and Moore [2] describe a universally quantified fragment of Presburger rational arithmetic as *linear arithmetic* (although in fact they work over the integers); that same theory sometimes goes by the name *Bledsoe real arithmetic*.

in PNA. That is, $\models_{PRA} s$ implies $\models_{PNA} s$, for the universally quantified formula s. The reverse implication does not hold, and so Hodes' procedure is not a decision procedure for PNA.

This idea of applying decision procedures for rationals to the integer case is at the heart of Bledsoe's Sup-Inf method [1], and can be seen as the start of the tradition of incomplete decision procedures. The tradition continued with Shostak's improved Sup-Inf [9, 10]; he showed it could decide invalid formulae, and so was indeed a decision procedure. However, the class of formulae for which Shostak's Sup-Inf decides has not been characterized syntactically: it is not a decision procedure for universally quantified PIA, but for some "semantically characterized" fragment.

Boyer and Moore too followed this track [2], although they reverted back to Hodes' procedure rather than using Sup-Inf. Their choice was unsurprising in some sense, since the Nqthm logic is quantifier-free, so the restriction for soundness is vacuous. Somewhat curiously, Boyer and Moore conclude in that same paper that efficiency of the DP is largely irrelevant in the wider setting of a heuristic prover: in that case why not use Cooper's procedure, and have a complete procedure to boot?[4] There is a question as to what degree *negative* results from a decision procedure can be used in a heuristic theorem prover. Some systems, such as CLaM [3], can use this information, for example, in controlling generalization, and other non-equivalence preserving heuristics. This is undoubtedly true of other theorem provers.

3 Experiments

3.1 Generating Presburger formulae

We randomly generated a corpus of 10 000 formulae of Presburger arithmetic. This was done using the grammar shown in figure 1 to generate quantifier-free formulae containing free variables (taken from a set of five symbols). Each rule was chosen with a probability given in the right-hand column.

3.2 Algorithms considered

In addition to Hodes' procedure and Cooper's procedure, we also used variants of these, using a heuristic that quickly rejects invalid formulae (we will call it the QR heuristic).

The heuristic is as follows: to invalidate $\forall \mathbf{x}.\Phi(\mathbf{x})$ we show that a particular instance $\Phi(\mathbf{c})$ is invalid. That is, we instantiate all universally quantified variables in a formula $\forall \mathbf{x}.\Phi(\mathbf{x})$ with particular ground values (say 0 and 100) in all ways. In that way we get a quantifier free formula $\Phi(\mathbf{c})$, for which validity is quickly decided. This simple heuristic is obviously sound, but not complete. However,

[4] We emphasize that the Nqthm decision procedure is stronger than Hodes' procedure since it also has heuristics to deal with non-Presburger formulae by calling the inductive part of the prover.

Rule	*Probability*
⟨formula⟩ := ⟨atomic formula⟩	0.75
⟨formula⟩ := ¬ ⟨formula⟩	0.1
⟨formula⟩ := ⟨formula⟩ ∨ ⟨formula⟩	0.05
⟨formula⟩ := ⟨formula⟩ ∧ ⟨formula⟩	0.05
⟨formula⟩ := ⟨formula⟩ ⇒ ⟨formula⟩	0.05
⟨atomic formula⟩ := ⟨term⟩ = ⟨term⟩	0.2
⟨atomic formula⟩ := ⟨term⟩ < ⟨term⟩	0.2
⟨atomic formula⟩ := ⟨term⟩ ≤ ⟨term⟩	0.2
⟨atomic formula⟩ := ⟨term⟩ > ⟨term⟩	0.2
⟨atomic formula⟩ := ⟨term⟩ ≥ ⟨term⟩	0.2
⟨term⟩ := ⟨term⟩ + ⟨term⟩	0.2
⟨term⟩ := s(⟨term⟩)	0.2
⟨term⟩ := 0	0.2
⟨term⟩ := ⟨variable⟩	0.4

Fig. 1. Grammar of Presburger arithmetic and probabilities assigned to rules for generating a corpus

our experiments showed that this heuristic could be very important and very useful.[5]

Thus we compared the following four procedures:

Hodes' procedure. For PRA. Recall from §2.2 that this procedure is incomplete even for the universally quantified fragment of PNA.

Cooper's procedure. For PNA. (Cooper presented two such procedures; we used the second, improved version [5]. Besides, Cooper's procedure is originally defined for Presburger integer arithmetic, and in our experiments we used our version, slightly modified for PNA.)

DP-A . For a given formula try to disprove it using the QR heuristic; if it succeeds, the formula is invalid; otherwise, apply Cooper's procedure; if Cooper's procedure says `yes`, the formula is a theorem, otherwise it is not.

DP-B . For a given formula try to disprove it using the QR heuristic; if QR succeeds, the formula is invalid; otherwise, if the given formula is universally quantified, apply Hodes' procedure; if the answer is `yes`, then the formula is valid, if the answer is `no` or if a given formula is not universally quantified, then apply Cooper's procedure; if the answer is `yes`, then the formula is valid, otherwise the answer is `no`, and it is invalid.

[5] Obviously, this procedure could be used for all types of formulae (not just universally quantified ones): using this procedure we could transform (simplify) a formula to existentially quantified formula and try to disprove it using Cooper's procedure.

4 Results

We ran the procedures described in §3.2 on each formula of the corpus, recording whether the formula was valid or invalid, and CPU time taken to decide, subject to a time limit of 100s. The following tables show the results with CPU time measured in milliseconds.[6]

4.1 QR heuristic contribution

Of the 10 000 formulae in the corpus, roughly 8000 were invalid, the remainder valid. DP-A using values of 0 and 100 was able to quickly reject all but 70 of these invalid formulae. (We decided to take values 0 and 100 because our experiments showed that additional values were not significantly contributing to the rejection rate, and on the other hand, using just 0, the heuristic rejected less than 5000 formulae.)

4.2 CPU time distribution

Table 1 shows number of formulae handled by procedures within a given time interval, together with mean CPU time (in ms). Each entry in the table is a pair, the first part of which is the number of formulae in that time interval, the second part is the mean time taken by the procedure. The totals column shows the total number of formulae handled by each procedure within 100 seconds, and the mean time for these formulae.

	CPU time (ms)				
Procedure	$< 10^2$	10^2–10^3	10^3–10^4	10^4–10^5	*Totals*
Hodes	9213/27	639/265	106/2546	28/30120	9986/154
Cooper	5678/48	3566/292	509/2620	120/31372	9873/650
DP-A	9361/18	552/278	51/2467	17/29012	9981/94
DP-B	9254/18	605/286	95/2298	22/30189	9982/122

Table 1. Number of formulae decided vs. CPU time

4.3 Effect of number of variables

Table 2 shows that the mean CPU time spent by the procedures increases with the number of variables. The number of formulae in the corpus containing a

[6] Programs were written in Quintus Prolog; experiments were run on a 32Mb Sun SPARC 4.

certain number of variables is shown, with a pair the first part of which is a percentage, the second is the mean CPU time.

The percentage is of those formulae having a particular number of variables completed within the 100 second time limit. In the 5-variable case Hodes' procedure processed 95.9%, but for Cooper's procedure this figure was 78.8%. This discrepancy seems to suggest that Cooper's procedure degrades with increasing number of variables; however, notice that DP-A performed *better* then Hodes' procedure. Thus the correct explanation is that Cooper's procedure is slower on *invalid formulae*, an effect seem more clearly in §4.5.

	# variables/# formulae					
Procedure	*0/598*	*1/3362*	*2/3603*	*3/1503*	*4/693*	*5/241*
Hodes	100/3	100/17	100/39	100/130	99.4/683	95.9/2883
Cooper	100/3	100/42	100/169	98.6/1202	92.1/3608	78.8/8306
DP-A	100/6	100/16	100/22	99.9/171	99.0/315	95.9/1432
DP-B	100/7	100/17	100/27	99.9/202	98.7/443	94.5/1978

Table 2. % completed/CPU time (ms) vs. number of variables

4.4 Effect of size of formula

The efficiency of Hodes' and Cooper's procedure is governed not just by propositional structure but by term structure too. To investigate this aspect of performance, we define the size of a formula/term as the sum of the sizes of its immediate subformulae/subterms plus one, taking the size of variables and constants as 0. Table 3 shows the results.

Cooper's procedure is most exposed here; the percentage of formulae decided within the time limit drops dramatically as the size increases. Hodes' on the other hand fares quite well. Once again though, the better performance of DP-A reveals that Cooper's procedure is struggling with invalid formulae which are more easily dealt with by the QR heuristic.

4.5 Effect of validity

Table 4 shows CPU time spent by the procedures according to validity of a formula (considering only those formulae decided by all procedures within the time limit).[7]

[7] 9868 formulae from our corpus were treated by all procedures—so, 132 formulae are 'missing' in table 4. However, our QR heuristic rejected 107 of them, and thus, at most 25 formulae could change the first two columns in the table 4 if we had taken

	Size/# formulae					
Proc.	*1–10/8785*	*11–20/1029*	*21–30/150*	*31–40/29*	*41–50/5*	*51–60/2*
Hodes	100/35	99.7/687	97.3/2772	79.3/4661	80.0/2430	100/1560
Cooper	99.8/225	93.7/3238	80.0/8655	55.2/10657	60.0/24690	50/38530
DP-A	100/24	99.3/460	94.7/1059	93.1/3156	80.0/4320	100/295
DP-B	100/31	99.2/647	94.0/1800	82.8/572	80.0/4537	100/310

Table 3. CPU time vs. size

The first column pertains to formulae valid in both rational and natural arithmetic;[8] the second is for those invalid in the rationals and valid in the naturals; the third for those invalid in both theories. Notice that those in the second column would not be found to be valid formulae of natural arithmetic by Hodes' procedure. Note that the heuristic versions dramatically improve the performance of Cooper's procedure in the invalid cases.

We considered the procedure DP-B , anticipating that it would take advantage of Hodes' procedure on formulae valid both in PRA and PNA. We expected that these gains would outweigh losses in other cases and therefore we expected, in general, DP-B to be faster than DP-A . However, surprisingly, it turned out that Hodes' procedure performed worse than Cooper's procedure in this group of formulae. Consequently, DP-B failed to improve upon DP-A in any of groups of formulae according to validity. Of course, DP-B cannot exploit Hodes' procedure in the case of formulae on which Hodes' procedure returns `no`, since Hodes' procedure is incomplete; in such cases, Cooper's procedure must be called, and time spent in Hodes' procedure is wasted.

4.6 Summary of results

On our corpus Cooper's procedure performed better than Hodes', on valid formulae, but it was much worse on invalid formulae. However, this is mitigated entirely by using the QR heuristic. Our conclusion then is that the combination of Cooper's procedure and QR performs better than Hodes' procedure.

---

higher time limit.

(Among the 'missing' formulae, there are formulae decided by one decision procedures and not decided by others, and some of them weren't decided by neither procedure. Generally, these formulae are formulae of large complexity, but it is a very difficult task to give some precise characterization of them — such a characterization would have to involve some deeper semantic knowledge.)

[8] Out of 756 formulae valid in both rational and natural arithmetic 330 are ground, 311 with 1, 51 with 2, 29 with 3, 27 with 4 and 8 with 5 variables.

Validity	$\models_{PRA} F$ $\models_{PNA} F$	$\not\models_{PRA} F$ $\models_{PNA} F$	$\not\models_{PRA} F$ $\not\models_{PNA} F$	*Total*
# formulae	756	1214	7898	9868
Hodes	117	256	81	105
Cooper	33	204	758	634
DP-A	66	267	49	77
DP-B	148	523	58	122

Table 4. CPU times according to validity/invalidity of formula

5 Future work

It would be interesting to compare Hodes' and Cooper's procedures with Sup-Inf procedures.

The assignment of probabilities to the rules of the grammar for Presburger formulae was chosen somewhat arbitrarily—we have yet to investigate the effect these parameters have on validity, time to decide etc. A quickly computed measure of expected run time might be useful in a heuristic theorem prover. One can imagine pursuing a line of enquiry similar to that in the propositional satisfiability community.

According to results of Boyer and Moore [2], an essential role of using an arithmetic decision procedure is to contribute to the proofs of deeper theorems in other theories (not just arithmetic). We have done some preliminary work in this direction in the CLaM proof-planning system [3]. Transformation of a problem to a Presburger formula should be done by a communication module. We used an extension of our procedure DP-A , which could handle defined arithmetic functions (*double*, *half*, *minus*, *p*) and relations (*odd*, *even*). For that purpose, we use rewrite rules, which, for instance, using the theorem $\forall x.\forall y.double(x) = y \Leftrightarrow y = 2x$, it is sound to rewrite $F(double(x))$ into $\forall y.2x = y \Rightarrow F(y)$. Similarly, $\forall x.even(double(x))$ can be rewritten to $\forall x.\exists u.\forall v.2x = v \Rightarrow v = 2u$. Notice that these translations move outside the quantifier-free fragment of Presburger arithmetic, so we are using DP-A in an area where Hodes' procedure would be unsound, and so useless. Preliminary results are encouraging, but much more work needs to be done. We also want to deal with non-arithmetic functions (e.g., length of a list), using more powerful techniques: our aim is to explore Bundy's idea of proof-plans for normalization [4].

6 Conclusions

The effectiveness of DP-A (and hence any claim that Cooper's procedure is useful in tandem with the QR heuristic) must be offset by the fact that 80% of the corpus is invalid. Although Cooper's procedure outperforms Hodes' procedure on valid formulae, it did so to a lesser degree than DP-A outperformed

Cooper's procedure on invalid formulae. Furthermore, another factor must be borne in mind. The corpus we generated did not contain many large constants, and the presence of these in formulae will often slow both Hodes' and Cooper's procedure, and hence DP-A . Note that the Sup-Inf family of procedures is not affected in this way.

We need to be cautious when advocating the use of DP-A more widely—more experiments on other corpora are required, especially on "real problems" generated during (say) inductive verification proofs. *With these caveats*, we draw the following conclusions from the corpus we used:

- Cooper's procedure was faster than Hodes' procedure on valid formulae; it was an order of magnitude slower on invalid formulae.
- Many invalid universally quantified formulae can be identified simply and quickly by checking ground instances over a small set of values.
- Cooper's procedure with the simple QR heuristic *outperformed* Hodes' procedures. This is a startling result. It goes against the grain of much work and commentary made on decision procedures in the past 25 years.
- When efficiency is comparable, it is highly preferable to use a decision procedure in a heuristic theorem prover rather than an incomplete decision procedure. Hodes' procedure (which is incomplete for quantifier-free PNA) fails to prove many PNA theorems; for such theorems much extra work may be incurred in trying other techniques (e.g., induction). We speculate that for most invalid PNA conjectures, even a slow decision procedure will be faster and more robust than heuristic techniques.
- Worst case analysis of complexity may be misleading: experimental evaluations can be useful.

On the basis of these experiments, we conclude that for quantifier-free Presburger arithmetic over the natural numbers, Cooper's procedure augmented with a 'quick reject' heuristic is superior to Hodes' procedure. This is a startling result that questions much of the perceived wisdom in the automated reasoning community.

References

1. W.W.Bledose. A new method for proving certain Presburger formulas. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, U.S.S.R., 3-8 September 1975.
2. R.S.Boyer and J.S.Moore. Integrating decision procedures into heuristic theorem provers: A case study for arithmetics. In J.E.Hayes, J.Richards and D.Mitchie, editors, *Machine Intelligence 11*, pages 83-124, 1988.
3. Alan Bundy, F. van Harmelen, C. Horn and A.Smaill. The Oyster-Clam system. In M.E.Stickel, editor, *10th International Conference on Automated Deduction*, pages 647-648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper No. 507.

4. Alan Bundy. The use of proof plans for normalization. In R.S.Boyer, editor, *Essays in Honor of Woody Bledsoe*, pages 149-166. Kluwer, 1991. Also available from Edinburgh as DAI Research Paper No. 513.
5. D.C.Cooper. Theorem proving in arithmetic without multiplication. In B.Meltzer and D.Mitchie, editors, *Machine Intelligence 7*, pages 91-99. Elsevier, New York, 1972.
6. Louis Hodes. Solving problems by formula manipulation in logic and linear inequalities. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 553-559. Imperial College, London, England, 1971.
7. Georg Kreisel and Jean Louis Krivine. *Elements of mathematical logic: Model theory*. North Holland, Amsterdan, 1967.
8. M.Presburger. Über die vőllstandigkeit eines gewissen systems der arithmetik ganzer zahlen in welchem die addition als einzige operation hervortritt. In *C.R 1er Congr. des Mathematiciens des PAys Slaves*, pages 92-101, Warsaw, 1930.
9. Robert E. Shostak. On SUP-INF method for proving Presburger formulas. *JACM*, 24(4):529-543, October 1977.
10. Robert E. Shostak. A practical decision procedure for arithmetic with multiplication for arithmetic with function symbols. *JACM*, 26(2):351-360, April 1979.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 103–110

Debugging and Testing of Prolog Programs

Gabriella Kókai[1], László Harmath[1] and Tibor Gyimóthy[2]

[1] Institute of Informatics József Attila University
Árpád tér 2, H-6720 Szeged Hungary
e-mail: kokai,harmat@inf.u-szeged.hu

[2] Research Group on Artificial Intelligence
Hungarian Academy of Sciences,
Aradi vértanuk tere 1, H-6720 Szeged Hungary
e-mail: gyimi@inf.u-szeged.hu.

Abstract. In this paper the IDTS (Integrated Debugging, Testing and Slicing), method is presented for the algorithmic debugging and functional testing of Prolog programs. This method integrates Shapiro's Interactive Diagnosis Algorithm with the Category-Partition Testing Method and a slicing technique. Shapiro's original method demands a lot of user interactions during the debugging process. The IDTS method can avoid irrelevant questions to the user by categorizing input parameters, and match them against test cases and test database. In addition, a slicing method is used to compute which parts of the program are relevant for the search. The IDTS method has since been used in a large ECG classifier program and in an interactive learner. [1]

1 Introduction

In this paper we present a method, called IDTS (Integrated Debugging, Testing and Slicing) for the algorithmic debugging and functional testing of Prolog programs. This method not only integrates Shapiro's Interactive Diagnosis Algorithm [12] with the Category Partition Testing Method (CPM) [10] but a common slicing technique [11],[5]. IDTS can improve the efficiency of the algorithmic debugging method for bug localization by using given test specifications and test results. The method can avoid irrelevant questions to the user by first categorizing input parameters, and then matching these against test cases in the test database. In addition, a slicing method is employed that is based on a program dependence graph to compute which parts of the program are relevant in the search, thus further improving bug localization. In the method presented the test database is modified during the debugging of a program and the test specification is improved. In this way our technique can reasonably be considered as an *integrated method for debugging and functional testing of logic programs.*

The basic concept behind IDTS is similar to the GADT (Generalized Algorithmic Debugging and Testing) system given in [4] [7]. The main difference

[1] This work was supported by the grant AMFK 193/96

between the GADT and the IDTS is that the GADT has been applied to imperative languages and the IDTS to logic programs.

As a large scale application the IDTS method has been used in an CG waveform classifier method called P CG [8].

The method has been also integrated in an interactive Inductive Logic Programming [9] learner called IMPUT [1].

In the remainder of this paper we furnish a brief overview of the IDTS method. Later short example is presented in Section 3. Then finally in Section 4 a summary and comments on future work are given.

1.1 The IDTS System for Prolog programs

IDTS combines Shapiro's algorithmic debugging technique [12] with the CPM functional testing method [6] and also a program slicing technique [11] to make it an even more advanced debugging system. The overall structure of the system more or less the following:

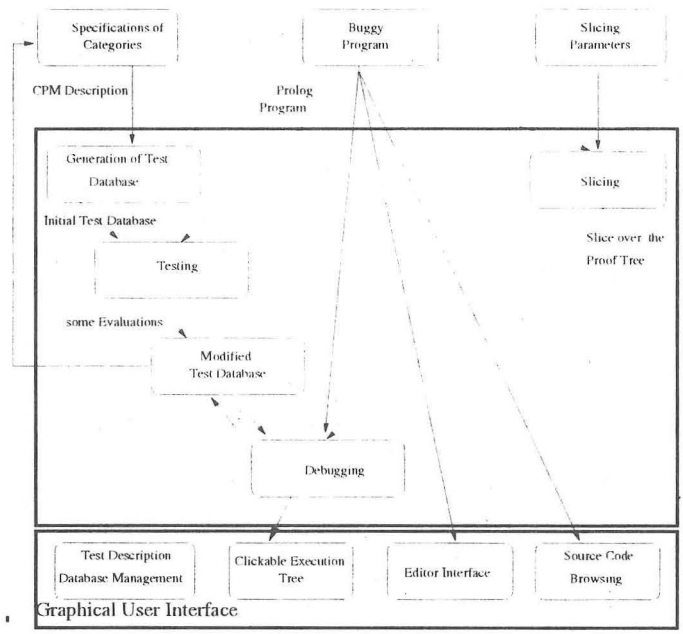

Fig. 1. The structure of the IDTS system

The algorithmic program debugging method introduced by Shapiro can isolate an erroneous procedure if a program and an input on which it behaves incorrectly is given. Shapiro's model was originally applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output and nontermination. A major drawback of this debugging approach is the great number of queries put to the user about the correctness of intermediate results of clause calls.

The main idea behind IDTS is the following. During the debugging of a program the user has to answer many difficult questions. If the program has already been tested, the test results of the procedures of the program can be directly applied to the debugging process without consulting the user. In addition the slicing method computes the relevant part of the proof tree.

1.2 Shapiro's Method for Algorithmic Debugging

Shapiro's algorithms can isolate any erroneous clause, given a program and an input on which it behaving incorrectly. The algorithm traverses the proof tree of a program in different ways and ask the user about the expected behaviour of each resolved goal.

The *bottom-up* method traverses the proof tree in postorder manner and asks the *oracle* about the correctness of the computed values of the nodes. If the result at a node is incorrect and all sons of this node are evaluated correctly results the algorithm identifies the clause applied to this node as a *buggy* one. The query complexity of this method is linear in the size of the tree.

The second method investigates the nodes in a *top-down* manner. If the result computed at a node is evaluated correctly by the *oracle* then the algorithm does not visit the nodes inside the sub-tree. Using this approach the query complexity can be reduced to a linear dependence in the depth of the proof tree.

The most efficient technique is the *divide-and-query* strategy which requires a number of queries logarithmic in the size of the proof tree. The divide-and-query algorithm splits the proof tree into two approximately equal parts, and makes a query for the node at the splitting point. If this node gives an incorrect evaluation the algorithm goes on recursively to the sub-tree associated with this node. If the node's answer is correct its sub tree is removed from the tree and a new mid-point is computed.

1.3 The CPM Testing Method

The CPM method for imperative programming languages has been defined in [10], with an extended version presented in [4]. A formal description of the CPM for logic programs can also be found in [6]. Informally this method can be outlined as follows: During the functional testing the programs (procedures) cannot be tested for all possible properties of their parameters. Hence the tester's first task is to define the critical properties (*categories*) of the input parameters which will be investigated in the testing process.

The categories are divided into disjoint classes called *choices* which presume that all the elements within a choice have an identical observable behaviour with respect to correctness. Once the categories and choices for a program have been defined all the possible *test frames* can be generated, each test frame covering exactly one choice from each category. In general there may be many superfluous test frames among the generated ones, namely property combinations that have no real relevance. These frames can be eliminated by associating *selector expressions* with the choices.

1.4 The Slicing Method

The program slicing part of IDTS is based on the *annotation* inference technique of [2]. Using this technique an annotating specification of directionality (*input, output*) can be automatically generated for the functional part of a logic program. The user may define the slicing point like the suspected *buggy* position in the program to start the slicing process.

From an annotation a *dependence graph* is constructed for the logic program [3]. A proof (refutation) tree is produced for a buggy program and using the dependence graph the tree is sliced, the segments having no obvious influence on the visible symptom of a bug being removed. The algorithmic debugger traverses the sliced proof tree only, thus concentrating on the suspect part of the program. The annotation of the program is helpful for preparing the test database as well, where the user may provide test cases on input arguments of the annotated program.

During the debugging the IDTS system does not investigate any node of a proof tree which is not in the sub tree determined by the slice of the *buggy* position.

1.5 The graphical user interface

The GUI supports the following activities

- *clickable execution trees:* In each step of the debugging process the GUI labels one node of the proof tree, the intended behaviour of which the debugger needs to know to carry on with the debugging process. Nodes can be hidden and extended. The user can answer by pressing one of the shown options *correct*, *incorrect* or *cancel*.
- *source code browsing*: The *static-call graph* shows which predicates could be invoked by a given procedure. With the help of this window a static program structure is displayable to a complement the dynamic proof tree window. *The information-retrieval display* shows the line number and the clauses matching the pattern selected by the user.
- *editor interface:* By pressing a mouse button over a node of the proof tree an editor window opens and displays the part of the Prolog program which corresponds to the node, the source code beeing editable also.
- *test description and database management:* All data is utilised by IDTS can be stored and loaded with a few mouse clicks.

2 Small Example in IDTS

In this section we demonstrate the operation of IDTS in finding a false clause in a *buggy program*.

2.1 The program compute

To illustrate how our integrated debugger works, a program *compute* (borrowed from [11] and listed in Appendix) was chosen which computes the difference between the lists in the predicate *complement* and the sum of their elements by the predicate *sigma*. Let us, say, introduce the following buggy version of clause *delete*:

```
(C8) delete(X,[Y|Z],[X|U]) :- delete(X,Z,U).
```

instead of the original clause

```
(C8) delete(X,[Y|Z],[Y|U]) :- delete(X,Z,U).
```

The CPM description for the predicates *member* and *delete* is listed the Appendix 4.2. The description begins with the line *type(predicate_name(...))* which defines whether the parameters of the predicate name are printed or not during the debugging sessions. The category-partition specification for predicate *member* contains two categories *number_of_elements* and *relation*. The first category is divided into three and the second into four disjoint classes. For example, category *relation* expresses the view that the number in the first argument of *member* is greater, smaller or equal to the first element of the second argument. The fourth choice *none* is used if the list in the second argument is empty. In each case searching functions are simple Prolog facts.

2.2 The questions asked in the debugging process

The most complex part of the system is the algorithmic debugging phase. Let us suppose that the top-down traversal algorithm is selected to demonstrate the usual questions asked by the system. In our example the system first builds up the proof tree (see in Figure 2) for the goal:

```
compute([1,2,3],[3],C,S).
```

If Shapiro's top-down method is employed (without CPM and slicing) the following questions are invoked to identify the buggy clause *delete*.

```
Is the goal sigma([1,2,3],[3],9) (y/n)? y
Is the goal complement([1,2,3],[3],[3,3]) (y/n)? n
Is the goal member(3,[1,2,3]) (y/n)? y
Is the goal delete(3,[1,2,3],[3,3]) (y/n)? n
Is the goal delete(3,[2,3],[3]) (y/n)? n
Is the goal delete(3,[3],[]) (y/n)? y
The false clause:
delete_ver(3,[2,3],[3]) :- delete_ver(3,[3],[])
```

If Shapiro's method is combined with the CPM technique the following questions will be asked:

```
Is the goal sigma([1,2,3],[3],9) (y/n)? y
Is the goal complement([1,2,3],[3],[3,3]) (y/n)? n
Is the goal member(3,[1,2,3]) (y/n)? y
Is the goal delete(3,[1,2,3],[3,3]) (y/n)? n
Is the goal delete(3,[3],[]) (y/n)? y
```

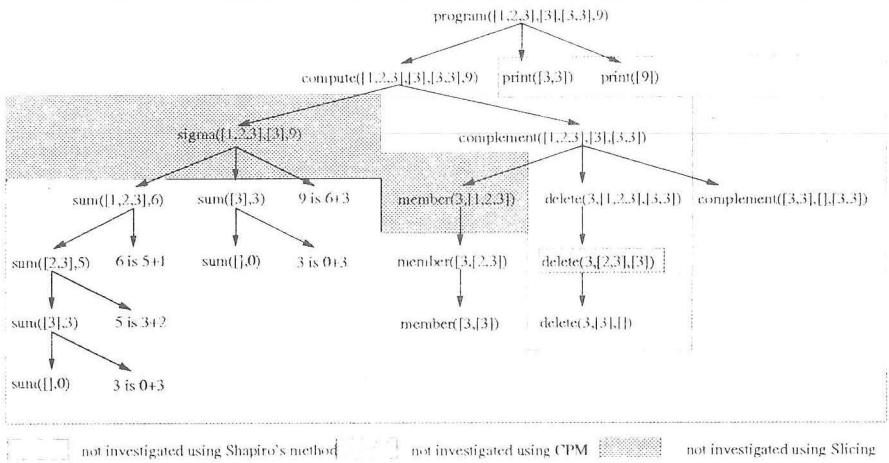

Fig. 2. The proof tree of the example

But, if the slicing method is used the following three questions are enough to identify the bug:

```
Is the goal complement([1,2,3],[3],[3,3]) (y/n)? n
Is the goal delete(3,[1,2,3],[3,3]) (y/n)? n
Is the goal delete(3,[3],[]) (y/n)? y
```

3 Conclusion, Further Work

In this paper a method has been presented which combines Category Partition Testing, the program slicing method with the algorithmic debugging techniques introduced in [12]. A similar method was presented in [4] to diagnose imperative programs but as far as is known the IDTS method presented in this paper is unique in the context of logic programming. This integrated system can be used in the testing and debugging of Prolog programs.

The first version of the IDTS method has been fully implemented. However, due to the poor implementation technique the computation of dependence graph required for slicing is very slow, hence we have embarked on a new implementation of this part. [2]

References

1. Alexin, Z., Gyimóthy, T., Boström, H.: Integrating *Algorithmic Debugging and Unfolding Transformation an Interactive Learner* In: Proceedings of the 12th

[2] The implementation language currently is SICStus Prolog Version 3.0 #3 or Quintus Prolog Version 3.2 running on a SUN SPARCStation.

European Conference on Artificial Intelligence ECAI-96 ed. Wolfgang Wahlster, Budapest, Hungary (1996) 403–407 John Wiley & Son's Ltd. 1996.
2. Boye J., Paakki J., Maluszyński J.:*Synthesis of Directionality Information for Functional Logic Programs.* In: Proc. 3rd Int. Workshop on Static Analysis, Padova, 1993. LNCS 724, Springer-Verlag, 1993, 165-177
3. Deransart P., Maluszyński J.: *Relating Logic Programs and Attribute Grammars* In: Journal of Logic Programming 2, 1985, 119-156.
4. Fritzson, P., Gyimóthy, T., Kamkar, M., Shahmeri, N.: *Generalized Algorithmic Debugging and Testing* In: Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario 1991. ACM SIGPLAN Notices 26, 6, (1991) 317–326
5. Gyimóthy T., Paakki J.:*Static Slicing of Logic Programs.* In: Proceedings of Automated and Algorithmic Debugging ed. Ducassé, M. Sain-Malo, France. (1995) 65-77
6. Horváth, T., Gyimóthy, T., Alexin, T., Kocsis, F.: *Interactive Diagnosis and Testing Logic Programs* In: Proceedings of the Third Symposium on Programming Languages and Software Tools, ed. Mati Tombak, Kääriku Estonia August 23-24. (1993) 34–46
7. Kamkar M.: *Interprocedural Dynamic Slicing with Applications to Debugging and Testing* Linköping Studies in Science and Technology - Dissertations No. 297, Department of Computer and Information Science, Linköping University, 1993.
8. Kókai, G., Alexin, Z., Gyimóthy, T.: *Analyzing and Learning ECG Waveforms* In: Proc. of The Sixth International Workshop on Inductive Logic Programming (ILP'96)Stockholm, Sweden 28-30 August, 1996 152-171
9. Muggleton S. (ed.): *Inductive Logic Programming* Academic Press, 1992.
10. Ostrand, T. J., Balker, M. J.: *The Category-Partition Method for Specifying and Generating Functional Tests* In: CACM 31:6 June (1988) 676–686
11. Paakki, J., Gyimóthy, T., Horváth T.: *Effective Algorithmic Debugging for Inductive Logic Programming* In: Proc of the Fourth International Workshop on Inductive Logic Programming (ILP-94) Bad Honnef/Bonn Germany September 12-14 (1994) 175-194
12. Shapiro, E. Y.: *Algorithmic Program Debugging* MIT Press (1983)

4 Appendix

4.1 The compute program

```
program(L1,L2,C,S)      :- compute(L1,L2,C,S), print(C), print(S).
compute(L1,L2,C,S)      :-  sigma(L1,L2,S), complement(L1,L2,C).
complement(L,[],L).
complement(L,[H|T],U) :- member(H,L), delete(H,L,L1), complement(L1,T,U).
member(X,[X|T]).
member(X,[Y|T])         :- member(X,T).
delete(X,[X|T],T).
delete(X,[Y|Z],[X|U]) :- delete(X,Z,U).
sigma(L1,L2,S)          :- sum(L1,S1), sum(L2,S2), S is S1+S2.
sum([],0).
sum([X|T],S)            :- sum(T,S1), S is S1+X.
```

4.2 Test description for the compute program

```
type(member(+,+)).
type(delete(+,+,+)).
%       first category for member/2 : number_of_elements
 choice_of(member,2, number_of_elements, empty).
 choice_of(member,2, number_of_elements, one).
 choice_of(member,2, number_of_elements, more).
%       second category for member/2 : relation
 choice_of(member,2, relation, none).
 choice_of(member,2, relation, greater).
 choice_of(member,2, relation, less).
 choice_of(member,2, relation, equal).
%       first category for delete/3 : number_of_elements
 choice_of(delete,3, number_of_elements, empty).
 choice_of(delete,3, number_of_elements, one).
 choice_of(delete,3, number_of_elements, more).
%       second category for member/3 : membership
 choice_of(delete,3, relation, none).
 choice_of(delete,3, relation, greater).
 choice_of(delete,3, relation, less).
 choice_of(delete,3, relation, equal).
%       searching functions:
 empty(member,2,_,[]).
 one(member,2,_,X)                  :- length(X,L), L ==1.
 more(member,2,_,Y)                 :- length(Y,L), L >1.
 none(member, 2, _, []).
 greater(member,2,X,[Y|_])          :- X > Y.
 less(member,2,X,[Y|_])             :- X < Y.
 equal(member,2,X,[Y|_])            :- X = Y.
 empty(delete,3,_,[],_).
 one(delete,3,_,X,_)                :- length(X,L), L ==1.
 more(delete,3,_,X,_)               :- length(X,L), L >1.
 none(delete, 3, _,[],_).
 greater(delete, 3, X,[Y|_],_)      :- X > Y.
 less(delete, 3, X,[Y|_],_)         :- X < Y.
 equal(delete,3, X,[Y|_],_)         :- X = Y.
%       properties:
 p_empty(member, 2, empty, X)     :- member(empty, X).
 p_empty_del(delete, 3, empty, X) :- member(empty, X).
%       selectors expressions:
 cond(member, 2, none, X)     :- !, p_empty(member, 2, _, X).
 cond(member, 2, greater, X)  :- !, \+ p_empty(member, 2, _, X).
 cond(member, 2, less,    X)  :- !, \+ p_empty(member, 2, _, X).
 cond(member, 2, equal,   X)  :- !, \+ p_empty(member, 2, _, X).
 cond( delete, 3, none, X)    :- !, p_empty_del(delete, 3, _, X).
 cond( delete, 3, greater, X) :- !, \+ p_empty_del(delete, 3,_, X).
 cond( delete, 3, less,    X) :- !, \+ p_empty_del(delete, 3, _, X).
 cond( delete, 3, equal,   X) :- !, \+ p_empty_del(delete, 3,_, X).
```

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE *LIRA '97*
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1–4, 1997, PP. 111–119

A Technique for the Implicational Problem Resolving for Generalized Data Dependencies

Ivan Luković, Ph.D., Assist. Prof., Petar Hotomski, Ph.D., Full Prof.,
Biljana Radulović, M.Sc., Assist., Ivana Berković, M.Sc., Assist.

Abstract - *A possible approach to resolving the implicational problem for generalized data dependencies in relational data model is presented in the paper. The approach is based on the automatic reasoning method. By means of generalized dependencies the other dependency types, such as: functional, multivalued, join, implied and inclusion, can be expressed.*

1. Introduction

A mechanism for generalized representation of various data dependency types, such as functional (*fd*), multivalued (*mvd*), join (*jd*), implied (*imd*) and inclusion (*id*) dependencies, has been developed in the relational data model theory. The initial supposition was that all the data dependencies (i.e. "rules" which hold among data) can be represented, in unified manner, by one, or more symbolic data templates, satisfying certain criteria, according to defined interpretation rules for such symbolic templates. On the basis of that supposition, the term of generalized data dependency has been introduced, so as to represent the other dependency types in the same way. One of the important questions, arising when new data dependency type is introduced, is how can it be stated if a data dependency is a logical consequence of a given set of data dependencies. This problem in relational databases is known as the *implicational problem*.

At the beginning, the terms of: tableau, as a symbolic data template, generalized dependency (*gd*) and its interpretation are defined without explanations, because they are considered as already known. In the central part of the paper, it is presented a possible approach to resolving the implicational problem for *gd*s. By identifying the testing procedure for the implicational problem for *gd*s, it is established at the same time a way of testing the implicational problem for all the specific data dependency types which can be formalized by means of *gd*s. The proposed approach considers a usage of the Automatic Theorem Proving System (ATP), based on the first order predicate calculus and the resolution procedure with variable searching strategies (see [5]).

The reader is supposed to be familiar with fundamentals of the relational data model theory on the level of [1], [4] and [8] (particularly with the terms of *gd* and implicational problem) and the term of resolution procedure, on the level of [6].

2. Tableau and Generalized Data Dependency

Definition 1. Let an attribute set R, infinite but countable set of individual symbols - variables $Sym = \{x, y, z, w, x_1,..., x_k,..., x_1^l,..., x_m^l,...\}$ and a tuple (n-tuple) of symbols l defined by mapping l: $R \rightarrow Sym$ be given. *Tableau* over R, denoted by $\tau(R)$, is a finite set of tuples of symbols: $\tau(R) = \{l_i \mid l_i: R \rightarrow Sym\}$. □

Definition 2. Let $\mathcal{U}$ be the universal set of attributes, $Dom = \bigcup_{A \in \mathcal{U}} dom(A)$ be the union of all the domains of the attributes from $\mathcal{U}$, g: $Sym \rightarrow Dom$ be any function, representing a mapping from symbols to values and $\tau(R)$, $R \subseteq \mathcal{U}$, be a tableau.

- *Interpretation of a tuple of symbols* $l \in \tau$, denoted by $g(l)$, is a tuple of values, i.e. a function $g(l)$: $R \rightarrow Dom$, such that for each $A \in R$, $(g(l))(A) = g(l(A))$ holds.
- *Interpretation of a tableau* τ, denoted by $g(\tau)$, is such a relation over the set of attributes R, that $g(\tau) = \{g(l) \mid l \in \tau\}$. □

Like the term of the projection of a relation to an attribute set, the term of the *projection of a tableau* $\tau(R)$ to an attribute set X, $X \subseteq R$, denoted by $\pi_X(\tau)$, is introduced. Thus, $\pi_X(\tau)$ is the set of tuples $\pi_X(\tau) = \{l[X] \mid l \in \tau\}$, where $l[X]$ represents the restriction of the original tuple l to X.

In the following text, the term of *gd* is introduced. Each *gd* can be either a tuple dependency, or an equality dependency. By the next definition, a syntax of the tuple dependency is established. Then, its semantic is defined through the interpretation over a relation r. After that, the term of equality dependency and its interpretation over a relation are introduced, too.

Definition 3. Expression of the form $<\tau(R), \tau'(X)>$, $X \subseteq R \subseteq \mathcal{U}$, where $\tau(R) = \{l_i \mid i \in \{1,...,k\}\}$ and $\tau'(X) = \{l_j' \mid j \in \{1,...,m\}\}$ are tableaux, such that:

$$(\forall A_i \in X)(\forall l_j' \in \tau')(\exists B \in R)(l_j'[A_i] \in \pi_B(\tau))$$

holds, is a *tuple dependency* or *generalized T - dependency* (*tgd*). □

Let $<\tau(R), \tau'(X)>$ be an arbitrary *tgd*. If it is $|\tau'(X)| = 1$, then $<\tau(R), \tau'(X)>$ will be written in the form $<\tau(R), (x_1,...,x_m)(X)>$, i.e. $<\tau(R), l(X)>$, where $l = (x_1,..., x_m) \in \tau'(X)$. If the equality $X = R$ holds, then $<\tau(R), \tau'(R)>$ is called *full tgd* (*ftgd*). Oppositely, if it is $X \subset R$, such *tgd* is called *embedded tgd* (*etgd*).

Definition 4. Let the set of symbols *Sym* and the union of all domains *Dom* be given. A relation $r(R)$ *satisfies tgd* $<\tau(R), \tau'(X)>$, which is denoted by $r \models <\tau(R), \tau'(X)>$, if:

$$(\forall g: Sym \rightarrow Dom)(g(\tau) \subseteq r \Rightarrow g(\tau') \subseteq \pi_X(r)).$$ □

Definition 5. Expression of the form $<\tau(R), E>$, where $\tau(R) = \{l_i \mid i \in \{1,...,k\}\}$ is a tableau and $E = \{Eq^i(\lambda_p^i, \lambda_q^i) \mid i \in \{1,...,m\}\}$ is a finite set of "equality" predicates, such that:

$$(\forall i \in \{1,...,m\})(\exists A, B \in R)(\lambda_p^i \in \pi_A(\tau) \wedge \lambda_q^i \in \pi_B(\tau))$$

holds, is an *equality dependency* or *generalized E - dependency* (*egd*). □

If tableau symbols λ_p^i and λ_q^i of a predicate $Eq^i(\lambda_p^i, \lambda_q^i) \in E$ are used over the same attribute, then the predicate $Eq^i(\lambda_p^i, \lambda_q^i)$ is presented in the form $Eq^i_A(\lambda_p^i, \lambda_q^i)$, where A is the attribute, for which $\{\lambda_p^i, \lambda_q^i\} \subseteq \pi_A(\tau(R))$ holds. If for an *egd* $<\tau(R), E>$ $|E| = 1$ holds, then it will be written as $<\tau(R), Eq(\lambda_p^i, \lambda_q^i)>$, i.e. $<\tau(R), Eq_A(\lambda_p^i, \lambda_q^i)>$.

Definition 6. Let the set of symbols *Sym* and the union of all domains *Dom* be given. A relation $r(R)$ *satisfies egd* $<\tau(R), E>$, which is denoted by $r \models <\tau(R), E>$, if:

$$(\forall g: Sym \rightarrow Dom)(g(\tau) \subseteq r \Rightarrow (\forall Eq(\lambda_p^i, \lambda_q^i) \in E)(g(\lambda_p^i) = g(\lambda_q^i))). \square$$

Detailed information, concerning the terms of tableau, *gd* and ways of representing the other dependency types by means of *gd*s can be found in [1], [2], [3], [4], [7] and [8].

3. Implicational Problem for Generalized Dependencies

Definition 7. Generalized dependency γ is a *logical consequence* of the set of *gd*s Γ, defined over the attribute set R, $R \subseteq \mathcal{U}$, which is denoted by $\Gamma \models \gamma$, if:

$$(\forall r \in SAT(R))(r \models \Gamma \Rightarrow r \models \gamma)$$

holds, where $SAT(R)$ denotes the set of all the relations over R, and $r \models \Gamma$ denotes the fact that the relation r satisfies all the dependencies from Γ. $\square$

To resolve the implicational problem for a given set of *gd*s Γ and an arbitrary *gd* γ means to establish if $\Gamma \models \gamma$ holds. It is not practically possible to test the implicational problem $\Gamma \models \gamma$ by exact applying of Definition 7, i.e. by systematic generating of all the relations from $SAT(R)$ and checking the implication $r \models \Gamma \Rightarrow r \models \gamma$, because $SAT(R)$ is, in most cases, the set of high cardinality and it can be even infinite. Therefore, the other methods have to be applied so as to resolve the problem.

According to the nature of *gd*s, it is concluded that for the automation of the test $\Gamma \models \gamma$, the resolution procedure can be applied. Therefore, the set Γ and the dependency γ will be represented by appropriate predicate formulas.

4. Formalization of the Problem by Means of the First Order Predicate Calculus

In order to automate the condition $\Gamma \models \gamma$ checking, the ATP System has been used, which is based on the OL - resolution with marked literals [5]. For the sake of ATP applying, it is necessary to introduce an appropriate formalization for *tgd*s and *egd*s, which is based on predicate formulas.

So as to formalize a *gd*, a predicate representation of a tableau $\tau(R) = \{l_1,...,l_n\}$ should be introduced. The fact that the tuple of symbols (i.e. variables) belongs to $\tau(R)$ will be denoted by means of a predicate named P. Namely, the predicate formalization for $l = (\lambda_1,...,\lambda_k) \in \tau$ is $P(\lambda_1,...,\lambda_k)$, i.e. $P(l)$, where λ_i ($i \in \{1,..., k\}$) denote the predicate calculus variables, corresponding to those symbols from *Sym*, which have been used in τ. Hence, $\tau(R)$ is represented by the predicate formula:

$$P(l_1) \wedge ... \wedge P(l_n). \qquad (1)$$

Let γ be a *tgd* $<\tau(R), l(X)>$, $\tau = \{l_1,...,l_n\}$, $X \subseteq R$, $Sym(\gamma)$ be the set of all symbols, appearing in γ: $Sym(\gamma) = \{\lambda_1,..., \lambda_q\}$, and $l_e(R)$ be an extension of the tuple $l(X)$, defined in the following way:

$$(l_e(R))[X] = l(X) \wedge (\forall A \in R \setminus X)(l_e(A) \in Sym \setminus Sym(\gamma)).$$

According to Definition 4, i.e. the fact that *for each interpretation* g_i the implication $g_i(\tau(R)) \subseteq r \Rightarrow g(l) \in \pi_X(r)$ must hold, a formalization for $<\tau, l>$ is introduced by the following predicate formula $T(\gamma)$:

$$T(\gamma) : (\forall\lambda_1)(\forall\lambda_2)...(\forall\lambda_q)(\exists\lambda_{q+1})...(\exists\lambda_r)((P(l_1) \wedge ... \wedge P(l_n)) \Rightarrow P(l_e)), \qquad (2)$$

where $Var(T(\gamma)) = \{\lambda_1, \lambda_2,..., \lambda_r\}$ is the set of all distinct variables, appearing in $T(\gamma)$ and $Sym(<\tau, l_e>) \setminus Sym(\gamma) = \{\lambda_{q+1},..., \lambda_r\}$ is (possibly empty) set of symbols, used to make the extension $l_e(R)$ from $l(X)$.

For each *tgd* $<\tau(R), \tau'(X)>$ there is an equivalent set of *tgd*s of the form $<\tau(R), l(X)>$. Let $\tau'(X) = \{l_1',...,l_m'\}$. Logical equivalence of the sets $\{<\tau, l_1'>,...,<\tau, l_m'>\}$ and $\{<\tau(R), \tau'(X)\}$ directly follows from Definition 4 and Definition 7. Therefore, a *tgd* of the form $<\tau(R), \tau'(X)>$ is formalized by using the equivalent set of *tgd*s $\{<\tau, l_1'>,...,<\tau, l_m'>\}$.

Let γ be an *egd* $<\tau(R), Eq(\lambda_i, \lambda_j)>$, $\tau = \{l_1,...,l_n\}$. According to Definition 6, i.e. the fact that *for each interpretation* g_i, $g_i(\tau(R)) \subseteq r \Rightarrow g(\lambda_i) = g(\lambda_j)$ must hold, a formalization for $<\tau(R), Eq(\lambda_i, \lambda_j)>$ is introduced by the predicate formula $E(\gamma)$:

$$E(\gamma) : (\forall\lambda_1)(\forall\lambda_2)...(\forall\lambda_r)((P(l_1) \wedge ... \wedge P(l_n)) \Rightarrow \lambda_i = \lambda_j), \qquad (3)$$

where $Var(E(\gamma)) = \{\lambda_1, \lambda_2,..., \lambda_r\}$ is the set of all distinct variables, appearing in $E(\gamma)$. With respect to Definition 5, $\lambda_i, \lambda_j \in \{\lambda_1,..., \lambda_r\}$ must hold.

As well as for *tgds*, it can be proved that for each *egd* $<\tau(R), E>$ there is an equivalent set of *egd*s of the form $<\tau, Eq(\lambda_i, \lambda_j)>$, such that the formalization of $<\tau, E>$ is reduced to the formalization of the appropriate set of *egd*s, which includes a set of predicate formulas, given by (3).

If the resolution procedure does not allow the operating with equalities, the formula $E(\gamma)$ is transformed into the form:

$$E(\gamma) : (\forall\lambda_1)(\forall\lambda_2)...(\forall\lambda_r)((P(l_1) \wedge ... \wedge P(l_n)) \Rightarrow R(\lambda_i, \lambda_j)), \qquad (4)$$

where the predicate $R(\lambda_i, \lambda_j)$ means that λ_i and λ_j are equal and a new formula $R(\gamma)$ is introduced to represent the equality predicate $R(\lambda_x, \lambda_y)$:

$$R(\gamma) : (\forall\lambda_x)(\forall\lambda_y)(R(\lambda_x, \lambda_y) \Rightarrow (\forall\lambda_1)...(\forall\lambda_{k-1})(P(\lambda_1,..., \lambda_{i-1}, \lambda_x, \lambda_{i+1},..., \lambda_{k-1}) \Rightarrow P(\lambda_1,..., \lambda_{j-1}, \lambda_y, \lambda_{j+1},..., \lambda_{k-1}))), \qquad (5)$$

such that indexes i and j correspond to attributes A and B, for which $\lambda_x \in \pi_A(\tau)$ and $\lambda_y \in \pi_B(\tau)$ hold. The meaning of the formula $R(\gamma)$ is that if equality $R(\lambda_i, \lambda_j)$ holds, then each appearance of the variable λ_i is replaceable by λ_j and vice versa.

On the basis of Definitions 4, 6, 7 and previous considerations, it follows the conclusion formulated by the next theorem.

Theorem 1. Observe the set of *gd*s $\Gamma = \{\gamma_1,..., \gamma_n\}$, $n \geq 1$, and *gd* γ, such that all the *tgd*s and *egd*s are of the form $<\tau(R), I(X)>$ and $<\tau(R), Eq(\lambda_i, \lambda_j)>$, respectively. Let $\mathbf{F}(\Gamma) = \{F_1,...,F_n, F_{n+1},..., F_{n+k}\}$, $0 \leq k \leq |R|$, be the set of initial formulas - assumptions, formed on the basis of Γ, such that:

$$(\forall i \in \{1,...,n\})(F_i = \begin{cases} T(\gamma_i), & \gamma_i: \text{ is } tgd \\ E(\gamma_i), & \gamma_i: \text{ is } egd \end{cases})$$

holds. The next k formulas $F_{n+1},..., F_{n+k}$ are of the form $R(\gamma_i)$, given by (5). Let F be the formula which is built with respect to γ, where: $F = T(\gamma)$ if γ is *tgd*, or $F = E(\gamma)$ if γ is *egd*. Beside that, formulas $F_1,..., F_n$ and F satisfy the following condition:

$$(\forall F_i, F_j \in \mathbf{F}(\Gamma) \cup \{F\})(i \neq j \Leftrightarrow Var(F_i) \cap Var(F_j) = \varnothing).$$

The implication $\Gamma \models \gamma$ holds if and only if F logically follows from $\mathbf{F}(\Gamma)$. □

Finally, it should be stated that the formulas: $T(\gamma)$, given by (2), and $E(\gamma)$, given by (3), follow the same logic as the original definitions of *tgd* and *egd*, shown in [4].

5. Implicational Problem Testing by Automatic Reasoning Method

According to the resolution theorem and Theorem 1, the test of the condition $\Gamma \models \gamma$, where $\Gamma = \{\gamma_1,..., \gamma_n\}$, is performed by disproving procedure, on the basis of the set of initial assumptions $\mathbf{F}(\Gamma)$ and the negation of the conclusion $\neg F$. In that case, the theorem that should be proved by ATP System is of the form:

$$\mathbf{F}(\Gamma) \rightarrow F.$$

To prove the theorem, the clauses should be built from the set of assumptions $\mathbf{F}(\Gamma)$ and negation $\neg F$. They represent the input for ATP. Beside that, two additional input parameters are: (i) maximal searching deep and (ii) maximal clause length. With respect to the resolution theorem, there are three possible outcomes from ATP: (a) *"positive"*: an empty clause has been reached, which means that the assertion F holds. According to Theorem 1, $\Gamma \models \gamma$ holds, too; (b) *"negative"*: the empty clause has not been reached and there are no more possibilities for new clause generating. It means that the conclusion F cannot be derived from $\mathbf{F}(\Gamma)$. According to Theorem 1, we conclude $\Gamma \models \gamma$ does not hold; (c) *"uncertain"*: the empty clause has not been obtained, whereas maximal searching deep and maximal clause length have been reached, or memory resources have been exhausted.

Example 1. Let the implicational problem $\{A \rightarrow B, BC \rightarrow D\} \models AC \rightarrow D$ be given. It represents the pseudo-transitivity rule for *fd*s. *Fd*s $A \rightarrow B$, $BC \rightarrow D$ and $AC \rightarrow D$ are generalized by *egd*s $<\tau_1, Eq_B(y_1, y_2)>$, $<\tau_2, Eq_D(w_3, w_4)>$ and $<\tau_3, Eq_D(w_5, w_6)>$, respectively, where tableaux τ_1, τ_2 and τ_3 are of Fig. 1.

τ_1	A	B	C	D
	x_1	y_1	z_1	w_1
	x_1	y_2	z_2	w_2

τ_2	A	B	C	D
	x_3	y_3	z_3	w_3
	x_4	y_3	z_3	w_4

τ_3	A	B	C	D
	x_5	y_5	z_5	w_5
	x_5	y_6	z_5	w_6

Fig. 1.

Predicate formalization of the mentioned *fd*s is $\mathbf{F}(\Gamma) = \{F_1, F_2, F_3, F_4\}$ and F, where:

- $F_1 = E(A \to B)$:
  $(\forall x_1)(\forall y_1)(\forall z_1)(\forall w_1)(\forall y_2)(\forall z_2)(\forall w_2)(P(x_1, y_1, z_1, w_1) \wedge P(x_1, y_2, z_2, w_2) \Rightarrow R(y_1, y_2))$
- $F_2 = E(BC \to D)$:
  $(\forall x_3)(\forall y_3)(\forall z_3)(\forall w_3)(\forall x_4)(\forall w_4)(P(x_3, y_3, z_3, w_3) \wedge P(x_4, y_3, z_3, w_4) \Rightarrow R(w_3, w_4))$
- $F = E(AC \to D)$:
  $(\forall x_5)(\forall y_5)(\forall z_5)(\forall w_5)(\forall y_6)(\forall w_6)(P(x_5, y_5, z_5, w_5) \wedge P(x_5, y_6, z_5, w_6) \Rightarrow R(w_5, w_6))$
- $F_3 = R(A \to B)$:
  $(\forall y_7)(\forall y_8)(R(y_7, y_8) \Rightarrow (\forall x_7)(\forall z_7)(\forall w_7)(P(x_7, y_7, z_7, w_7) \Rightarrow P(x_7, y_8, z_7, w_7)))$
- $F_4 = R(AC \to D) = R(BC \to D)$:
  $(\forall w_8)(\forall w_9)(R(w_8, w_9) \Rightarrow (\forall x_9)(\forall y_9)(\forall z_9)(P(x_9, y_9, z_9, w_8) \Rightarrow P(x_9, y_9, z_9, w_9)))$.

On the basis of F_1, F_2, F_3, F_4 and $\neg F$, the following clauses are formed:

1. $\neg P(x_1, y_1, z_1, w_1) \vee \neg P(x_1, y_2, z_2, w_2) \vee R(y_1, y_2)$
2. $\neg P(x_3, y_3, z_3, w_3) \vee \neg P(x_4, y_3, z_3, w_4) \vee R(w_3, w_4)$
3. $P(a_1, b_1, c_1, d_1)$
4. $P(a_1, b_2, c_1, d_2)$
5. $\neg R(d_1, d_2)$
6. $\neg R(y_7, y_8) \vee \neg P(x_7, y_7, z_7, w_7) \vee P(x_7, y_8, z_7, w_7)$
7. $\neg R(w_8, w_9) \vee \neg P(x_9, y_9, z_9, w_8) \vee P(x_9, y_9, z_9, w_9)$.

With respect to the fact that universal quantifiers are inverted into the existential ones by negating of the formula F, the variables x_5, y_5, z_5, w_5, y_6 and w_6 have been transformed into the corresponding *Skolem constants* a_1, b_1, c_1, d_1, b_2 and d_2.

ATP produces the positive answer. It follows that *fd* $AC \to D$ is a consequence of Γ. In the following text, the extract from the resolution procedure which leads to the empty clause is shown, instead of the real output from ATP which is more complex and less readable. The notation "(m, n) & (k, l)", at the beginning of each clause, means that the clause is obtained by resolving nth literal of mth clause with lth literal of kth clause.

8. (1, 1) & (3, 1): $\neg P(a_1, y_2, z_2, w_2) \vee R(b_1, y_2)$
9. (8, 1) & (4, 1): $R(b_1, b_2)$
10. (9, 1) & (6, 1): $\neg P(x_7, b_1, z_7, w_7) \vee P(x_7, b_2, z_7, w_7)$
11. (10, 1) & (3, 1): $P(a_1, b_2, c_1, d_1)$
12. (11, 1) & (2, 1): $\neg P(x_4, b_2, c_1, w_4) \vee R(d_1, w_4)$
13. (12, 1) & (4, 1): $R(d_1, d_2)$
14. (13, 1) & (5, 1): - (*Empty clause*). □

Example 2. By the implicational problem $\{[A,B] \subseteq [B,C], [B,C] \subseteq [C,D]\} \models [A,B] \subseteq [C,D]$ transitivity rule for *id*s is represented. We generalize previously mentioned *id*s by *tgd*s defined over $R = ABCD$: $<(x_1, y_1, z_1, w_1)(R), (x_1, y_1)(BC)>$, $<(x_1, y_1, z_1, w_1)(R), (y_1, z_1)(CD)>$ and $<(x_1, y_1, z_1, w_1), (x_1, y_1)(CD)>$ and obtain:

- $F_1 = T([A, B] \subseteq [B, C])$:
  $(\forall x_1)(\forall y_1)(\forall z_1)(\forall w_1)(\exists x_2)(\exists w_2)(P(x_1, y_1, z_1, w_1) \Rightarrow P(x_2, x_1, y_1, w_2))$
- $F_2 = T([B, C] \subseteq [C, D])$:
  $(\forall x_3)(\forall y_3)(\forall z_3)(\forall w_3)(\exists x_4)(\exists y_4)(P(x_3, y_3, z_3, w_3) \Rightarrow P(x_4, y_4, y_3, z_3))$
- $\neg F = \neg T([A,B] \subseteq [C,D])$:
  $(\exists x_5)(\exists y_5)(\exists z_5)(\exists w_5)(\forall x_6)(\forall y_6)(P(x_5, y_5, z_5, w_5) \wedge \neg P(x_6, y_6, x_5, y_5))$.

The following clauses (where f_1, f_2, f_3, f_4 are Skolem functions and a_1, b_1, c_1, d_1 are Skolem constants) are inferred from F_1, F_2 and $\neg F$:

1. $\neg P(x_1, y_1, z_1, w_1) \vee P(f_1(x_1, y_1, z_1, w_1), x_1, y_1, f_2(x_1, y_1, z_1, w_1))$
2. $\neg P(x_3, y_3, z_3, w_3) \vee P(f_3(x_3, y_3, z_3, w_3), f_4(x_3, y_3, z_3, w_3), y_3, z_3)$
3. $P(a_1, b_1, c_1, d_1)$
4. $\neg P(x_6, y_6, a_1, b_1)$.

ATP also generates an empty clause, so the implicational problem is positive:

5. (3, 1) & (1, 1): $P(f_1(a_1, b_1, c_1, d_1), a_1, b_1, f_2(a_1, b_1, c_1, d_1))$
6. (5, 1) & (2, 1): $P(f_3(f_1(a_1, b_1, c_1, d_1), a_1, b_1, f_2(a_1, b_1, c_1, d_1)),$
   $f_4(f_1(a_1, b_1, c_1, d_1), a_1, b_1, f_2(a_1, b_1, c_1, d_1)), a_1, b_1)$
7. (6, 1) & (4, 1) - (*Empty clause*). □

Example 3. For the implicational problem $\{\triangleright\triangleleft(AB, AC), A \rightarrow C\} \models C \rightarrow B$, which has a negative solution, ATP can produce the negative answer (it does not generate the empty clause). The predicate formulas are:

- $F_1 = T(\triangleright\triangleleft(AB, AC))$:
  $(\forall x_1)(\forall y_1)(\forall z_1)(\forall y_2)(\forall z_2)(P(x_1, y_1, z_1) \wedge P(x_1, y_2, z_2) \Rightarrow P(x_1, y_1, z_2))$
- $F_2 = E(A \rightarrow C)$:
  $(\forall x_3)(\forall y_3)(\forall z_3)(\forall y_4)(\forall z_4)(P(x_3, y_3, z_3) \wedge P(x_3, y_4, z_4) \Rightarrow z_3 = z_4)$
- $F = E(C \rightarrow B)$:
  $(\forall x_5)(\forall y_5)(\forall z_5)(\forall x_6)(\forall y_6)(P(x_5, y_5, z_5) \wedge P(x_6, y_6, z_5) \Rightarrow y_5 = y_6)$. □

Example 4. Observe the implicational problem: $\{A \rightarrow C, B \rightarrow C\} \models \{(A, B)\} \rightarrow C$, which represents so called chaining rule for *imd*s. ATP gives the positive answer, where:

- $F_1 = E(A \rightarrow C)$:
  $(\forall x_1)(\forall y_1)(\forall z_1)(\forall y_2)(\forall z_2)(P(x_1, y_1, z_1) \wedge P(x_1, y_2, z_2) \Rightarrow z_1 = z_2)$
- $F_2 = E(B \rightarrow C)$:
  $(\forall x_3)(\forall y_3)(\forall z_3)(\forall x_4)(\forall z_4)(P(x_3, y_3, z_3) \wedge P(x_4, y_3, z_4) \Rightarrow z_3 = z_4)$
- $F = E(\{(A, B)\} \rightarrow C)$:
  $(\forall x_5)(\forall y_5)(\forall z_5)(\forall y_6)(\forall z_6)(\forall z_7)(P(x_5, y_5, z_5) \wedge P(x_5, y_6, z_6) \wedge P(x_6, y_6, z_7) \Rightarrow z_5 = z_7)$.
  □

6. Conclusion

It appears the implicational problem for *gd*s can be tested relatively easy by using the automatic reasoning method, i.e. by the resolution procedure. Therefore, the first order predicate calculus formalization of the problem is introduced and the experiments are performed, by using the ATP System, which is developed on Technical Faculty in Zrenjanin. An alternative and similar technique to this one is *Chase* algorithm. There are two reasons, why we decided to use the automatic reasoning method versus *Chase*: (i) the existence of ATP System, which is based on the resolution procedure and (ii) nature of *gd*s, i.e. the fact that the structure and interpretation of a *gd* can be formalized by such a predicate formula, for which we believe it is adequately readable. Nevertheless which of the techniques we use, there are two problems: exponential complexity of the techniques and general undecidability of the problem.

As it concerns the complexity, one of the ways to improve that, is to reduce the problem on the specific dependency type (if it is possible), such as *egd*s, or commonly used *fd*s, for which there is a polynomial membership algorithm. However, our aim was to consider *gd*s as a unified manner of representing the other, various dependency types. There are also possibilities to improve average complexity of the resolution procedure in the cases when the equality predicate $R(\lambda_x, \lambda_y)$ is used. One of them concerns renaming of $R(\lambda_x, \lambda_y)$ into $R_i(\lambda_x, \lambda_y)$ (i.e. its indexing), in the case where λ_x and λ_y have been used over the same attribute A_i in the predicate Eq: $Eq_{A_i}(\lambda_x, \lambda_y)$.

In those situations when our implicational problem is decidable (it is always the case for full *tgd*s and *egd*s), there are several techniques by implementation of ATP can be "forced" to behave decidably and avoid uncertain outcomes. However, the uncertainty appears for the implicational problems (nevertheless the answer is positive or negative) that include both embedded *tgd*s and *egd*s, such that there is an interaction in the resolution procedure of equality predicates and Skolem functions, which come from existential quantifiers. This interaction requires introducing of equality axioms predicates (reflexivity, symmetry and transitivity), which makes the complexity worse.

Practical usefulness of described technique is limited to theoretical purposes and smaller examples due to exponential complexity of the problem (but it is the case for *Chase*, too) and the fact that *gd*s are converted into the clauses by hand. One of the next steps is to build a translator for converting the set of *gd*s into the set of clauses for ATP. As it concerns the design of commercial database schemas, the techniques based on resolution procedure or *Chase* are generally impractical. On the other hand, the implicational problem in such situations is bounded to *fd*s or it is even avoided by applying the conceptual design techniques, such as entity-relationship diagrams.

7. References

[1] Ullman D. J, *Principles of Database Systems*, Computer Science Press, Inc. Rockville, Maryland, 1982.

[2] Sadri F, Ullman J. D, *"Template Dependencies: A Large Class of Dependencies in Relational Databases and Its Complete Axiomatization"*, Jour. of ACM, Vol. 29, No. 2, April 1982, pp. 363-372.

[3] Hull R, *"Finitely Specifiable Implicational Dependency Families"*, Journal of the ACM, Vol. 31, No. 2, April 1984, pp. 210-226.

[4] Paredaens J., De Bra P., Gyssens M., Van Gucht D., *The Structure of the Relational Database Model*, Springer-Verlag, Berlin Heidelberg, 1989.

[5] Berković I, *"Variable Searching Strategies in The Educationally Oriented System for Automatic Theorem Proving"*, M.Sc. Thesis, Technical Faculty "Mihajlo Pupin", Zrenjanin, 1994. (in Serbian)

[6] Hotomski P, *"Artificial Intelligence Systems"*, Technical Faculty "Mihajlo Pupin", Zrenjanin, 1995. (in Serbian)

[7] Luković I, *"Integration of The Information System Database Module Schemas"*, Ph.D. Thesis, Faculty of Technical Science, N. Sad, 18. 01. 1996. (in Serbian)

[8] Mogin P, Luković I, *"Database Principles"*, University of Novi Sad, Faculty of Technical Science and STYLOS Novi Sad, April 1996. (in Serbian)

[9] Luković I, Hotomski P, Radulović B, Berković I, *"A Proof of Generalized Data Dependency Satisfaction by The Automatic Reasoning Method"*, II Symposium on Comp.Sc. and Informatics YUINFO, Brezovica, 02-05. 04. 1996. (in Serbian)

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 121–127

Pruning Nodes in Feedforward Neural Networks

Ana Madevska, Katerina Zdravkova
Institute of Informatics
Faculty of Natural Science and Mathematics
P.O. box 162, 91000 Skopje, Macedonia
ana@robig.pmf.ukim.edu.mk
keti@robig.pmf.ukim.edu.mk

Abstract: *Multi-layered feedforward neural networks are highly parallel processing elements, with each node contributing to the final output response. The problem of pruning nodes in the network is studied, in order to determine whether the process of pruning degrades the network performances. An alternative approach to the process of pruning network nodes is suggested, as well as a novel way of reconstructing the degraded network.*

1. Introduction

Many problems, such as signal processing, natural language processing (Cundeva, 1993), meet the problem of controlled pruning nodes. The most important objective is the improvement of network generalization ability. There are various algorithms for pruning weights or nodes of a trained neural network. Here, the problem of induced, or controlled pruning of input nodes is examined and studied. Unlike other pruning algorithms that interfere the learning algorithm itself, the suggested method operates within the training set (input/output pattern pairs). This method disconnects and reconnects the input nodes of the network. The performance degradation rate of the trained network, when certain number of input nodes were pruned, was also proposed and investigated.

2. Theoretical Foundation

The term *zero-pruning of a node* i is introduced. During each network training iteration, when the weights are adjusted according to the backpropagated error signals (Rumelhart *et al.*, 1986), all weights connecting the input node i and the hidden nodes get zero value. When the training process is terminated, those weights remain with zero value. Nevertheless, the network works properly, without the zero-valued weights participating in modeling the network output. The i-th input node can be pruned, without disturbing network performance.

In practice, to achieve the zero-pruning of a node the following algorithm is used:

Algorithm for zero-pruning

- i-th input node is chosen for zero-pruning;
- the learning algorithm is modified: after the weight adjustment in each iteration, the following cycle is added:

```
j=1
repeat
    w_ij=0;
    j=j+1;
until j>N_hid. ■
```

This modification of the learning algorithm has an O(n) complexity, where n is the number of hidden nodes in the network.

The zero-pruning algorithm can be confirmed by analyzing the input signals of the j-th hidden node according to BP algorithm (Rumelhart *et al.*, 1986):

$$(1)\quad net_j = \sum_k w_{kj} \cdot o_k = \sum_{k \neq i} w_{kj} \cdot o_k \quad , j=1,\ldots,N_{hid}$$

because w_{ij} has a zero value.

Compared to the network trained without the above modification, zero-pruning affects the internal representation of the network, while the learning process remains equally fast.

An important parameter for comparing two networks trained with same initial seed is the error function E (2).

$$(2)\qquad E = \frac{1}{2} \sum_i (t_i - o_i)^2$$

The following example (Fig. 1) shows that the number of required iterations for training the network with 4:3:4 topology (4 input, 3 hidden and 4 output nodes) (Madevska, 1996), used as a heteroassociative memory, is the same in both cases.

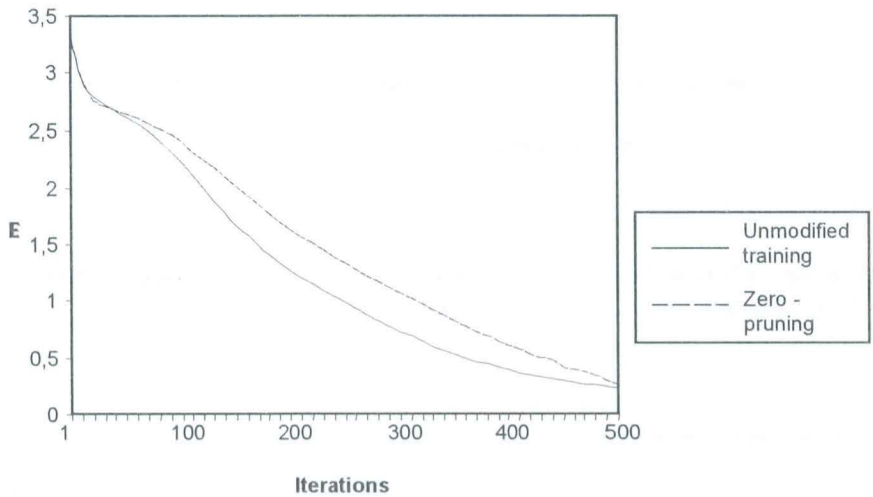

Fig. 1. Training the network without modification and with zero-pruning

3. Induced Pruning

The idea of setting certain weights to zero and pruning the associated input nodes, is used for developing a novel algorithm for pruning input nodes in the feedforward neural network, through the input training set. In this case, the objective is to prune input nodes after completing the training process, without zero-pruning. The advantage of this approach is that there is no need for an intervention in the learning algorithm, i.e. the weights are manipulated *from outside* (Madevska, 1996).

The expression *modified training set* is used: the i-th coordinates of all input patterns get inhibitory activation, while output patterns remain unchanged.

The method of induced pruning can be realized with the following algorithm:

Algorithm for pruning the input node i

- network is trained to the certain value of the error function E;
- training set is modified;
- network is retrained with the modified training set;
- i-th input node is pruned. ∎

A theoretical explanation of this algorithm can be found in the following expression (net input of the j-th hidden node):

(3) $$net_j = \sum_k w_{kj} \cdot o_k = \sum_{k \neq i} w_{kj} \cdot o_k ,$$

because o_i, the i-th component of the input patterns, has a zero value.

Weight w_{ij} is modified by the formula:

(4) $w_{ij}(t + 1) = w_{ij}(t) + \eta \delta_j o_i$.

Therefore, when patterns of the modified training set are presented during the retraining process, weight w_{ij} remains unchanged:

(5) $$w_{ij}(t + 1) = w_{ij}(t), \qquad |\Delta w_{ij}| = 0 .$$

This process is repeated over *all* hidden nodes.

During the retraining process, all weights between the i-th input node and the hidden layer remain unchanged, so pruned nodes no longer affect the modeling of error function E. Therefore, pruning of a specified node i will not decrease network performance.

The efficiency of this algorithm strongly depends on the retraining process. The number of required iterations should not be close to the number of training iterations. By analyzing this problem, it was concluded that the number of retraining iterations is significantly less than the number of training iterations. If the relevance (Smolensky, 1986) of the node i has insignificant value, its pruning does not increase the value of the error function E. Hence, the modified input pattern set will be recognized and no retraining is necessary.

The relevance of the input node can be computed in an alternative way, which does not require modification of the learning algorithm. The advantage of this approach is the opportunity of operating via the modified training set:

Algorithm for computing the relevance of the i-th input node

- network is trained to the certain value of the error function, E_{min};
- training set is modified;
- the error function is computed for the presented modified training set, E_i;
- the difference $\rho_i = | E_{min} - E_i |$ is the *relevance* of the input node i. ∎

The only additional computation in this method for calculating the relevance, is determination of the error function value E_i, i.e. when the modified input pattern set is presented.

In the cases where the relevance of the pruned node has a notable value (bigger than 0.4), the network has to be retrained with the modified training set; however, the

number of retraining iterations is significantly smaller than the training iterations. The damage done by this kind of pruning upsets the network, but it is quite likely to move into a state that has a large gradient toward the correct solution, so when retraining occurs, the network moves along this steep gradient and quickly recovers the solution (Beale, 1992). This is due to similarity of the training set and the modified training set (Rumelhart, 1986).

The benefit of the suggested pruning method of input nodes is also in the ability to reconnect the pruned nodes, which were not actually removed. If needed, the network is retrained with the initial training set. This process does not require a large amount of iterations, for the same reasons as in the pruning case. All initially learned patterns were not lost from memory.

4. Simulation Results

All neural networks used in the following simulations were fully connected three-layered networks with distinct topologies, trained with binary input/output pairs. The proposed pruning algorithm has been tested and confirmed by a number of experiments, including pruning more than one input node. Here, the modification of the training set was extended by setting all corresponding input components with inhibitory activation.

If the pruning of input nodes causes removal of distinctive input components, then no retraining is possible. The network falls into a meta-stable position, because different outputs correspond to the same input pattern.

Network with topology 10:6:8, used as a heteroassociative memory, was trained with the given training set within 1000 iterations until an error function value of E=0.09. Fig. 2. shows the required number for retraining in the worst case when different combinations of input nodes were pruned.

Number of iterations for E=0.09	**10:6:8** Number of iterations for retraining the network when pruning the input nodes (worst case)						
	number of pruned input nodes						
	1	**2**	**3**	**4**	**5**	**6**	**7**
1000	10	50	50	50	50	50	50

Fig. 2. Network 10:6:8

The achievement and potential of these results motivated another, broader range of experiments, in which the network topology and training sets from the Nettalk project (Sejnowski, 1987) were used.

The training set consisting of 200 patterns was used to train the network with topology 203:108:26. Results show that networks with immense topology and bigger training set, also give good results with pruning nodes. The larger numbers of network weights, the more retraining iterations are needed. Fig. 3. presents the results of training the network 203:108:26 within 400 iterations (E=15.27).

Value of the error function E for 400 iterations	**203:108:26** Number of iterations for retraining the network when pruning the input nodes (worst case)			
	number of pruned input nodes			
E=15.27	20	40	60	80
	60	80	110	190

Fig. 3. Network 203:108:26

The Nettalk network has difficulties with learning (E=15.27 when 400 iterations are done). The results can be significantly improved if simple scaling of the input pattern training set is applied, i.e. instead of using the default neutral and excitatory activation values, polar activations (inhibitory and excitatory) are used (Madevska, 1996). Furthermore, the number of retraining iterations is smaller. Table 3 shows these results.

Value of the error function E for 400 iterations	**203:108:26, polar activations** Number of iterations for retraining the network when pruning the input nodes (worst case)			
	number of pruned input nodes			
E=4.22	20	40	60	80
	20	30	40	60

Fig. 4. Network 203:108:26, polar activations

5. Conclusion

The theoretical and experimental results described in this paper clearly indicate the advantages of the proposed pruning algorithm – the nodes are pruned by the modified training set, instead of intervening in the learning algorithm. Also, the pruned nodes can be reconnected in the network, by retraining with the initial training set. It was shown that the process of retraining is either needless or inexpensive. Future research would concentrate on investigating the possibilities of pruning hidden nodes via the training set.

6. References

1. Cundeva, K. (1993), "Learning in Example Based Machine", Proceedings of 15th International Conference on Information Technology Interface, Pula, 15-18 June, 1993, pp. 146-153.
2. Gill, P., Murray, W. and Wright, M. H. (1988), *Practical Optimization*, Academic Press Limited, London.
3. Hertz, J., Krogh, A. and Palmer, R. G. (1991), *Introduction to the theory of Neural Computation*, Addison-Wesley Publishing Company.
4. Krogh, A. and Hertz, J. (1995), "A Simple Weight Decay can improve generalization", Advances in NIPS, J. E. Moody, S. J. Hanson, R. P. Lippmann (eds.), Morgan-Kaufmann, pp. 950-957.
5. Madevska, A. (1996), "Modification of multi-layered feedforward neural networks", M.Sc. thesis, Institute of Informatics, Skopje.
6. Ripley, B. D. (1996), *Pattern Recognition and Neural Networks*, Cambridge University Press.
7. Rumelhart, D. E. and McClelland, J. L. (1986), *Parallel Distributed Processing, Vol.1 - Foundations*, The MIT Press-USA.
8. Sejnowski, T. J. and Rosenberg, C. R. (1987), "Parallel networks that learn to pronounce English text", Complex Systems, 1, 145-168.

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE *LIRA '97*
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1 - 4, 1997., PP. 129-138

Implementing Lazy Foreign Functions in a Procedural Programming Language

Dragan Maćoš
Humboldt Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany,
e-mail: macos@informatik.hu-berlin.de

Zoran Budimac
Faculty of Science, Institute of Mathematics,
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: zjb@unsim.ns.ac.yu

Abstract

Foreign functions could be described as a possibility for users to extend a functional language on the implementation level. SK-graph reduction is a technique for implementing functional languages which provides a „natural" realization of non-strict language semantics. In this paper we show an extension of the SK-graph reduction system with foreign functions in the (eager) procedural language Modula-2 and discuss the possibilities of delaying (forcing) their evaluation with the goal of compatibility with the lazy SK-system. Thus, we make the SK-system faster and utilize foreign functions. The main contribution of the paper is to provide lazy evaluation in Modula-2.

We analyze two ways of delaying the evaluation of foreign functions: a) calling the SK-evaluator which stops the evaluation if an expression in weak head normal form (WHNF) is on the top of the spine stack and b) implementing foreign functions without calling the graph reducer. The second approach is based on adding data structures and functions for delaying and forcing evaluation.

1 Introduction

There exist many programming styles and languages but none provide the generality for solving any kind of problem. For that reasons, the possibilities of connecting different program paradigms are investigated.

Foreign functions are functions written in a programming language that is different from the language in which the main program is written. Foreign functions written in procedural

programming languages could be described as a bridge between functional and procedural programming paradigms. They are a good possibility to improve an implementation for the following reasons:

- A functional language could be extended with a library of functions that have already been implemented in a procedural language.
- Programs written in functional languages execute sometimes much slower than programs written in procedural languages. By realizing the „critical" parts of a functional program in a procedural language the execution of functional program could be made more efficient.
- With the definition of a foreign function, one can extend the language without changing the implementation.

In this paper, we give a short description how to extend an SK-graph reduction based implementation of a functional language with foreign functions. We concentrate on the implementation of *lazy* foreign functions in a procedural programming language. By the term "lazy functions" we mean the functions whose evaluation can be delayed and therefore that can operate with infinite data structures. Since procedural programs are always "eager" to evaluate everything they can in advance, the implementation of lazy functions in a procedural language is a challenging task. Throughout the paper we use Modula-2 as a suitable representative of procedural programming languages.

2 SK-combinator Graph Reduction

First we define combinators and give a short description of the SK-graph reduction.

2.1 Combinators

The combinators are λ-expressions without free variables. The following λ-abstractions are for example combinators (*K* and *S*):

$K \equiv \lambda xy.x$
$S \equiv \lambda fgx.\ fx\ (gx)$

In practice we define a larger set of combinators to implement a functional language. This technique is called SK because every combinator could be expressed as a combination of S and K combinators.

The implementation of a functional language consists of compiling the functional language into a combinator term and its reduction into a weak head normal form (WHNF.) The compilation process is beyond the scope of this paper (see for example tutorial texts in [2,3,5] for more details.) Let us now describe the SK-reducer (abstract machine).

2.2 SK-abstract machine

The SK-abstract machine reduces the combinator term into its WHNF. The combinator term in WHNF is actually the result of the evaluation of functional program. The components of the SK-reducer are:

- combinator graph, that has to be reduced,
- "spine stack" whose top points to the current combinator while the elements below the top point to combinator "arguments."

The reduction process consists of the following steps:

- unwinding the spine stack i.e., searching for the combinator to be reduced and pointing to all of its arguments on the way,
- reduction, i.e., replacing the root of the reduced expression with the result of evaluation

Both steps are repeated until the top of the spine stack reaches the WHNF.

Next we describe the possibility of extending the implementation of SK-abstract machine with defining foreign functions.

2.3 Implementation of Foreign Functions in the SK-reduction Model

For the implementation of foreign functions, we have to define following:

- A new combinator corresponding to the calls of foreign functions (named *Foreign*),
- the compilation rule for the translating the calls of foreign functions into a combinator term (with combinator *Foreign*),
- reduction rules for the combinator *Foreign*.

Let us show how the calls of the foreign functions are translated into a combinator term and how the combinator *Foreign* is applied.

2.4 Translation of the Foreign Function Calls

Let the call of a foreign function start with "_" for unique naming purposes. Thus, every call of some foreign function has the following form:

$$(\_FunName\ A_1\ A_2 \ldots A_n)$$

where *FunName* is the name of the implemented foreign function and $A_1\ A_2 \ldots A_n$ are the arguments of the function (we use the lisp-notation for the function applications.) The combinator term that represents the call of foreign function has to contain following components:

- combinator *Foreign*,
- the name of the called foreign function,
- arguments of the foreign function.

We have chose the following form of the combinator form:

$$(Foreign\ FunName'\ (A'_1\ A'_2 \ldots A'_n))$$

where $FunName'$, A'_i, $i = 1,\ldots,n$ are translations of $FunName$, A_i, $i = 1,\ldots,n$, respectively. In the following section we describe the reduction rule for the combinator *Foreign*.

2.5 Reduction Rule for the Combinator *Foreign*

The communication between the SK-reduction machine and foreign functions is implemented through the global variables **Argument** and **Result** [1,4]. SK-abstract machine is implemented as a coroutine that transfers control to the foreign function explicitly, passing the arguments through the global **Argument**. Every foreign function is implemented as well as a coroutine that explicitly passes control back to the SK-machine, passing the result of its evaluation through the global **Result**.

The reduction of the combinator *Foreign* consists of the following steps:

- evaluation of the arguments of the foreign function,
- evaluation of the name of the foreign function,
- call of the appropriate foreign function,
- updating the root of the combinator term that has been evaluated with the value of the variable **Result**.

What follows is the Modula-2 procedure for the reduction of the combinator *Foreign*.

```
PROCEDURE RedForeign(s : SExp);
VAR Ind : CARDINAL;
BEGIN
    Argument := Eval(List_of_Arguments);
    FunctionName := Eval(FunctionName);
    Ind := Find(FunctionName);
    (* finds address of the foreign function *)
    TRANSFER(ExecAdr, MF.Address[Ind])
    (* foreign function call *)
    Update(s, Result)
    (* Updating root of graph with result of foreign function
       evaluation *)
END RedForeign;
```

The function **Eval** is the SK-reducer.

We have so far described how to extend the SK-evaluation model with foreign functions. In the following section, we discuss how to define *lazy* foreign functions.

3 Delaying the Evaluation of Foreign Functions

In this section we give two possibilities of delaying the evaluation of foreign functions. The first possibility is based on the SK-machine calls whose evaluating strategy is lazy. The second one is based on abstract data structures implemented in a procedural language. As an example we discuss the processing of infinite lists.

3.1 Delaying the evaluation with SK-machine calls

As mentioned earlier, the evaluation of foreign functions could be delayed by calling the SK-machine simulator, which is lazy. For example, if we want to operate with an infinite list, we can force the evaluation of elements of the list one by one because the SK-evaluator stops the evaluation if the current node to be reduced is a list-constructor.

Some procedures for operations on infinite lists follow. SK combinator term is represented as an s-expression (abstract data type `SExp`) over which several operations have been defined (`Atom`, `Eq`, `Head`, `Tail`, ...) with obvious meaning.

```
PROCEDURE LazyEqual(E, L : SExp) : BOOLEAN;
VAR
   hE, hL, tE, tL: SExp;
BEGIN
   IF Atom(E) OR Atom(L) THEN
      RETURN Eq(E, L)
   ELSE
      tL := Tail(L); hL := Head(L);
      tE := Tail(E); hE := Head(E);
      RETURN LazyEqual(Eval (hE), Eval(hL))) AND (*)
             LazyEqual(Eval(tE), Eval(tL)) (*)
   END
END LazyEqual;

PROCEDURE Memb(E, L : SExp): BOOLEN;
VAR hL, tL: SExp;
BEGIN
   IF Eq(L, Null) THEN
      RETURN FALSE
   ELSE
      hL := head(L);
      IF LazyEqual(Eval(E), Eval(hL)) THEN (*)
```

```
            RETURN TRUE
        ELSE
            tL := tail(L);
            RETURN Memb(E, Eval(tL)) (*)
        END
    END
END
```

Let us now analyze the procedures **LazyEqual** and **Memb**. The arguments of these procedures (lists) are unevaluated. When we need an element of the list, its evaluation has to be forced. The program lines forcing the evaluation of the list are designated with (*). **LazyEqual(Eval (hE), Eval(hL))** means "Evaluate the head of the list E, then evaluate the head of the list L and examine if they are equal." If the "heads" of the lists are not atomic the evaluation in SK-machine is going to be stopped. Through recursive calls of **LazyEqual** and calls of the SK-machine (**Eval**) we force the evaluation step by step. The next procedure shows the foreign function **Member** that calls the defined functions.

```
PROCEDURE Member;
VAR a1, a2: SExp;
BEGIN
LOOP
    a1 := Head(Argument );
    a2 := Head(Tail(Argument));
    IF Memb(a1, a2) THEN
        Result := Quote("T");
    ELSE
        Result := Quote("F")
    END
    TRANSFER(MF.Address[1], Execute)
END
END Member;
```

This model for delaying the execution is quite expensive. In our example, we had to call the SK-executor to evaluate every element of the list. That means, for every list element we had to unwind the spine stack, evaluate the element of the list, update the root with the result of the evaluation. We now show how to delay the execution of foreign functions without calling the SK-machine.

3.2 Delaying the Evaluation Without an Abstract Machine

In the previous section, the lazy evaluation of the lists is based on the laziness of the SK-machine. Now, we don't want to call the SK-machine but implement the whole mechanism directly in a procedural language. We first define the base data structures and functions for processing infinite

lists.

The main idea is to save the parameters for the creation of a list element instead of evaluating the whole list. When we need some unevaluated element, we can simply evaluate it on the fly. Elements of infinite lists are represented as nodes which contain the function and its arguments for evaluating the next list element. The next several procedures show the definition of an infinite list node (function **NewLazyNode**), infinite list (function **LazyCons**) and the evaluation of the list head and tail (**LazyHead** and **LazyTail**), respectively.

```
PROCEDURE NewLazyNode(Fun: FunTypePtr; Arg: SExp): SExp;
VAR Hlp : SExp;
BEGIN
   New(Hlp);
   SetType(Hlp, LazySE);
   ALLOCATE(Hlp^.LazyEl, SIZE(LazyNode));
   Hlp^.LazyEl^.Function := Fun;
   Hlp^.LazyEl^.Arguments := Arg;
   RETURN Hlp;
END NewLazyNode;

PROCEDURE LazyCons(Car:SExp; CdrFun:FunType; CdrArgs:SExp):SExp;
VAR Hlp : SExp;
    LazyList : SExp;
BEGIN
   New(Hlp);
   Hlp := CdrArgs;
   LazyList := NewLazyNode(CdrFun, CdrArgs);
   Hlp := Cons(Car, LazyList);
   RETURN Hlp
END LazyCons;

PROCEDURE LazyHead(L: SExp): SExp;
BEGIN
   IF IsTypeSE(L, LazySE) THEN
      EvalLazyNode(L);
      RETURN Head(L);
   ELSE
      RETURN Head(L)
   END;
END LazyHead;

PROCEDURE LazyTail(L: SExp): SExp;
BEGIN
IF IsTypeSE(L, LazySE) THEN
   EvalLazyNode(L);
   RETURN Tail(L)
```

```
ELSE
   RETURN Tail(L)
END;
END LazyTail;
```

LazySE is the data structure thet internally represents the lazy list node. The function **Apply** applies its first argument (which is a function) to its second argument and returns the result of the application. The function **Odds** below shows how to create an infinite list of odd numbers starting at *n*: the start element of the list is *n*, and the rest are the odds beginning at *n*+2. Instead of a diveregent recursive call, we create a „lazy node". If we need it, its evaluation is going to be forced.

```
PROCEDURE EvalLazyNode( LazyList : SExp) ;
VAR
   Arg : SExp;
   Fun : FunType;
BEGIN
   IF IsTypeSE(LazyList, LazySE) THEN
      Fun := LazyList^.LazyEl^.Function;
      Arg := LazyList^.LazyEl^.Arguments;
      Update(LazyList,Apply(Fun, Arg));
   END;
END EvalLazyNode;

PROCEDURE Odds( n : SExp): SExp;
BEGIN
   IF (NOT (ODD(ValIntSE(n)))) THEN
      n := QInt(ValInt(n) + 1) ;
   END;
   RETURN LazyCons(n, OddsAdr, Cons(QInt(ValInt(n) + 2), Null));
END Odds;
```

The function **NthEl** shows how to find nth element of the list which is created with function **Odds**. The functions **LazyHead** and **LazyTail** force the evaluation of "lazy nodes.". The evaluation of odds (which could be divergent) stops in the function **Odds** resulting in a "lazy node."

```
PROCEDURE NthEl(n : INTEGER; L : SExp) : SExp;
BEGIN
  IF n = 1 THEN
     RETURN LazyHead(L)
  ELSE
     RETURN NthEl(n-1, LazyTail(L))
  END;
END NthEl;
```

The appropriate functions for examining if an expression is an element of a list from the previous example:

```
PROCEDURE LazyEqual(E, L : SExp): BOOLEAN;
VAR
 hE, hL, tE, tL: SExp;
BEGIN
 IF (Atom(E) OR Atom(L)) THEN
    RETURN EqSE(E, L)
 ELSE
    tL := LazyTail(L); hL := LazyHead(L);
    tE := LazyTail(E); hE := LazyHead(E);
    RETURN LazyEqual(hE, hL) AND LazyEqual(tE, tL)
 END
END LazyEqual;

PROCEDURE Memb(E, L: SExp) : BOOLEAN;
VAR hL, tL: SExp;
BEGIN
 IF Eq(L, NullSE) THEN
    RETURN FALSE
 ELSE
    hL := LazyHead(L);
    IF LazyEqual(E, hL) THEN
       RETURN TRUE
    ELSE
       tL := tail(L);
       RETURN Memb(E, tL)
    END
 END
END Memb;
```

We don't call the machine evaluator at all. The entire evaluation is performed within the procedural model. This way provides a realization of fast foreign functions with delayed evaluation.

4 Future work

There are a lot of functional language compilers with runtime systems written in procedural languages. The discussed method to define foreign functions with delayed evaluation could be used as an optimization of such implementations for lazy functional languages. We are currently working on such an optimization of Glasgow *Haskell* Compiler (direct translation of *Haskell* functions into *C*-with unboxed values and extension of the implementation with foreign functions).

5 Related work

The primary motivation for this work was to emend the SK-reduction based implementation of a functional language via communication with Modula-2. There are meny methods to combine *Scheme* and *C* functions and data structures via foreign functions. There also exists a foreign function interface which allows a *Common Lisp* program to access a database management system. But *Scheme* and *Common Lisp* are strict languages and the evaluation strategies between *Scheme/Common LISP* and *C* are compatible. We defined foreign functions in a strict language but lazy environment and gave a model for their lazy evaluation.

The Glasgow implementation of the non-strict functional language *Haskell* provides an interface to call C-functions from *Haskell* programs but there is no possibility to define foreign functions in our sense (as an extension to the procedural language).

6 Conclusion

We have shown how to implement the foreign functions with delayed evaluation within procedural paradigm when implementing functional languages by SK-graph reduction. The first approach shows how to implement lazy foreign functions by calling the SK-executor and the second one shows how to reach it by defining new data structures and runtime functions. The first way is rather natural: If something has to be evaluated lazy, let the lazy SK-reducer do that. The costs of that are the usual costs of the SK-graph reduction. The second way is the way which provides fast foreign functions with delayed evaluation. The overhead of the latter are more runtime system functions and data structures.

References

[1] Budimac, Z. and Ivanović, M., *On some Modula-2 Influences to an Implementation of Functional Language*. In Proc. of II Int. Conf. "Modula-2 and beyond" (Loughborough, Great Britain), 1991, 322-331.

[2] Diller, A., *Compiling Functional Languages*, John Wiley & Sons, 1988.

[3] Field, A. J. and Harrison, P.G., *Functional Programming*, Addison Wesley, New York, 1988.

[4] Ivanović, M. and Budimac, Z., *Involving Coroutines in Interaction Between Functional and Conventional Language*, SIGPLAN Notices, 11(25), 1990, 65-74.

[5] Peyton Jones, S., *The Implementation of Functional programming Languages*, Prentice Hall, 1987.

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 139–146

SK Implementation of Some Data Types

Saša Malkov[1], Nenad Mitić[2], Goran Lazić[2], and Aleksandra Gačević[3]

[1] Department of Genetics, Institute for Biological Research,
29. novembra 142, Belgrade
[2] Faculty of Mathematics, Studentski trg 16, Belgrade
[3] Faculty of Civil Engineering, Bulevar Revolucije 73, Belgrade

Abstract. The efficiency of SK reduction machine implementation can be increased by extending the primitive data types set. Including primitive data types (numbers, lists, strings,...) into the environment of pure combinators, the question arises about combinatorial properties of the primitive constants and their application and reduction. In this paper we describe a solution of this problem.

1 SK Combinators

We can define combinators as λ expression in which there are no occurrences of free variables [1]. Combinators have properties of operators which value depends exclusively on the argument values. The formal theory of the combinators is described in [2]. In theory any λ-expression can be represented using two basic combinators called `S` and `K`. Turner ([6]) described how functional programming languages can be compiled to `SK` combinatorial expressions, and first used SK reduction machine for its evaluation.

Even they can be expressed and defined using `S` and `K`, usually the SK expression language is extended by adding sets of constants and some primitive operators, with intention to make combinatorial expressions shortly and to make translation from a functional language simpler. When one adds some primitive types, for example numbers and strings, the question of combinatorial properties of added constants and operators often stay open. Here we pay attention to these properties defining some constants and operators on them using `S` and `K`. Using these constants and operators, some complex type constants and operators are defined. Finally, we analyse the behavior of all defined constants and operators inside combinatorial expressions, and their application and reduction, especially. To make combinatorial expressions simpler, we use the extension of `SK` expression language described in [5]. This extension allows direct recursive notation without usage of `Y` combinator.

Here we remind on some basic combinators (others than `S` and `K`) and their reduction rules:

`I= SKK`	$\mathtt{I}x \to x$
`B= S(KS)K`	$\mathtt{B}fxy \to f(xy)$
`C= S(BBS)(KK)`	$\mathtt{C}fxy \to fyx$
`P= S(K(SI))K`	$\mathtt{P}xy \to yx$

Let $K_{n,m}$ be the combinator that selects n-th of m arguments. The following equations holds:

$\mathtt{K}_{1,2} = \mathtt{K}$ $\qquad$ $\mathtt{K}_{2,2} = \mathtt{SK}$

$\mathtt{K}_{1,m+1} = \mathtt{S(KK)}\mathtt{K}_{1,m}$ $\qquad$ $\mathtt{K}_{n+1,m+1} = \mathtt{KK}_{n,m}$

2 Realisation of Logical Constants And Operators

The definition of the `if` operator is tightly related to definition of logical constants `true` and `false`. These two constants have to be defined in such a way to be useful by `if` combinator and other logical operators' definitions. It is very suitably to define `true` and `false` as selectors:

`true` $= \mathtt{K}$ $\qquad$ `true` $x\ y \rightarrow x$

`false` $= \mathtt{SK}$ $\qquad$ `false` $x\ y \rightarrow y$

Now, the `if` combinator can be defined as:

`if` $= \mathtt{I}$ $\qquad$ `if` $x\ \mathtt{e}_1\ \mathtt{e}_2 \rightarrow x\ \mathtt{e}_1\ \mathtt{e}_2$

Using these definitions it is possible to define logical operators `not`, `and`, `or`.

`not` $= \mathtt{C}\ (\mathtt{P\ false})\mathtt{true}$ $\qquad$ `not` $x \rightarrow$ `P false` x `true` $\rightarrow x$ `false true`

`and` $= \mathtt{C\ C\ false}$ $\qquad$ `and` $x\ y \rightarrow$ `C C false` $x\ y \rightarrow$

$\rightarrow$ `C` x `false` $y \rightarrow x\ y$ `false`

`or` $= \mathtt{P\ true}$ $\qquad$ `or` $x\ y \rightarrow$ `P true` $x\ y \rightarrow x$ `true` y

3 Realisation Of The List Constants And Operators

To make the usage of the lists possible, it is necessary to define list constructors and appropriate operators. Lists are built using `nil` constructor which denotes the empty list, and `cons` constructor which builds a new list from an element and an existing list. Operators `head` and `tail` (for extracting head and tail of a list), `isemptylist` and `notemptylist` (for testing if list is empty or not), and `append` (that joins two lists) are defined. We show that some other data types can be defined using these list constants and operators, for example natural numbers.

The `cons` constructor has to be defined as a combinator whose application on arguments H (head) and T (tail) results in combinator from which we can extract either the head or the tail. Because of that, we consider cons is three arguments operator, where the third argument is an operator that applies to list:

`cons` $H\ T\ F \rightarrow f\ H\ T$

The combinator that satisfies these propositions can be defined as: `cons = BCP`. As a consequence, the combinator that represents a list operator f has to be defined as `P` f:

`P` f `(cons H T)` $\rightarrow$ `cons H T` $f \rightarrow f$ `H T`

The `head` operator is `P`f, where f selects the first of two arguments, and the `tail` operator is `P`g where g selects the second of two arguments:

`head` $= \mathtt{PK}$ $\qquad$ `head (cons H T)` $\rightarrow$ `cons H T K` $\rightarrow$ `K H T` $\rightarrow$ `H`

`tail` $= \mathtt{P\ (SK)}$ $\qquad$ `tail (consH T)` $\rightarrow$ `cons H T (SK)` $\rightarrow$ `S K H T`$\rightarrow$ `T`

To make the lists' manipulation possible, it is necessary to allow testing on the list emptiness. We do that using the `notemptylist` operator that reduces to `true` when applied on non-empty list, and to `false` when applied on `nil`[1]. Idea is to represent an empty list with a constant operator, which will guarantee the different result than the `notemptylist` application on a non-empty list, when `cons` is evaluated. Appropriate `notemptylist` operator can be defined as:

`notemptylist` = `P(K(Ktrue))`

with reduction rule

`notemptylist (cons H T)` $\rightarrow$ `cons H T (K (K true))` $\rightarrow$ `K (Ktrue) H T` $\rightarrow$
$\rightarrow$ `K true T` $\rightarrow$ `true`

Using `notemptylist` we define `isemptylist` and append:

`isemptylist` = `B not notemptylist`
`append` l m = `isemptylist` l m (`cons (head` l`) (append (tail` l`)` m`)`)

Constructor `nil` has to ensure that `notemptylist nil` evaluates to `false`. It is simple to achieve this constructing a constant operator that reduces to `false` when applied on one argument:

`nil` = `K false` `nil` x $\rightarrow$ `K false` x $\rightarrow$ `false`
`notemptylist nil` $\rightarrow$ `nil (K(K true))` $\rightarrow$ `false`

Notice that `head` and `tail` evaluate to `false` when applied on `nil`.

4 Natural Numbers Introduction

The set of natural numbers can be introduced using lists, with Peano's arithmetic as basis. We consider the empty list is zero constant, and introduce larger numbers using `succ` operator[2]:

$\mathtt{N}_0$ = `zero`
$\mathtt{N}_{n+1}$ = `succ`$\mathtt{N}_n$ = $\mathtt{succ}^{n+1}$ `zero`

Define `zero` and `succ` combinators using lists:

`zero` = `nil`
`succ` = `cons K`

Define `pred` operator that evaluates number smaller by one using list operators:
`pred` = `tail` `pred` $\mathtt{N}_{n+1}$ $\rightarrow$ `tail (cons K` $\mathtt{N}_n$`)` $\rightarrow$ $\mathtt{N}_n$

Notice that `pred` is applicable only to numbers larger than `zero`, because application of `pred` on `zero` evaluates to `false`, which is not a number constant. The testing, if a natural number is zero or not, is defined as:

`notzero` = `notemptylist`
`iszero` = `isemptylist`

[1] Notice that we can define these combinators in opposite direction, defining first `isemptylist` as combinator `P(K(Kfalse))` which evaluates `false` when applied to a non-empty list, and defining `nil` as combinator `Ktrue` which evaluates `true` when applied on one argument. Then, it is: `notemptylist = B not isemptylist` [4]

[2] The power of combinators is defined as: $f^0\ \ y = y$, $f^{n+1}\ y = f\,(f^n\ y)$.

The comparison of natural numbers can be defined using Lisp-like syntax as:

```
eqnum x y = if (iszero x) (iszero y)
                (if (iszero y) false (eqnum (pred x) (pred y)))
```

which can be transformed to combinatorial expression

```
eqnum = C(BS(S(BB notzero) (B(S(S iszero (K false)))
        (B(C(B eqnum pred)) pred)))) iszero
```

Accordingly to the last expression complexity, in the further text we use Lisp-like syntax instead of transforming expressions to combinatorial language.

Operators that evaluate relations 'less than', 'less equal', 'greater than' and 'greater equal' can be defined as:

```
isltnum x y = iszero y false (iszero x true (isltnum (pred x)(pred y)))
islenum x y = or (isltnum x y) (eqnum x y)
isgtnum x y = not (islenum x y)
isgenum x y = not (isltnum x y)
```

After all, we define addition, multiplication and subtraction operators:

```
add x y = iszero x y (add (pred x) (succ y))
mul x y = iszero x zero (add (mul (pred x) y) y)
sub x y = iszero y x (sub (pred x) (pred y))
```

Notice that subtraction evaluates correct results only if arguments are correct.

A different approach is used in [2], with idea to use numbers as iterators[3,4]: $\mathtt{Z}_n\ f\ x \rightarrow f^n\ x$. Our definition is used to make the creation of combinators, that compares constants of defined types, possible. Notice that it is not simple to compare iterators from inside the combinatorial language.

5 Other Data Types

It is natural to represent strings as lists. A character we represent as a number, and a string as a list of characters:

```
emptystring  = nil
concatstring = append
```

We define vectors as lists of constant length. Vector implementation is based on `newvector` (the constructor of a vector with given size and initialisation), `getvector` (the extractor of a given element of a vector) and `setvector` (the operator that replaces given element of a vector with a given value) combinators with the following definition:

---

[3] Numbers are defined as: $\mathtt{Z}_0\ \ x\ y = y$, $\mathtt{Z}_{n+1}\ \ x\ y = x\ (\mathtt{Z}_n\ \ xy)$, `zero = false`, `succ = SB`. This approach produces interesting outcomes. Some numbers operations are defined elementary: `add= S' B`, where `S'` $f\ x\ y\ z \rightarrow f\ (x\ z)\ (y\ z)$, `mul= B`, `pow= I`, where `pow` $\mathtt{Z}_n\ \ \mathtt{Z}_m\ \rightarrow \mathtt{Z}_{mn}$.

[4] A variant of this approach is pointed out in [3], where the next definition is used: $\mathtt{Z}_{n+1}\ xy = \mathtt{Z}_n\ x(x\ y)$. The result is different `succ` operator definition: `succ= BW(BB)`, where `W=CSI`.

```
newvector n x  = iszero n nil cons x (newvector (pred n) x)
getvector v n  = notemptylist v (iszero n (head v)
                                  (getvector (tail v) (pred n))) false
setvector v n x = notemptylist v (iszero n (cons x (tail v))
                    (cons (head v) (setvector (tail v) (predn) x)) ) nil
```

Notice, that application of `getvector` with incorrect arguments evaluates to `false`, and application of `setvector` on incorrect arguments evaluates to `nil`.

Every combinator presents a program by itself, but in complex operations against combinators it becomes important if combinator involved in the evaluation process represent a constant, a data structure or a program. We introduce a data type that makes using marked programs possible: `prog` is constructor of marked program, and `apply` is its evaluator. It is goal to differ marked programs from other data types.

Idea is similar to one used when lists were defined, but `prog` operator captures only one argument, not two as `cons` does:

```
apply (prog comb) → comb
```

The next definitions of these combinators present a solution of previous condition, but allows much more than simple use of marked program (in opposition to, for example, definition `prog = K`):

```
prog              = P
apply             = PI
apply (prog comb) = PI (P comb) → P comb I → I comb → comb
```

6 Consequences

Behaviour analysing of defined types of combinators is based on their application and comparison. Application analysis is done examining constructors of all mentioned data types.

The logical constants are selectors of one from two arguments and they apply with next results:

```
true x y  → x
false x y → y
```

An empty list always evaluates to `false` (when applied on one argument), while a non-empty list application depends on its argument:

```
nil x       → false
cons H T x → x H T
```

The natural number application is similar to the list application. If we do not assume that a specific element is used in the list representing a number, numbers evaluate:

```
N0 x    → nil x        → false
Nn+1 x → cons H Nn x → x H Nn
```

Especially, if we assume that a number is a list that consists only of `K` combinators some other results arise:

$\mathtt{N}_{n+1}\ x \quad \rightarrow \mathtt{cons\ K\ N}_n\ x \quad \rightarrow x\ \mathtt{KN}_n$
$\mathtt{N}_{n+1}\ \mathtt{N}_{m+1} \rightarrow (\mathtt{succ\ N}_m)\ \mathtt{K\ N}_n \rightarrow \mathtt{K\ K\ N}_m\ \mathtt{N}_n \rightarrow \mathtt{KN}_n$
$\mathtt{N}_{n+1}\ \mathtt{N}_0 \quad \rightarrow \mathtt{N}_0\ \mathtt{K\ N}_n \quad \rightarrow \mathtt{false\ N}_n \rightarrow \mathtt{SKN}_n$

where $\mathtt{KN}_n$ is a constant operator that evaluates $\mathtt{N}_n$ when applied on one argument, and $\mathtt{SKN}_n$ is an identity operator, same as I is.

The strings, the vectors and the matrixes apply as lists do.

Shown application rules should be considered when incorporating these primitive types in combinators' reduction machine. The reduction machine implementation is not correct if primitive constants' reduction is not supported, because there are non-strict functional languages that can be expressed and programmed using combinators' reduction machine.

Another aspect of data types' behaviour is the possibility of their comparison. It is important to implement such data types that can be differed from inside combinator language. At least three different equality can be considered: syntactical ($=_1$), reductional ($=_2$) and semantical ($=_3$) equality.

Two combinators are syntactically equal iff the expressions that represent them are identical:

$\mathtt{K(KK)} \neq_1 \mathtt{S(K(KK))}$
$\mathtt{S(KKK)} \neq_1 \mathtt{SK}$

Two combinators are reductionally equal iff they can be eager reduced to syntactically equal expressions:

$\mathtt{K(KK)} \neq_2 \mathtt{S(K(KK))}$
$\mathtt{S(KKK)} =_2 \mathtt{SK}$

Two combinators are semantically equal iff they evaluate same results in all conditions. The semantical equality is often noticed as extensional equality:

$\mathtt{K(KK)} =_3 \mathtt{S(K(KK))}$
$\mathtt{S(KKK)} =_3 \mathtt{SK}$

Syntactical equality is most specialised, reductional represent somewhat wider relation, and semantical is most wider considered. For every two combinators Q and R:

$\mathtt{Q} =_1 \mathtt{R} \Rightarrow \mathtt{Q} =_2 \mathtt{R}$
$\mathtt{Q} =_2 \mathtt{R} \Rightarrow \mathtt{Q} =_3 \mathtt{R}$

Semantical equality is not calculable, i.e. the universal algorithm evaluating the semantical equality of two operators does not exist. Syntactical equality is calculable using metalanguage (the language in which reduction machine is implemented) and there can exist a primitive combinator that evaluates the syntactical equality of two combinators. Reductional equality is not calculable because reduction of combinators expressions if semi-calculable.

Here we consider only a segment of comparison problem: Is it possible to differ basic combinator types (logical constants, lists, marked programs) from within combinatorial language? Because all other structures are realised using lists, it is enough to recognise only mentioned types. If we can differ basic combinator types, it is enough to differ the constants of these separate types to differ all

available constants. Notice we already have defined the number comparison operator `eqnum`.

Assume our type testing operator is selection operator that selects appropriate of given arguments:

$$\texttt{testtype}\ x\ \textit{isbool}\ \textit{islist}\ \textit{isprog} \rightarrow \begin{cases} \textit{isbool} \text{ iff } x \text{ is } \texttt{true} \text{ or } \texttt{false} \\ \textit{islist} \text{ iff } x \text{ is } \texttt{nil} \text{ or } (\texttt{cons}\, H\ T) \\ \textit{isprog} \text{ iff } x \text{ is } (\texttt{prog comb}) \end{cases}$$

Combinator can not be analysed from outside - only technique to examine a combinator is by examining its behaviour. It is necessary to apply the tested combinator in conditions that provide a unique type depending result. Our `testtype` operator has to be of following form:

$\texttt{testtype}\ x\ a\ b\ c\ \ldots \rightarrow x\ a\ b\ c\ \ldots$

The arrity of `testtype` is not obvious. We have to compare applications of defined constants:

$\texttt{true}\ a\ b \rightarrow a$
$\texttt{false}\ a\ b \rightarrow b$
$\texttt{nil}\ a\ b\ c \rightarrow \texttt{false}\ b\ c \rightarrow c$
$\texttt{cons H T}\ a \rightarrow a\ \texttt{H T}$
$\texttt{prog comb}\ a \rightarrow a\ \texttt{comb}$

In many cases the result is argument a or its application. After replacing $a = \mathtt{K}_{5,5}$, and supplementing to seven arguments, all typed constants give different results:

$\texttt{true}\ a\ b\ c\ d\ e\ f\ g\ \textit{Bool List Prog} \rightarrow a\ c\ d\ e\ f\ g\ \textit{Bool List Prog}$
$\rightarrow g\ \textit{Bool List Prog}$
$\texttt{false}\ a\ b\ c\ d\ e\ f\ g\ \textit{Bool List Prog} \rightarrow b\ c\ d\ e\ f\ g\ \textit{Bool List Prog}$
$\texttt{nil}\ a\ b\ c\ d\ e\ f\ g\ \textit{Bool List Prog} \rightarrow c\ d\ e\ f\ g\ \textit{Bool List Prog}$
$\texttt{cons H T}\ a\ b\ c\ d\ e\ f\ g\ \textit{Bool List Prog} \rightarrow a\ \texttt{H T}\ b\ c\ d\ \textit{Bool List Prog}$
$\rightarrow d\ e\ f\ g\ \textit{Bool List Prog}$
$\texttt{prog comb}\ a\ b\ c\ d\ e\ f\ g\ \textit{Bool List Prog} \rightarrow a\ \texttt{comb}\ b\ c\ d\ e\ f\ g\ \textit{Bool List Prog}$
$\rightarrow e\ f\ g\ \textit{Bool List Prog}$

Now it is possible to define other arguments as selectors:

$a = \mathtt{K}_{5,5}$ $\quad b = \mathtt{K}_{6,8}$
$c = \mathtt{K}_{6,7}$ $\quad d = \mathtt{K}_{5,6}$
$e = \mathtt{K}_{5,5}$ $\quad g = \mathtt{K}_{1,3}$

while f is unimportant and can be `K`, for example.

Finally, the definition of `testtype` is:

$\texttt{testtype}\ x \rightarrow x\ \mathtt{K}_{5,5}\ \mathtt{K}_{6,8}\ \mathtt{K}_{6,7}\ \mathtt{K}_{5,6}\ \mathtt{K}_{5,5}\ \mathtt{K}\ \mathtt{K}_{1,3}$

Now we can define specific testing operators:

$\texttt{isboolean}\ x = \texttt{testtype}\ x\ \texttt{true false false}$
$\texttt{islist}\ x \quad = \texttt{testtype}\ x\ \texttt{false true false}$
$\texttt{isprog}\ x \quad = \texttt{testtype}\ x\ \texttt{false false true}$

Because of impossibility to compare programs, we can define:

$\texttt{eqprog}\ x\ y = \texttt{false}$

It is not complex to compare two logical constants:

$$\texttt{eqbool}\ x\ y = \texttt{if}\ x\ y\ (\texttt{not}\ y)$$

The list comparison is somewhat complexly and can be successfully evaluated only if lists, constructed of acceptable constants, are used:

$$\begin{aligned}\texttt{eqlist}\ x\ y = \texttt{if}\ &(\texttt{isemptylist}\ x)\ (\texttt{isemptylist}\ y)\ (\texttt{and}\ (\texttt{eqcomb}\ (\texttt{head}\ x)\\ &(\texttt{head}\ y))\ (\texttt{eqlist}\ (\texttt{tail}\ x)\ (\texttt{tail}\ y)))\end{aligned}$$

where eqcomb is used, which evaluates the equality of combinators of any of defined types:

$$\begin{aligned}\texttt{eqcomb}\ x\ y = \texttt{testtype}\ x\ &((\texttt{isbool}\ y)\ (\texttt{eqbool}\ x\ y)\ \texttt{false})\\ &((\texttt{islist}\ y)\ (\texttt{eqlist}\ x\ y)\ \texttt{false})\\ &\texttt{false}\end{aligned}$$

where this is applicable on numbers, if they are used strictly, as lists of K-s.

7 Conclusion

Some combinatorial data types and appropriate operators are defined. The combinatorial behaviour of constants of these data types is analysed. It is proposed that an implementation of combinators' reduction machine, that contains these types as primitive ones, should preserve described behaviour of constants.

The equality operators are defined for use inside these types, and an operator for comparing data types is defined. Using previous, an operator that compares all introduced data types is described and defined. By this we suggest that it is possible to create any complex type in a way its constants can be distinguished from constants of other types, from inside of combinatorial language. For that reason it is good to implement primitive types in reduction machine considering preservation of their constants combinatorial behaviour.

References

1. Barendregt, H.P.: *The Lambda Calculus*, 2nd edition, p.24. *North-Holland*, 1984
2. Curry, H. B., and Feys, R. *Combinatory Logic. Vol I.*, North-Holand 1958
3. Curry, H. B., Hindley, J. R., and Seldin, J. P. *Combinatory Logic. Vol II.*, North-Holand 1972
4. Field, A.J., and Harrison, P.G. *"Functional programming"*, *Addison-Wesley*, 1989
5. Malkov, S., Lazić, G., Gačević, A., and Mitić, N. *Implementation of Optimized SK Reduction Machine*, Proceedings of the VII Conference on Logic and Computer Science, LIRA'95, Novi Sad, September 26-30, 1995.
6. Turner, D. A. *Aspects of the implelemntation of programming languages*, PhD thesis, University of Oxford

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science *LIRA '97*
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 147–156

On Gödel Numbering Systems

Massimo Marchiori
CWI
Kruislaan 413, NL-1098 SJ,
Amsterdam, The Netherlands
max@cwi.nl

Abstract

Numberings are one of the fundamental milestones on which logic and computer science are based. Despite this, their study has been so far somehow neglected, maybe because their simplicity led to the erroneous assumption that, after all, there was not much to say about them. In this paper, we start to shed some light on numberings, exploring their structure with respect to their economicity. We introduce the framework of numbering systems, and refine it imposing a monotonicity condition. Within these concepts, that properly formalize numberings for sequences of data, we show that there are no most economical numberings due to the richness of the numbering structure. Then we consider those numberings systems that are allowed by machines as used in computer science. This leads to the result that there are still no most economical numbering systems except for one noticeable case, the two-counter register machine, where there is a most economical numbering system: quite surprisingly, it turns out to be just the original numbering system used by Gödel to prove his famous theorem on the incompleteness of formal theories.

Mathematics Subject Classification: 03D10, 03D45, 03E45, 03F40, 03F60, 03F65, 68Q05, 68T30.
CR Categories: B.2.0, B.4.0, E.4, F.0, F.1.0, F.1.1, F.4.1, H.1.1.
Keywords: numbering, economicity, register machine, recursive function.

1 Introduction

Numberings are ubiquitous both in logic and computer science. Their introduction dates back to Kurt Gödel who, in his masterpiece (Gödel, 1931) proving the incompleteness of formal theories, had the brilliant idea to code signs into (natural) numbers. His so-called Method of Arithmetization, cf. the discussions of R.B. Braithwaite in (Gödel, 1962) and of S.C. Kleene in (Gödel, 1986), assigns to any finite sequence $n_1, \ldots, n_k$ of natural numbers the 'Gödel number' $2^{n_1} \cdot 3^{n_2} \cdots \pi_k^{n_k}$ (where π_k is the k-th prime number). Via this numbering map every string (by associating to each of the primitive symbols a distinct number)

or finite sequence of numbers can be coded into a single number. This map, that was the core tool of Gödel's proof, was naturally generalized to the concept of numbering (also said Gödel numbering), that is an injective map to the natural numbers (sometimes with the additional requirement that the image is decidable).

This notion plays a fundamental role in a multitude of disciplines, ranging from the study of logical systems to recursion theory. However, its importance has been somehow inversely proportional to the number of studies on it. This is due to the fact that, after all, numberings (besides their 'philosophical' meaning) seem more a technical toolbox to be utilized rather than a topic to study on its own: indeed, the (to the best of our knowledge, negligible) work entirely devoted to them (e.g. see Manin, 1977, pp. 233–238) depicted numberings alone as trivial mathematical objects. For instance, the usual notion of equivalence between numberings (cf. Manin, 1977), saying that f and g are equivalent if $f \circ g^{-1}$ and $g \circ f^{-1}$ are computable, gives the flimsy result that all numberings are equivalent.

In this paper, we start to shed some light on numberings for their own: we formalize the numbering for sequences of data using the obvious concept of *numbering system*, much like recursive functions has been formalized using programming systems (Machtey and Young, 1978; Kfoury, Moll, and Arbib, 1982; Phillips, 1992), and natural numbers has been formalized into the λ-calculus using numeral systems (Barendregt, 1981).

Then, we refine this notion introducing a *monotonicity* requirement, roughly stating that the coding of sequences of a given length encompasses that of sequences of minor length, that has also a practical relevance for its natural connections with knowledge representation. Next, we address the problem of economicity of such (monotonic or not) numbering systems: a natural notion of 'expensiveness' is introduced, and it is shown that, due to the rich structure of numbering systems, no most economical (viz. minimal) one can be found in the general case.

We then relate numbering systems and machines (computational devices like Turing or register machines). Machines act by performing some operations on their internal memory, hence defining a class of partial functions over it. To compare the computational power of a machine with other classes of functions (like other machines' ones, the recursive functions etc.) one must use encoding and decoding functions. These functions play for machines the role that numberings play for logical systems (also under the point of view of their so far neglected study). It is so introduced a notion of allowedness that says when a numbering system can be safely used as an encoding function for a given machine, in the sense that it doesn't diminish the computational power of the machine itself. The question of economicity is then tackled again in this new context, searching for the most economical monotonic numbering systems among the ones allowed by a machine. The answer is in all similar to the general case, *except for one case only*: the two-counter register machine (the 'smallest' among the register machines). For it, there is a most economical encoding that, surprisingly

enough, turns out to be just the pioneering Gödel numbering from which the whole story began, thus showing that Gödel's original idea of numbering is not only an elegant trick among many, but has a deeper significance.

The paper is organized as follows. After introducing some notations in Section 2, machines are presented in Section 3. Then, Section 4 introduces numbering systems and monotonicity. In Section 5 economicity of numbering systems is discussed, both in the general case and in the monotonic case. Section 6 then establishes a link between numbering systems and encoding functions used in computer science via the notion of allowedness, and performs the analysis of economicity in this new context.

2 Notation

The symbols $\mathbb{N}$ and $\mathbb{N}_+$ stand for the set of natural numbers and the set of positive natural numbers respectively, and $\mathbb{N}^j$ will denote the cartesian product $\mathbb{N} \times \cdots \times \mathbb{N}$ (j times). We indicate the i-th prime with π_i (viz. $\pi_1 = 2$, $\pi_2 = 3$, $\pi_3 = 5$, etc.). With $\mathcal{P}_{fin}(A)$ we denote the set of the finite sets of elements in A. The symbol $\mathbf{REC}_k$ stands for all the (partial) recursive functions from $\mathbb{N}^k$ to $\mathbb{N}$.

It is well known (see e.g. Davey and Priestley, 1990) that from every (partial) order relation $\trianglelefteq$ one can recover the corresponding strict order $\triangleleft$, and vice versa. Hence, in the following we will arbitrarily use orders or strict orders.

As usual, given a function $f : A \to B$ and an element $b \in B$, $f^{-1}(b)$ denotes the set $\{a \in A : f(a) = b\}$. Also, we will often write $f(C)$ to denote the set $\{f(c) : c \in C\}$. Finally, the cardinality of a set A will be indicated by $card(A)$.

3 Machines and Programs

We fix three sets of symbols: $FUNC$ (the function symbols), $PRED$ (the predicate symbols) and $LABEL$ (the label symbols). $LABEL$ is also required to be infinite, and to have a distinguished element named $START$.

Then, a *machine* assigns to every function symbol $f \in FUNC$ (resp. to every predicate symbol $p \in PRED$) a partial function M_f (resp. a partial predicate M_p) over the set $|M|$, which is said the *memory set* of the machine M.

The strings of symbols of the form

i) $START$: GOTO L;
ii) L: DO f GOTO L';
iii) L: IF p THEN GOTO L' ELSE GOTO L'';
iv) L: HALT

with $f \in FUNC$, $p \in PRED$ and $L, L' \in LABEL$, are said the *instructions*. A *program* P is then a finite set of instructions that has exactly one instruction of type i), that is only one start instruction, and for every label L at most one instruction beginning with that label.

Given a machine M, every program P defines a partial function M_P over $|M|$ defined in the obvious way, simply 'computing' the program P starting from the (unique) instruction of type i) present in P, and using the function M_f in case an instruction of type ii) is found, and the predicate M_p in case a test instruction of type iii) is found. The computation ends when an halt instruction of type iv) is encountered. The formal definition is trivial but lengthy, so we omit it (see e.g. Clark and Cowell, 1976).

This way of defining machines is absolutely general, since it can represent all of the usual machines, like Turing ones, pushdown automata and so on (see Scott, 1967; Clark and Cowell, 1976).

For every program P, the function M_P is defined over $|M|$: if we want to compare the functions computable with M with other classes of functions having different domain and/or codomain, two fixed encoding and decoding functions must be introduced, where an encoding function is a computable function with codomain $|M|$, and a decoding function is a computable function with domain $|M|$. This way, we can compare some functions from A to B with the corresponding functions of a machine M by means of an encoding function $e : A \to |M|$ and of a decoding function $d : |M| \to B$, taking for every program P the function $d \circ M_P \circ e$.
For every machine M, if e and d are an encoding and decoding function for M, let us indicate with $\mathcal{F}(M, e, d)$ the set of functions $\{d \circ M_P \circ e \mid P \text{ is a program}\}$. Hence, saying that a machine M has full computational power equals to say that $\forall k \in \mathbb{N}$ there are functions $e_k : \mathbb{N}^k \to |M|$, $d_k : |M| \to \mathbb{N}$ such that $\mathcal{F}(M, e_k, d_k) = \mathbf{REC}_k$.

The *n-register machine* $\mathbf{SR}_n$ has as memory set n registers $R_1, \ldots, R_n$, each capable of holding a natural number (viz. $|\mathbf{SR}_n| = \mathbb{N}^n$), instructions $R_i \leftarrow R_i - 1$ and $R_i \leftarrow R_i + 1$ and predicates $R_i = 0?$ $(1 \leq i \leq n)$, with the obvious meaning of decrementing/incrementing a register by one and testing if a register has a zero value. The register machines $\mathbf{SR}_0$ and $\mathbf{SR}_1$ are readily too poor, but if $n \geq 2$ then $\mathbf{SR}_n$ has full computational power. Hence the most 'economical' register machine having full computational power is $\mathbf{SR}_2$, sometimes said the *two-counter register machine*.

En passant, we note how the importance of $\mathbf{SR}_2$ is not only purely theoretical, but even practical since, just to mention a few recent applications, it has been used to give the simplest known proof of undecidability of termination for logic programs (cf. Apt and Pellegrini, 1992) and for term rewriting systems (Bezem, Klop, and de Vrijer, 1997), and to provide the link between many seemingly unrelated topics in computer science (Kanovich, 1994).

4 Numbering Systems

Definition 4.1 A *numbering* is an injective computable function with codomain $\mathbb{N}$. □

Note that this definition, present e.g. in (Barendregt, 1981) for λ-terms, is not

completely standard: many other authors (for instance Manin, 1977; Phillips, 1992; Ebbinghaus, Flum, and Thomas, 1984) require also that the numbering image is decidable. We preferred here the more general definition: anyway, all the presented results hold using the more restrictive definition of numbering as well.

Definition 4.2 A *numbering system* $f_{[]}$ is an assignment to each $k \in \mathbb{N}$ of a numbering $f_{[k]} : \mathbb{N}^k \to \mathbb{N}$. □

To avoid confusions, an historical digression on the terminology is needed. The term "(Gödel) numbering" has also been used in another context with a different meaning, namely initially by Rogers with his seminal paper (Rogers, 1958) with the meaning of an enumeration of the partial recursive functions. To be precise, in the aforementioned paper Rogers distinguishes between the "usual" meaning of (Gödel) numbering (that is the one we employ) and his, which he calls of "special sort". Other later studies on the subject (until Hartmanis and Baker, 1973), have used this name. Other works within the period 1958–1977 (e.g. Meyer, 1972) used the names "effective enumeration of the partial recursive functions" or "indexing". Finally, roughly from 1978 onwards, the more appropriate term "programming system" was introduced (cf. Machtey and Young, 1978; Kfoury, Moll, and Arbib, 1982), and has become the actual standard (Phillips, 1992).

Definition 4.3 A numbering system $f_{[]}$ is *monotonic* if

$$\forall k \in \mathbb{N}, \forall (x_1, \ldots, x_k) \in \mathbb{N}^k.\ f_{[k+1]}(x_1, \ldots, x_k, 0) = f_{[k]}(x_1, \ldots, x_k) \qquad \square$$

The above definition formalizes the concept that every numbering $f_{[k]} : \mathbb{N}^k \to \mathbb{N}$ encompasses all the previous numberings $f_{[i]}$ $(0 \leq i < k)$. In fact, the usual numbering to cope with sequences of naturals (the numbering system) can be seen just as a numbering with domain $\cup_{i \in \mathbb{N}} \mathbb{N}^i$, thus treating sequences of different lengths as completely unrelated objects: merely a collection of numberings. Here, instead, we embed sequences of lesser length into ones of greater lengths utilizing the canonical embeddings $\mathbb{N}^k \hookrightarrow \mathbb{N}^{k'}$ $(k < k')$, given by $(x_1, \ldots, x_k) \mapsto (x_1, \ldots, x_k, 0, \ldots, 0)$. What we are doing, intuitively, is to give the flexibility of considering one number as a 'space' symbol, forgetting about leading spaces (just like in natural language processing we can find convenient not to treat as indistinguishable objects the five-character word 'hello' and the eight-character word 'hello ').

This notion has a precise mathematical significance too: instead of taking a numbering with domain $\cup_{i \in \mathbb{N}} \mathbb{N}^i$, it corresponds to take as domain the *direct limit* of the sequence

$$\mathbb{N}^0 \underset{\epsilon_0}{\hookrightarrow} \mathbb{N}^1 \underset{\epsilon_1}{\hookrightarrow} \mathbb{N}^2 \underset{\epsilon_2}{\hookrightarrow} \mathbb{N}^3 \hookrightarrow \ldots$$

(where the $\epsilon_k : \mathbb{N}^k \to \mathbb{N}^{k+1}$ are the canonical embeddings $(x_1, \ldots, x_k) \mapsto (x_1, \ldots, x_k, 0)$).

Moreover, this approach allows to arbitrarily enlarge the 'domain of the discourse' without changing everything regarding the domain: that is, if we worked with, say, at most sequences of length k and we want to enlarge the input domain, we can use longer sequences with the assurance that sequences from the original domain of the discourse (of length $\leq k$) are numbered exactly the same way as before. As a meaningful example, consider usage of numbering as an encoding function for providing inputs to some computing device: if the input domain is extended with some optional parameters, the old algorithm has only to be integrated to process the optional parameters when present, and not to be rewritten as a whole algorithm from the beginning.

We now introduce the well-known Gödel numbering system:

Definition 4.4 The *Gödel numbering system* $G_{[]}$ is defined as

$$G_{[k]}(x_1, \ldots, x_k) = \pi_1^{x_1} \cdot \pi_2^{x_2} \cdots \pi_k^{x_k} - 1 \qquad \square$$

Note that $G_{[]}$ is monotonic as well. The '-1' that distinguishes this definition from Gödel's one is simply due to the fact that it is customary to consider register machines with registers $\mathbb{N}$-valued, hence counting from 0 onwards: if we decide to count from 1 onwards defining $|\mathbf{SR}_n| = \mathbb{N}_+^n$ (and putting in place of $R_i = 0?$ the predicate symbol $R_i = 1?$ with the obvious meaning), then the '-1' can be dropped.

5 Economicity

In defining whether a numbering is better (that is, more economical) than another numbering, we have first of all to define what is the cost of a certain (finite) set of natural numbers D, let's say $\nu(D)$. For instance, we could set $\nu(D) = \sum_{x \in D} x$, or some other more involved measure. Abstracting, we generalize as much as possible requiring the cost function to be increasing, in the sense that greater numbers have greater cost:

Definition 5.1 A (cost) function $\nu : \mathcal{P}_{fin}(\mathbb{N}) \to \mathbb{N}$ is said to be *increasing* if

$$\nu(A) \leq \nu(B), r < s, r \notin A, s \notin B \Rightarrow \nu(A \cup \{r\}) < \nu(B \cup \{s\}) \qquad \square$$

Hence, if we have a numbering $f : \mathbb{N}^k \to \mathbb{N}$, the cost of codifying a certain set $A \subseteq \mathcal{P}_{fin}(\mathbb{N}^k)$ can be defined as $\nu(\{f(\bar{x}) : \bar{x} \in A\})$ or, briefly, $\nu(f(A))$. Then we could say that a numbering $f : \mathbb{N}^k \to \mathbb{N}$ is *no more expensive* than a numbering $g : \mathbb{N}^k \to \mathbb{N}$ ($f \preceq g$) if

$$\forall A \in \mathcal{P}_{fin}(\mathbb{N}^k).\ \nu(g(A)) - \nu(f(A)) \geq 0$$

(where ν, the cost function, is an increasing map from $\mathcal{P}_{fin}(\mathbb{N})$ to $\mathbb{N}$).

It can be proved that

Lemma 5.2 *The relation $\preceq$ defines an order on numberings.*

Now it has come the moment to define the corresponding cost ordering for numbering systems.

What is the minimal requirement we can impose on such a relation? If a numbering system $f_{[]}$ is 'less expensive' than $g_{[]}$, then there should be at least a k such that $f_{[k]} \prec g_{[k]}$ (i.e. at least a numbering of $f_{[]}$ should be less expensive than the corresponding numbering of $g_{[]}$). On the other hand, the reverse should not be true, that is to say there should be no k such that $g_{[k]} \prec f_{[k]}$. This is formalized into the following definition:

Definition 5.3 Taken two numbering systems $f_{[]}$ and $g_{[]}$, $f_{[]}$ is *less expensive* than $g_{[]}$ (notation $f_{[]} \lhd g_{[]}$) if

$$(\exists k.\ f_{[k]} \prec g_{[k]}) \wedge (\neg \exists k.\ g_{[k]} \prec f_{[k]}) \qquad \square$$

It can be proved the following:

Lemma 5.4 *The relation $\lhd$ is a strict order on monotonic numbering systems.*

Considering economicity w.r.t. $\lhd$, we obtain the following result:

Theorem 5.5 *There is not a monotonic numbering system that is minimal w.r.t. $\lhd$.*

6 The General Case

So far, we have only considered monotonic numbering systems. In this subsection we study the structure of numbering systems in complete generality, dropping the monotonicity requirement.

The problem with numbering systems is that the relation $\lhd$ is no more a strict order: indeed, it is not transitive, as it is not difficult to see. Hence, we must employ an expensiveness relation $\blacktriangleleft$ that fulfills the minimal requirements we have previously seen (i.e. $f_{[]} \blacktriangleleft g_{[]} \Rightarrow f_{[]} \lhd g_{[]}$) and that is a strict order as well.

Another minimal requirement the relation $\blacktriangleleft$ should satisfy is that if $\forall k.\ f_{[k]} \prec g_{[k]}$ then $f_{[]} \blacktriangleleft g_{[]}$ (i.e. if *every* numbering in $f_{[]}$ is less expensive than the corresponding in $g_{[]}$, then it should be the case that $f_{[]}$ is considered to be less expensive than $g_{[]}$ by the relation $\blacktriangleleft$).

In view of what seen, we have the following result:

Theorem 6.1 *Suppose $\blacktriangleleft$ is a strict ordering on numbering systems such that*

1. $f_{[]} \blacktriangleleft g_{[]} \Rightarrow f_{[]} \lhd g_{[]}$
2. $(\forall k.\ f_{[k]} \prec g_{[k]}) \Rightarrow f_{[]} \blacktriangleleft g_{[]}$

Then there is not a numbering system that is minimal w.r.t. $\blacktriangleleft$.

Hence, these results show that the structure of numbering systems, both in the general case and in the monotonic case, is so rich that there are no 'optimal' numbering systems.

7 Relating Numbering Systems with Machines

So far we viewed numbering systems as abstract objects: however, there is a natural way to relate them with machines, namely to consider them as encoding functions. This standpoint leads us to consider what numbering systems can be used by a given machine, and what not.

First, notice as properly speaking a numbering system cannot be used as encoding for register machines since it outputs to $\mathbb{N}$, whereas an encoding for a machine M must have as codomain $|M|$.

Hence, given a particular machine M we need to fix some embedding ϵ_M (i.e. an injective function) from $\mathbb{N}$ to $|M|$.

Definition 7.1 A numbering system $e_{[\,]}$ is *allowed* by a machine M (w.r.t. ϵ_M) if

$$\forall k \in \mathbb{N}, \exists d_k.\ \mathcal{F}(M, e_{[k]} \circ \epsilon_M, d_k) = \mathbf{REC}_k \qquad \square$$

In the case of register machines, since $|\mathbf{SR}_n| = \mathbb{N}^n$, we can simply use as $\epsilon_{\mathbf{SR}_n}$ the natural embedding $\imath_n : \mathbb{N} \hookrightarrow \mathbb{N}^n$ given by $m \mapsto (m, 0, \ldots, 0)$.

The nice fact is that this choice is not restrictive, since this natural embedding enjoys the following universal property:

Theorem 7.2 *If a numbering system is allowed by* $\mathbf{SR}_n$ *w.r.t. some embedding* $\epsilon_{\mathbf{SR}_n}$, *then it is also allowed w.r.t.* $\imath_n$.

In the sequel, we will therefore omit mentioning $\imath_n$, which is assumed to be the understood embedding for $\mathbf{SR}_n$.

We could so try to weaken the richness of numbering systems just seen in Theorem 5.5 by considering only (monotonic) numbering systems allowed by a given machine. It turns out that for all usual machines (e.g. Turing machines etc.), even imposing the allowedness requirement there are no most economical (monotonic or not) numbering systems. The only exception is, noteworthy, the case of the two-counter machine $\mathbf{SR}_2$.

Before going on, we need a preliminary definition, which generalizes the usual notion of injective function:

Definition 7.3 Let $f : \mathbb{N}^k \to \mathbb{N}$, and $I = \{i_1, \ldots, i_m\}$ a subset of $\{1, \ldots, k\}$. Then the function f is said to be *I-injective* if $(x_{i_1}, \ldots, x_{i_m}) \neq (x'_{i_1}, \ldots, x'_{i_m})$ implies that $f(x_1, \ldots, x_{i_1}, \ldots, x_{i_m}, \ldots, x_k) \neq f(x_1, \ldots, x'_{i_1}, \ldots, x'_{i_m}, \ldots, x_k)$. $\square$

Through a not easy proof, it can thus be given the following complete characterization of the numbering systems allowed by $\mathbf{SR}_2$.

Theorem 7.4 *A numbering system* $f_{[\,]}$ *is allowed by the two-counter register machine if and only if for every* $k \in \mathbb{N}$

$$f_{[k]}(x_1, \ldots, x_k) = q_1^{h_1(x_1,\ldots,x_k)} \cdot q_2^{h_2(x_1,\ldots,x_k)} \cdots q_{m_k}^{h_{m_k}(x_1,\ldots,x_k)} \cdot a_k + b_k$$

where

- $a_k \geq 1$
- b_k *is an integer*
- $m_k \geq 1$
- $q_1, \ldots, q_{m_k}$ *are coprime integers*
- $h_1, \ldots, h_{m_k}$ *are computable functions*

and there is a mapping $\Im_k : [1,k] \to [1,m_k]$ *such that* $\forall j \in [1,m_k]$ h_j *is* I_j*-injective, where* $I_j = \Im_k^{-1}(j)$.

This result, besides being important for its own, allows us to state the following:

Theorem 7.5 *The Gödel numbering system* $G_{[]}$ *is the minimum (w.r.t.* $\lhd$*) monotonic numbering system allowed by* $\mathbf{SR}_2$.

Hence, $G_{[]}$ is less expensive than *every other* monotonic numbering system allowed by $\mathbf{SR}_2$.

As said previously, the character of exception of this result is strengthened when we try to study what happens for the other register machines, since we have:

Theorem 7.6 *If* $n \geq 3$, *every numbering system is allowed by* $\mathbf{SR}_n$.

and so by Theorem 5.5 we obtain

Corollary 7.7 *There is no minimal monotonic numbering system w.r.t.* $\lhd$ *for the register machines* $\mathbf{SR}_n$ *when* $n \geq 3$.

The same happens when turning to all the other usual machines like Turing machines, Post machines and so on: Following the same headlines here presented for the register machines, it can be proved that

Theorem 7.8 *There is no minimal (monotonic or not) numbering system w.r.t.* $\lhd$ *for every machine*[1] *different from* $\mathbf{SR}_2$.

Thus, these results show that the structure of allowed numbering systems is extremely rich, such not to allow economical elements. Nevertheless, there is *only one case* where things are different: the two-counter register machine (the simplest among the register machines). For it, there is not only an economical element, but even *the* most economical one, which turns out to be just the first pioneering numbering system introduced by Gödel. In conclusion, what seen somehow justifies the fact that so far the study of numberings *per se* has been neglected: we have shown that their structure is so rich that it does not allow economical elements in any case, but for one important exception, where it unexpectedly reveals one of Kurt Gödel's jewels.

[1]Here 'every machine' means every standard machine introduced in the literature, e.g. all the machines in (Clark and Cowell, 1976; Hopcroft and Ullman, 1979)

References

Apt, K., and Pellegrini, A. (1992). On the occur-check free Prolog programs. Tech. rep. CS-R9238, CWI, Amsterdam, The Netherlands. Revised version appeared in *ACM TOPLAS* 16(3), pp. 687–726, 1994.

Barendregt, H. (1981). *The Lambda Calculus: its Syntax and Semantics*. North–Holland.

Bezem, M., Klop, J., and de Vrijer, R. (1997). *Term Rewriting Systems*, Vol. 25 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press. To appear.

Clark, K., and Cowell, D. (1976). *Programs, machines, and computation*. McGraw-Hill.

Davey, B., and Priestley, H. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

Ebbinghaus, H., Flum, J., and Thomas, W. (1984). *Mathematical Logic*. Springer–Verlag.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik, 38*.

Gödel, K. (1962). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Oliver&Boyd, Edinburgh and London.

Gödel, K. (1986). *Collected Works*, Vol. 1. Oxford University Press. Edited by S. Feferman.

Hartmanis, J., and Baker, T. (1973). On Simple Gödel Numberings and Translations. In Loeckx, J. (Ed.), *2nd Colloquium on Automata, Languages and Programming*, Vol. 14 of *LNCS*, pp. 301–316. Springer–Verlag.

Hopcroft, J., and Ullman, I. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Kanovich, M. (1994). Petri Nets, Horn Programs, Linear Logic, and Vector Games. In Hagiya, M., and Mitchell, J. (Eds.), *International Symposium on Theoretical Aspects of Computer Software*, Vol. 789 of *LNCS*, pp. 642–666. Springer–Verlag.

Kfoury, A., Moll, R., and Arbib, M. (1982). *A programming approach to computability*. Springer–Verlag, New York.

Machtey, M., and Young, P. (1978). *An Introduction to the General Theory of Algorithms*. North–Holland.

Manin, Y. (1977). *A Course in Mathematical Logic*. Springer–Verlag.

Meyer, A. (1972). Program Size in Restricted Programming Languages. *Information and Control, 21*, 382–394.

Phillips, I. (1992). Recursion Theory. In Abramsky, S., Gabbay, D. M., and Maibaum, T. (Eds.), *Handbook of Logic in Computer Science*, Vol. 1, chap. 1, pp. 79–188. Clarendon Press, Oxford.

Rogers, H. (1958). Gödel numberings of partial recursive functions. *Journal of Symbolic Logic, 23*(3), 331–341.

Scott, D. (1967). Some definitional suggestions for automata theory. *Journal of Computer and System Sciences, 1*, 187–212.

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 157–162

Using quasigroups for one-one secure encoding

Smile Markovski, Danilo Gligoroski, Suzana Andova

University "St. Cyril and Methodius"
Faculty of Natural Sciences, Institute of Informatics
P. O. Box 162, Skopje, Republic of Macedonia

Abstract. In this article we apply a method for encrypting messages based on the properties of the quasigroups. According to the analysis given in the article the method is extremely secure. Beside that, the plain text and its cipher text are of the same length, and the encoding is of stream nature guarantying a very fast implementation.

AMS Subject Classification (1991): 94A60, 68P25, 20N05

1 Introduction

In today modern Internet civilization more than ever we are faced by the need of fast and secure communication. The security in the communication is implemented mainly through two types of algorithms: 1. Algorithms that use public keys [3, 13, 2] (a typical example is the well known RSA algorithm) which are appropriate for offline communications and for authentications, but they are usually slow for online communication. 2. Algorithms that use secret key (a typical example is the so called "*Data Encryption Standard - DES*"[4, 12, 10]), which are more appropriated for online communications. The encoding method we are proposing in this paper is of the second type.

Of course, when we are dealing with the algorithms with secret key, one needs a secure channel for key transfer, and that problem can be solved using some algorithm with public key. Communication with secret key implies that for every couple (or group) that wants a secure communication, a separate secret key should be generated and memorized.

The method we describe here uses secret key that represent a quasigroup cipher. In fact, quasigroups have very useful properties which can be used for construction of functions for encryption and decryption. The problem of obtaining suitable quasigroups is also considered in the article [7] where it is shown how, by using isotopes of quasigroups, one can produce $(n!)^3$ different quasigroups on a carrier of power n. Here we use a modified Hall algorithm [6] for

generating $n \times n$ Latin squares and there are at least $n!(n-1)!\ldots 2!1!$ such latin squares. Clearly, any Latin square can be viewed as a quasigroup, and vice versa. We also mention here that [5] contains a method that can be reduced to the one considered in this paper.

2 Basic mathematical definitions

Here we give some basic notions for quasigroups.

Definition 1. A quasigroup is an algebra $(Q, *)$ with one binary operation satisfying the law: $(\forall a, b \in Q)(\exists! x, y \in Q) \;\; a * x = b \wedge y * a = b$

Definition 2. A $k \times n$ Latin rectangle on an alphabet $A = \{a_1, \ldots, a_n\}$ is a matrix with entries $a_{i,j} \in A$, $i = 1, 2, \ldots, k$, $j = 1, 2, \ldots, n$ such that each row and each column consists of different elements of A. If $k = n$ we say a Latin square instead of a Latin rectangle.

It is clear that if $A = \{a_1, \ldots, a_n\}$ is a carrier of a quasigroup $(A, *)$ then its Cayley table can be considered as a $n \times n$ Latin square, and vice versa. This correspondence allows us to use any method of constructing a Latin square for obtaining a quasigroup. One such method gives us a corollary of the well known P. Hall's theorem ([6, 8]), which states that any $k \times n$ Latin rectangle can be extended to a $(k+1) \times n$ Latin rectangle, for each $k = 0, 1, \ldots n-1$, and the extension can be made in at least $(n-k)!$ ways. As a consequence we have that there are at least $n!(n-1)!\ldots 2!1!$ $n \times n$ Latin squares over an alphabet with cardinality n. A table of the numbers of the $n \times n$ Latin squares for $n = 1, \ldots, 10$ is given in [11].

Proposition 3. *Given a quasigroup $(Q, *)$ define a binary operation $\backslash$ on Q as follows: $x \backslash y = z \iff x * z = y$, for all $x, y \in Q$. Then the groupoid $(Q, \backslash)$ is also a quasigroup.* □

Definition 4. We say that the operation $\backslash$ is dual to $*$, and that $(Q, \backslash)$ is a dual quasigroup to $(Q, *)$. We also say that the algebra $(Q, *, \backslash)$ is a quasigroup (-an expansion of $(Q, *)$).

Proposition 5. [1] *The quasigroup $(Q, *, \backslash)$ satisfies the following identities: $x \backslash (x * y) = y$. $x * (x \backslash y) = y$.* □

3 Description of the method

Let $A = \{a_1, a_2, \ldots, a_n\} (n \geq 1)$ be an alphabet, and let $(A, *, \backslash)$ be the quasigroup defined as in the previous section. Denote by A^+ the set of all nonempty words of the alphabet A. Define two unary operations f_* and $f_\backslash$ on A^+ as follows:

Definition 6. Let $u_i \in A, k \geq 1$. Then

$$\begin{aligned} & f_*(u_1 u_2 \dots u_k) = v_1 v_2 \dots v_k \\ \iff & v_1 = a_1 * u_1,\ v_{i+1} = v_i * u_{i+1},\ i = 1, 2, \dots, k-1, \end{aligned} \tag{1}$$

$$\begin{aligned} & f_\backslash(u_1 u_2 \dots u_k) = v_1 v_2 \dots v_k \\ \iff & v_1 = a_1 \backslash u_1,\ v_{i+1} = u_i \backslash u_{i+1},\ i = 1, 2, \dots, k-1. \end{aligned} \tag{2}$$

We say that the sixtuple $(A, *, \backslash, a_1, f_*, f_\backslash)$ is a quasigroup cipher over the alphabet A.

Lemma 7. *If $(A, *, \backslash, a_1, f_*, f_\backslash)$ is a quasigroup cipher over the alphabet $A = \{a_1, \dots, a_n\}$, then $f_\backslash \circ f_* = 1_{A^+}$ where 1_{A^+} is the identical map on A^+ and $\circ$ is the composition of maps.*

Proof. Let $u_i \in A$, $k \geq 1$ and $f_*(u_1 \dots u_k) = v_1 \dots v_k$, $f_\backslash \circ f_*(u_1 \dots u_k) = f_\backslash(v_1 \dots v_k) = w_1 \dots w_k$. Then we have: $v_1 = a_1 * u_1$, $v_{i+1} = v_i * u_{i+1}$, $w_1 = a_1 \backslash v_1$, $w_{i+1} = v_i \backslash v_{i+1}$, for $i = 1, 2, \dots, k-1$. So, by Proposition 5, $w_1 = a_1 \backslash (a_1 * u_1) = u_1$, $w_{i+1} = v_i \backslash (v_i * u_{i+1}) = u_{i+1}$ for $i = 1, 2, \dots, k-1$. □

Now, it is quite clear from Lemma 7 that we can take $f_e = f_*$ as an encoding function, and $f_d = f_\backslash$ as a decoding function, for enciphering and deciphering over an alphabet A. Namely, if $u \in A^+$ is a plain text, then $f_*(u)$ is its cipher text, and as we have seen, $f_\backslash(f_*(u)) = u$.

Example 1. Let $A = \{a, b, c\}$ and let the quasigroup $(A, *)$ i.e. $(A, *, \backslash)$ be defined by the Table 1. Let $a_1 = a$ and $u = bcaaabbca$. Then the cipher text of u is $v =$

Table 1. The quasigroup $(A, *, \backslash)$

$*$	a	b	c
a	b	c	a
b	c	a	b
c	a	b	c

$\backslash$	a	b	c
a	c	a	b
b	b	c	a
c	a	b	c

$f_*(u) = ccabcbaab$. Applying decoding function $f_\backslash$ on v we get $f_\backslash(ccabcbaab) = bcaaabbca = u$.

4 Some properties of the method

At first, we should note that the above method produces a cipher text with the same length as the plain text. Moreover, each letter of the plain text is encoded by a single letter, and in fact the encoding is of a stream nature. That is the reason why this method is appropriate for a fast online communication. When

computer realizations are considered, in comparatione with DES algorithm which uses several rounds of simple computations, this method needs only access to the memory.

Another nice property of this method is its robustness on errors. Namely, we have:

Proposition 8. *Let $u = u_1u_2 \dots u_k \in A^+$ be a plain text, $v = f_*(u) = v_1v_2 \dots v_k$ its cipher text and $v' = v_1v_2 \dots v_{i-1}v'_iv_{i+1} \dots v_k$ $(v'_i \in A)$. Then*

$$f_\backslash(v') = u_1u_2 \dots u_{i-1}u'_iu'_{i+1}u_{i+2} \dots u_k,$$

for some $u'_i, u'_{i+1} \in A$.

Proof. It follows directly of the definition of the decoding function $f_\backslash$. □

The last property implies that this method can be used for designing secure databases, the security being levelled at the contents of the fields of a database [9].

As we already mentioned, this method can be used for online digital communications where data are represented by 8 bits, i.e. we take the alphabet $A = \{0, \dots, 255\}$. In that case there are at least $256!255! \cdots 2!1! > 10^{58000}$ quasigroups. Now, suppose that an intruder knows a cipher text $v = v_1 \dots v_k = f_*(x_1 \dots x_k)$, where $x_1 \dots x_k$ represents the unknown plain text. Then for recovering the quasigroup $(A, *)$, which is the key of the encoding method, he(she) should solve a system of equalities of the form:

$$\begin{aligned} v_1 &= a_1 * x_1 \\ v_2 &= v_1 * x_2 \\ &\dots \\ v_k &= v_{k-1} * x_k. \end{aligned}$$

But, the above system of equalities has as many solutions as there are quasigroups of order 256, which means that the method can successfully resist on the brute force attack.

Unfortunately, if an intruder knows both the plain and the cipher text, he(she) can easily recover the quasigroup $(A, *)$. To avoid this weakness of the method we propose two (or more) quasigroups as a key for the encoding. Namely, let $(A, *)$ and $(A, *')$ be two different and not mutually dual quasigroups, with the corresponding quasigroups ciphers $(A, a_1, *, \backslash, f_*, f_\backslash)$ and $(A, a'_1, *', \backslash', f_{*'}, f_{\backslash'})$. Then as encoding and decoding functions we take

$$f_e = f_{*'} \circ f_*, \quad f_d = f_\backslash \circ f_{\backslash'}.$$

Now, if the plain text $u = u_1 \dots u_k$ and its cipher text $v = v_1 \dots v_k = f_e(u_1 \dots u_k)$ are known, for finding the key consisting of two quasigroups $(A, *)$ and $(A, *')$, the intruder should solve a system of equalities of the form

$$x_1 = a_1 * u_1, \; x_2 = x_1 * u_2, \; \dots, \; x_k = x_{k-1} * u_k \tag{3}$$

$$v_1 = a'_1 *' x_1,\ v_2 = v_1 *' x_2,\ \ldots,\ v_k = v_{k-1} *' x_k, \tag{4}$$

where $a_1, a'_1, x_1, \ldots, x_k$ are unknown. If (3) and (4) are considered separately, then any quasigroup $(A, *)$ (i.e. $(A, *')$) is a solution of (3) (i.e. (4)). But, not any pair of quasigroups is a solution of the system of equalities (3) and (4). This means that if one wants to solve the above system of equalities, he(she) should choose an arbitrary quasigroup $(A, *)$, and after that to check if the equalities (4) are satisfied in some quasigroup $(A, *')$. Of course, one may not generate the whole quasigroup $(A, *)$ for checking the satisfiability of (4), since during the process of building of $(A, *)$, simultaneously can be checked (4). Nevertheless, one should make as many attempts as there are quasigroups $(A, *)$.

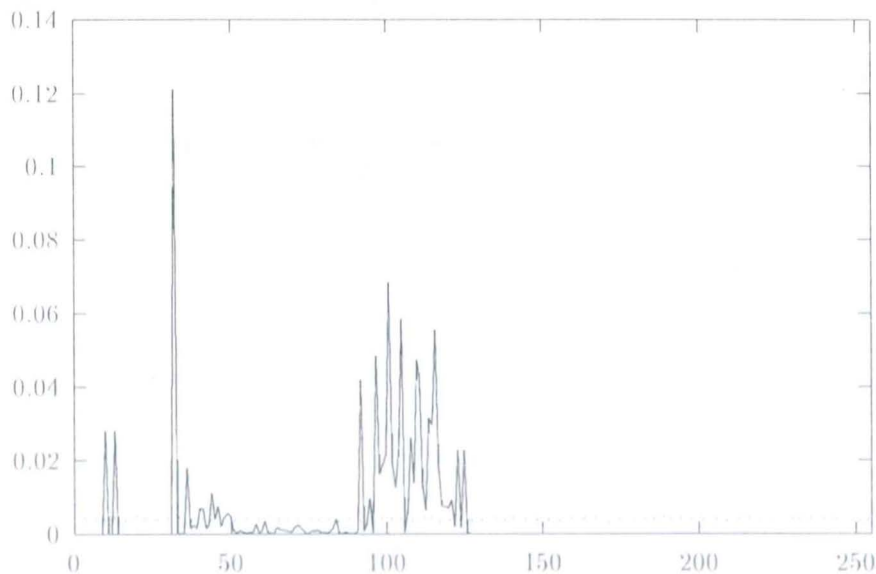

Fig. 1. Distribution of characters found in usual TEX file (solid line) and the distribution of characters found in its cipher text (doted line).

The resistance of the method on statistical attacks seems to be very well. In Figure 1 we represent the distributions of the characters of a plain text and of its cipher text, and the uniform distribution of the characters of the cipher text is evident. We note that the same uniform distributions occurred in every of more than 100 experiments we have made. The same phenomena appears when pairs and triplets of characters are considered.

The problem of constructing a random quasigroup of order n is easily solvable. In fact, we can produce such a quasigroup by using the algorithm for finding a system of different representatives of a family of sets [6, 8].

5 Conclusion

In this article we described a method for enciphering messages that uses transformations defined by quasigroups. The method is extremely resistive on the brute force or any statistical attack, as well as by the attack when both plain and cipher text are known. Beside that, it is robust on errors. Enciphering can be done very fast and the corresponding cipher text has the same length as the plain text. In practical implementations it gives cipher texts at the output that has uniform distribution. The method is practically implemented in programming language C for online communication as an utility for UNIX.

We note that instead of quasigroups, one can use left cancelative finite groupoids $(A, *)$, since then the equations $a * x = b$ has the unique solution $x = a \setminus b$. Unfortunately, in this case bad statistical properties may appear.

References

1. Belousov, V.D.: Osnovi teorii kvazigrup i lup. (1967) Nauka Moskva
2. Brassard, G.: Modern Cryptology. Lecture Notes in Computer Science **325** (1988) Springer Verlag Berlin
3. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Informat. Theory **22** (1976) 644–654
4. Diffie, W., Hellman, M.E.: Privacy and authentication: An introduction to cryptography. Proc. IEEE **67** (1979) 397–427
5. Gligoroski, D., Dimovski, D., Kocarev, L., Urumov, V., Chua, L.O.: A method for encoding messages by time targeting of the trajectories of chaotic systems. Int. J. Bif. Chaos **6** (1996) 2119–2125
6. Hall, M.: Combinatorial theory. (1967) Blaisdell Publishing Company, Massachusetts
7. Kościelny, C.: A method of constructing quasigroup-based stream-ciphers. Appl. Math. and Comp. Sci. **6** (1996) 109–121
8. Markovski, S.: Konechna matematika. (1993) Univerzitet Sv. Kiril i Metodij, Skopje
9. Markovski, S., Gligoroski, D.: Using quasigroups for designing secure database contents. (preprint)
10. Massey, J.L.: On the security of multiple encryption. Comm. ACM **24** (1981) 465–467
11. McKay, B.D., Rogoyski, E.: Latin squares of order 10. Electronic J. Comb. **2** (1995) http://ejc.math.gatech.edu:8080/Journal/journalhome.html
12. Moris, R.: The data encryption standard - retrospective and prospects. IEEE Commun. Mag. **16** (1978) 11–14
13. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Comm. ACM **21** (1978) 120–126

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 163–166

An algorithmic approach to the infimum of a graph

Dragan Mašulović

Institute of Mathematics, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: masul@unsim.im.ns.ac.yu

Abstract. This paper deals with an integer invariant of graphs which we shall refer to as the infimum of a graph: $\inf G := \max_{H \leq G} \delta(H)$. After a bit of motivation, we present an $O(n^2)$ algorithm to compute $\inf G$ for an arbitrary graph G.

Key Words and Phrases: graph, algorithm

AMS Subject Classification (1991): 05C85

1 Notation and Preliminaries

Throughout the paper, by a *graph* we mean a *simple nondirected finite graph without loops*. Given a graph G, $V(G)$ denotes the set of vertices of G while $E(G)$ denotes the set of edges of G. Let $n(G) = |V(G)|$ and $m(G) = |E(G)|$. By $\delta_G(v)$ we denote the degree of a vertex v in G. The subscript is usually omitted if the context is unambiguous. Let $\delta(G) = \min_{v \in V(G)} \delta(v)$. By a subgraph we mean a subgraph of the given graph induced by a set of vertices. $H \leq G$ stands for "H is a subgraph of G". If $W \subseteq V(G)$, $G[W]$ denotes the subgraph of G induced by W. Let $\operatorname{sub} G = \{H : H \leq G \wedge m(H) > 0\}$. Let $\kappa(G)$ denote the chromatic number of G. Let $v \in V(G)$ be an arbitrary vertex. $G - v$ is a graph obtained from G by removing the vertex v and all the edges incident to v. A vertex $v \in V(G)$ is isolated iff $\delta_G(v) = 0$.

This paper considers an integer invariant of graphs which we shall refer to as *the infimum of a graph*:

$$\inf G := \max_{H \leq G} \delta(H).$$

Section 2 lists several graph-theoretic notions that are connected to the infimum of a graph and gives the motivation for the result. Section 3 presents a $O(n^2)$ algorithm to compute $\inf G$ for arbitrary graph G.

2 Motivation

Players A and B play the following game: each player is given a copy of a graph G and player A marks an edge $e^* \in E$. Player B is required to find the edge e^* by a sequence of tests. Each test is a set $S \subseteq V$, and the answer to a test is 0 if $e^* \cap S = \emptyset$ or 1, otherwise. Let $\overline{c}(G)$ be the minimum number of tests required to find any edge in G. A graph G is called almost 2-optimal iff

$$\lceil \log_2 m(G) \rceil \leq \overline{c}(G) \leq \lceil \log_2 m(G) \rceil + 1.$$

It is proved in [1] that a graph G is almost 2-optimal if $\inf G \leq 5$.

The algorithmic nature of search problems on graphs is, however, sharply contrasted with a "non-algorithmic" definition of $\inf G$. This contrast initiated an attempt to provide an efficient algorithm to compute $\inf G$. Before we turn to the algorithm, let us look at a few more graph-theoretic concepts which are connected to the infimum of a graph.

The vertex-arboricity, $a_v(G)$, of a simple graph G is the minimum cardinality k of a partition $\{W_1, \ldots, W_k\}$ of $V(G)$ with the property that induced subgraphs $G[W_i]$ are forests for all $i = 1 \ldots k$. Similarly, the edge-arboricity, $a_e(G)$, of a simple graph G is the minimum cardinality ℓ of a partition $\{F_1, \ldots, F_\ell\}$ of $E(G)$ with the property that induced subgraphs $G[F_j]$ are forests for all $j = 1 \ldots \ell$. Given a graph G with at least one edge, in [2, 5] and [3, 5] the following bounds for $a_v(G)$ and $a_e(G)$, respectively, are given:

$$a_v(G) \leq 1 + \left\lfloor \frac{\inf G}{2} \right\rfloor \qquad \text{and} \qquad a_e(G) \geq \left\lceil \frac{1 + \inf G}{2} \right\rceil.$$

In [4, 5] one can find the following upper bound for the chromatic number of a graph: $\kappa(G) \leq 1 + \inf G$.

3 The Algorithm

Given a graph G with $m(G) > 0$, let $\delta^+(G)$ denote the least positive degree of a vertex of G, i.e. $\delta^+(G) = \min\{\delta_G(v) : v \in V(G) \wedge \delta_G(v) > 0\}$. If $m(G) > 0$, then, obviously, $\inf G = \max\{\delta^+(H) : H \in \operatorname{sub} G\}$.

We shall now present another approach to $\inf G$ based on a sequence of vertices of "small" degree. This sequence can be easily determined. As we shall see, considering such sequences suffices to determine the infimum. Let us start with a technical definition.

Definition 1. Given a graph G with at least one edge, consider a sequence of vertices $v_0, \ldots, v_\ell$ and a sequence of subgraphs $A_0, \ldots, A_\ell, A_{\ell+1}$ of G constructed recursively as follows:

(*i*) $A_0 = G$

(*ii*) v_i is any vertex of A_i such that $\delta_{A_i}(v_i) = \delta^+(A_i)$, $i = 0 \dots \ell$

(*iii*) $A_{i+1} = A_i - v_i$

(*iv*) $m(A_\ell) > 0$ and $m(A_{\ell+1}) = 0$.

A sequence $v_0, \dots, v_\ell$ is called $\delta_{\min}$-sequence of vertices, while $A_0, \dots, A_\ell, A_{\ell+1}$ is called $\delta_{\min}$-sequence of subgraphs of G. We say that the two sequences are corresponding.

Note that $\delta_{\min}$-sequences are not unique. What follows is a lemma which connects $\delta_{\min}$-sequences of vertices with subgraphs of G.

Lemma 2. *Given a graph G with at least one edge, arbitrary $\delta_{\min}$-sequence $v_0, \dots, v_\ell$ of vertices of G and a subgraph H of G, there is an integer $k \in \{0, \dots, \ell\}$ such that $v_k \in V(H)$ and $\delta_H(v_k) > 0$.*

Proof. Remove all the isolated vertices from H and denote the new graph with H'. Obviously, $m(H') = m(H) > 0$ and $H' \in \operatorname{sub} G$. Let us show that there is a vertex v_k such that $v_k \in V(H')$. Suppose, on the contrary, that $V(H') \cap \{v_0, \dots, v_\ell\} = \emptyset$. Then $H' \leq G - v_0 - \dots - v_\ell = A_{\ell+1}$. But $m(A_{\ell+1}) = 0$, while $m(H') > 0$. Contradiction.

It is now easy to see that $v_k \in V(H')$ implies $v_k \in V(H)$ and $\delta_H(v_k) > 0$.

Theorem 3. *Let G be a graph with at least one edge. Let $v_0, \dots, v_\ell$ be a $\delta_{\min}$-sequence of vertices of G and let $A_0, \dots, A_\ell, A_{\ell+1}$ be the corresponding $\delta_{\min}$-sequence of subgraphs of G. Then*

$$\inf G \overset{*}{=} \max\{\delta_{A_i}(v_i) : i = 0 \dots \ell\}.$$

Proof. ($\geq$) Since $\operatorname{sub} G \supseteq \{A_0, \dots, A_\ell\}$, we have $\inf G \geq \max\{\delta^+(A_i) : i = 0 \dots \ell\} = \max\{\delta_{A_i}(v_i) : i = 0 \dots \ell\}$.

($\leq$) Firstly, we show that for each $H \in \operatorname{sub} G$ there is an integer j such that $\delta^+(H) \leq \delta_{A_j}(v_j)$. W. l. o. g. we can assume that H has no isolated vertices. According to lemma 2 there is an integer k such that $v_k \in V(H)$ and $\delta_H(v_k) > 0$. Let j be the minimum of all those k's. $V(H) \cap \{v_0, \dots, v_{j-1}\} = \emptyset$ and $H \leq G - v_0 - \dots - v_{j-1} = A_j$. From $v_j \in V(H)$ and $\delta_H(v_j) > 0$ one easily concludes that $\delta^+(H) \leq \delta_H(v_j)$. Since $H \leq A_j$, we have $\delta_H(v_j) \leq \delta_{A_j}(v_j)$.

Therefore, for each $H \in \operatorname{sub} G$ we have $\delta^+(H) \leq \max\{\delta_{A_i}(v_i) : i = 0 \dots \ell\}$. Taking the maximum of the lefthand side of the inequality over all $H \in \operatorname{sub} G$ we get $\inf G = \max\{\delta^+(H) : H \in \operatorname{sub} G\} \leq \max\{\delta_{A_i}(v_i) : i = 0 \dots \ell\}$.

The previous theorem is straightforwardly interpreted in the form of an algorithm:

```
function inf(G : Graph) : cardinal;
var
    d : cardinal;
    v : vertex;
begin
    d := 0;
    while m(G) > 0 do
        v := a vertex of G with the least positive degree;
        d := max(d, δ_G(v));
        G := G − v
    end;
    return d
end inf
```

The time-complexity of the algorithm is $O(n^2)$, where $n = n(G)$, which justifies the adverb "efficient".

Acknowledgements

I would like to acknowledge dr Ratko Tošić and dr Vojislav Petrović for their valuable comments.

References

1. Tošić R., Mašulović D.: *A binary search problem on graphs*, Univ. u Novom Sadu Zb. Rad. Prirod.-Mat. Fak. Ser. Mat. 24, 1 (1994) 231–243
2. Chartrand G., Kronk H. V.: *The point arboricity of planar graphs*, J. London Math. Soc. 44 (1969) 612–616
3. Burr S. A.: *An inequality involving the vertex arboricity and edge arboricity of a graph*, J. Graph Theory 10 (1986) 403–404
4. Szekeres G., Wilf H. S.: *An inequality for the chromatic number of a graph*, J. Combin. Theory 4 (1968) 1–3
5. Chartrand G., Lesniak L.: *Graphs & Digraphs*, Chapman & Hall, 3rd Ed, 1996

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 167–175

Structural properties of intersection types

Emilie Sayag, Michel Mauny*

INRIA-Rocquencourt, Projet Cristal, B.P. 105, F-78153 Le Chesnay Cedex, France

Abstract. We study an intersection type system which is a restriction of the intersection type discipline. This restriction leads to a purely syntactic and completely characterized notion of principal types. Using the equivalence between principal types and normal forms, we define an expansion operation on types which allows us to recover all possible types for any normalizable λ-term. The contribution of this work is a new and simpler definition of the operation of expansion and the description of the structure of principal types.

1 Introduction

In the approach of untyped λ-calculus as a model of programming languages, Curry's type system is the basis of type systems of programming languages like ML. Indeed, Curry's type system has the principal type property *i.e.*, for each typable λ-term there exists a type, the principal type, from which we can find all possible types for this term. However, this type system has some limitations: polymorphic abstractions are not allowed and types are not preserved under β-conversion.

To supply a type system that does not have these drawbacks, the intersection type discipline has been developed. Using intersection types, terms and term variables can have more than one type. This allows polymorphic abstraction, and types are invariant under β-conversion of terms i.e., two λ-terms which are β-equivalent have the same type. Moreover, intersection types characterize normalizable λ-terms: a term is normalizable if and only if it is typable. Intersection type systems are therefore very expressive.

However, the price of this expressiveness is the loss of the principal type property in the classical sense. As a matter of fact, in order to find all possible types of a term from a unique type, we must have more than just substitutions. In [2, 4, 7, 8] a property which is similar to the principal type property is proved by adding new operations on types. The most important and the most complex of these operations is the expansion which is essential for the type inference. In fact, expansion is a complex operation on pairs. As S. van Bakel explains in [7], the expansion of a sub-term ρ of a type ρ' replaces the occurrences of ρ in ρ' by a number of copies of that sub-term. To be applied an expansion must therefore specify the type to be expanded and the number of necessary copies.

* Email: {Emilie.Sayag,Michel.Mauny}@inria.fr.

Intuitively, expansion corresponds to the duplication of a sub-derivation in a derivation tree. So it is not enough to duplicate one type: we must also copy all the types of this sub-derivation. Until now, this point was the source of the complexity of the definitions of expansion [2, 4, 3, 7]. Even if the need of duplicating more than one type is well understood, the definition of the set of types to be copied, is still a difficult problem. So far, no convincing justification has been given.

In this paper, in order to fill this gap, we propose a new approach to intersection types. The work presented here is based on the intersection type system introduced in [5]. This type system is a restriction of the one presented in [1] in the sense that intersections occur only in the left hand side of arrow types. In [5], we have defined a new notion of principal type, corresponding exactly to the notion of normal form in the λ-calculus. We now extend this notion to all normalizable λ-terms and using the structural properties of principal types that we proved in [5], we give a simpler definition of the expansion operation than the one proposed in [2, 3, 7, 8], and a simpler proof of the existence of a principal type for each normalizable λ-term.

The general outline of this paper is as follows: in section 2, we recall the type system of [5] and its main properties. In section 3, we define the operation of expansion and we give some of its properties. The main result of section 4 states the principal type property for normalizable terms and section 5 gives an overview of the related works. Finally, section 6 contains a few concluding remarks.

2 The type theory

For more details about this section, one can see [5]. We recall here only the main definitions and properties. The set of types is defined as the following:

$$\begin{array}{rll} \rho \in \mathcal{T} ::= & \alpha & \text{type variable} \\ & \mid [\rho_1, \ldots, \rho_n] \rightarrow \rho \text{ for } n \geq 0 & \end{array}$$

We assume a countably infinite set TV of type variables.

Definition 1. We define the *positive* and *negative occurrences* of a type variable α in a type ρ by induction on the structure of ρ in the following way:

- if ρ is a type variable, then the possible occurrence of α in ρ is positive
- if $\rho = [\rho_1, \ldots, \rho_n] \rightarrow \rho'$, then the positive (resp. negative) occurrences of α in ρ are the positive (resp. negative) occurrences of α in ρ' and if $n \geq 1$, the negative (resp. positive) occurrences of α in ρ_i for $i = 1, \ldots, n$.

Definition 2. Let ρ be a type in $\mathcal{T}$ and α a type variable. We say that α has a *final occurrence* in ρ if one of the following cases is verified:

- $\rho = \alpha$
- $\rho = [\rho_1, \ldots, \rho_n] \rightarrow \rho'$ and α has a final occurrence in ρ'.

$$\vdash x : \rho \,;\, \{x : [\rho]\} \qquad \text{(VAR)}$$

$$\frac{\vdash e_1 : \rho_1 \,;\, A_1}{\vdash \lambda x.e_1 : A_1(x) \to \rho_1 \,;\, A_1 \setminus \{x\}} \qquad \text{(ABS)}$$

$$\frac{\vdash e_1 : [\rho_2^1, \ldots, \rho_2^n] \to \rho_1 \,;\, A_1 \quad \vdash e_2 : \rho_2^1 \,;\, A_2^1 \ldots \vdash e_2 : \rho_2^n \,;\, A_2^n}{\vdash e_1\ e_2 : \rho_1 \,;\, A_1 + A_2^1 + \ldots + A_2^n} \quad (n \geq 0) \qquad \text{(APP)}$$

Fig. 1. Inference rules

Definition 3. Let $\rho \in \mathcal{T}$, the set $L_0(\rho)$ of *left sub-terms* of ρ is defined by induction on the structure of ρ, in the following way:

- if $\rho = \alpha$, $L_0(\rho) = \emptyset$
- if $\rho = [\rho_1, \ldots, \rho_n] \to \rho'$, $L_0(\rho) = \{\rho_1, \ldots, \rho_n\} \cup L_0(\rho')$.

We also define a mapping *TypeVar* from types to sets of type variables. This function returns the set of type variables which occur in a type.

Definition 4. A *constraint environment* A, is a mapping from the set $\mathcal{V}$ of term variables to the set of multi-sets of types.

As usual, we can restrict the domain of a constraint environment:

$$A \setminus \{x\}(y) = \begin{cases} A(y) & \text{if } y \neq x \\ [\,] & \text{otherwise} \end{cases}$$

and extend it: $(A_1 + A_2)(x) = A_1(x) \cup A_2(x)$, for all $x \in \mathcal{V}$
where $\cup$ is the union of multi-sets.

Remark. We use metavariables $x, y, \ldots$ to denote term variables and $\alpha, \beta, \gamma, \ldots$ for type variables.

The type assignment relation relating λ-terms, types and constraint environments, is defined in figure 1. We write the constraint environnement A at the right of the relation symbol to insist on the fact that A is computed during type inference instead of being a simple argument as it is in more classical systems. We notice that in the rule for applications, if $n = 0$ then there is only one premisse in that inference rule and the argument of the application does not interfere in the derivation. As an example, we can derive:

$$\vdash \lambda x.\lambda y.x(y\ x) : [\alpha, [\beta] \to \gamma] \to [[\alpha] \to \beta] \to \gamma \,;\, \{\,\}$$

The type inference algorithm for normal forms is presented in figure 2.

$Infer(N) =$
- **Case** $N = x$
  let α be a new type variable
  return $(\alpha, \{x : [\alpha]\})$
- **Case** $N = \lambda x.N_1$
  let $(\rho_1, A_1) = Infer(N_1)$
  return $(A_1(x) \rightarrow \rho_1, A_1 \setminus \{x\})$
- **Case** $N = x\ N_1\ \ldots\ N_n$
  let $(\rho_1, A_1) = Infer(N_1)$
  $\vdots$
  $(\rho_n, A_n) = Infer(N_n)$
  α be a new type variable
  return $(\alpha, \{x : [[\rho_1] \rightarrow \cdots \rightarrow [\rho_n] \rightarrow \alpha]\} + A_1 + \cdots + A_n)$

Fig. 2. Type inference algorithm for normal forms

3 B-types

We now study the structure of pairs which are closed under expansions.

We give mutually recursive definitions of $\mathcal{T}_{E_g}$ and $\mathcal{T}_g$. $\mathcal{T}_{E_g}$ is the set of the type constraints of term variables, that is the set of types which occur in constraint environnements or in the left hand side of arrow types. $\mathcal{T}_g$ is the set of types of λ-terms, that is the set of types which occur in the right hand side of arrow types.

$$\nu \in \mathcal{T}_{E_g} ::= \alpha \mid [\mu_1, \ldots, \mu_n] \rightarrow \nu$$

with $n > 0$, $\forall i \in \{1, \ldots, n\}$, $\mu_i \in \mathcal{T}_g$, $TypeVar(\mu_i) \cap TypeVar(\nu) = \emptyset$ and $\forall j \in \{1, \ldots, n\}$ such that $j \neq i$, $TypeVar(\mu_i) \cap TypeVar(\mu_j) = \emptyset$

$$\mu \in \mathcal{T}_g ::= \alpha \mid [\nu_1, \ldots, \nu_n] \rightarrow \mu$$

with $n \geq 0$ and $\forall i \in \{1, \ldots, n\}$, $\nu_i \in \mathcal{T}_{E_g}$ rom now on, metavariables ν and μ denote elements of $\mathcal{T}_{E_g}$ and $\mathcal{T}_g$ respectively.

In the following, we always study pairs of types and constraint environnements. In order to easily handle these pairs, we define *B-types* from $\mathcal{T}_{E_g}$ and $\mathcal{T}_g$, in the following way:

$$U ::= [\nu_1, \ldots, \nu_n] \Rightarrow \mu \text{ with } n \geq 0$$

The term variables of constraint environnements disappear to simplify the notation and because they don't play a significant role in the following. We use a double arrow to link constraint environnements and types to highlight the similarity between the type constraints and the types on the left hand side of arrows in types. So we extend easily to B-types the notion of sign of an occurrence and *TypeVar*.

The notion of left sub-terms does not take into account the full recursive structure of a type. We now define a notion of *generalized left sub-terms*, following

the recursive structure of types to consider all possible sub-terms which are to the left of an arrow at any level in the recursive structure of a type.

Definition 5. Let U be a B-type, we define the set $\mathcal{L}(U)$ of *generalized left sub-terms* of U in the following way:

- $L_0(U) = \{\nu_1, \ldots, \nu_n\} \cup L_0(\mu)$ if $U = [\nu_1, \ldots, \nu_n] \Rightarrow \mu$
- $\forall n > 0,\ L_n(U) = \bigcup_{\rho \in L_{n-1}(U)} L_0(\rho)$
- $\mathcal{L}(U) = \bigcup_{n \geq 0} L_n(U)$

Definition 6. A B-type U is *closed* if each type variable of $TypeVar(U)$ has exactly one positive occurrence and one negative occurrence in U.

Definition 7. Let $U = [\nu_1, \ldots, \nu_n] \Rightarrow \mu$ be a B-type. U is *finally closed* if the variable α in the final occurrence of μ is also in the final occurrence of a type which is element of $L_0(U)$.

Definition 8. Let U be a B-type. U is *minimally closed* if there is no closed B-type strictly held in U.

The following definition gives a short way to talk about the three previous properties simultaneously.

Definition 9. Let $U = [\nu_1, \ldots, \nu_n] \Rightarrow \mu$ be a B-type. We say that U is *complete* if U is closed, finally closed and minimally closed.

Definition 10. We say that U is a *ground B-type* if U is complete and if it is one of the following forms:

- $U = [\rho] \Rightarrow \rho$ with $\rho \in \mathcal{T}_0 \cap \mathcal{T}_{E_\lambda}$.
- $U = [\nu_1, \ldots, \nu_n] \Rightarrow \alpha$ and $\exists i \in \{1, \ldots, n\}$ such that ν_i has the following shape

$$[\mu_1^1, \ldots, \mu_1^{n_1}] \to \cdots \to [\mu_p^1, \ldots, \mu_p^{n_p}] \to \alpha$$

  with $p > 0$, and $\exists (E_j^k)_{j=1\ldots p, k=1\ldots n_j}$, a partition of $[\nu_1, \ldots, \nu_{i-1}, \nu_{i+1}, \ldots, \nu_n]$ such that each $E_j^k \Rightarrow \mu_j^k$ is a ground B-type.
- $U = [\nu_1, \ldots, \nu_n] \Rightarrow [\nu_{n+1}, \ldots, \nu_{n+p}] \to \mu'$ with $[\nu_1, \ldots, \nu_{n+p}] \Rightarrow \mu'$ a ground B-type.

Remark. The partition $(E_j^k)_{j=1\ldots p, k=1\ldots n_j}$ is unique. Since U is closed, each type variable has only two occurrences in U and we have no choice on the definition of each E_j^k.

Example 1. $[\epsilon, [\epsilon] \to [\beta] \to \delta, [\alpha] \to \beta, \alpha, \epsilon] \Rightarrow \delta$ is not a closed B-type, but $[[\gamma_1, \gamma_2] \to [\beta] \to \delta, [\alpha] \to \beta, \alpha, \gamma_1, \gamma_2] \Rightarrow \delta$ is a ground B-type. $[[[\alpha, [\alpha] \to \beta] \to \beta] \gamma, [\delta] \to [\delta]] \Rightarrow \gamma$ is closed, finally closed but not minimally closed.

$Clos(\mu, U) =$
- **Case** $U = [\rho] \Rightarrow \rho$
  return $[\rho]$
- **Case** $U = [\nu_1, \ldots, \nu_n] \Rightarrow \alpha$
  let $i \in \{1, \ldots, n\}$ such that $\nu_i = [\mu_1^1, \ldots, \mu_1^{n_1}] \rightarrow \cdots \rightarrow [\mu_p^1, \ldots, \mu_p^{n_p}] \Rightarrow \alpha$
  let $(E_j^k)_{j=1,\ldots,p,k=1,\ldots,n_j}$ the partition of $[\nu_1, \ldots, \nu_{i-1}, \nu_{i+1}, \ldots, \nu_n]$
  such that $\forall j \in \{1, \ldots, p\}, \forall k \in \{1, \ldots, n_j\}$, $E_j^k \Rightarrow \mu_j^k$ is a ground B-type
  if $\exists j, k$ such that $\mu = \mu_j^k$
  then return E_j^k **else**
  if $\exists j, k$ such that $\mu \in \mathcal{L}(E_j^k \Rightarrow \mu_j^k) \cap \mathcal{T}_{E_g}$
  then return $Clos(\mu, E_j^k \Rightarrow \mu_j^k)$
  else fail
- **Case** $U = [\nu_1, \ldots, \nu_n] \Rightarrow [\nu_{n+1}, \ldots, \nu_{n+m}] \rightarrow \mu'$
  return $Clos(\mu, [\nu_1, \ldots, \nu_{n+m}] \Rightarrow \mu')$

Fig. 3. Closure algorithm

In order to define the expansion operation, we need to describe several further notions and prove some properties about the structure of ground B-types. The complexity of the expansion operation comes from the definition of the set of types that the expansion must duplicate. The expansion operation corresponds to the duplication of the typing derivation of a sub-term in a derivation tree. So all types of this sub-derivation must be duplicated.

The contribution of this section is precisely the definition of this problematic set of types. The justification of this definition is obvious according to the previous results about the structure of principal types [5] and B-types.

We define in figure 3 an algorithm constructing the multi-set of types that we must duplicate when we expand a type.

Lemma 11. *Let U be a ground B-type and $\mu \in \mathcal{L}(U) \cap \mathcal{T}_g$. $Clos(\mu, U)$ is well-defined and verifies the following conditions:*

- *$Clos(\mu, U) \subset \mathcal{L}(U) \cap \mathcal{T}_{E_g}$*
- *$Clos(\mu, U) \Rightarrow \mu$ is a ground B-type*
- *$Clos(\mu, U)$ is the unique sub-multi-set of $\mathcal{L}(U) \cap \mathcal{T}_{E_g}$ which verifies the previous condition.*

An expansion makes a number of copies of several types. We want each copy of a type to be disjoint from all others, *i.e.* two copies of the same type have no common type variables. In order to be precise, we define specific substitutions which will make the copies of types exactly as we need.

Definition 12. Let S be a substitution, we say that S is a *renaming substitution* if for all $\alpha \in Dom(S)$, $S(\alpha) = \beta$ where β is a type variable and S is injective on its domain. Furthermore, if $Range(S)$ is a set of new type variables, we say that S is a *fresh renaming substitution.*

Definition 13. Let p be an integer. For all types μ in $\mathcal{T}_g$ we define an operation of *expansion* of μ, on the ground B-type U, written $E_{(p,\mu)}$, by:

$$E_{(p,\mu)}(U) = \begin{cases} U & \text{if } \mu \notin \mathcal{L}(U) \\ U' & \text{otherwise} \end{cases}$$

where U' is obtained from U by replacing each occurrence of an element ν of $Clos(\mu, U)$ by $R_1(\nu), \ldots, R_p(\nu)$ and μ by $R_1(\mu), \ldots, R_p(\mu)$, if $R_1, \ldots, R_p$ are p fresh renaming substitutions of domain $TypeVar(Clos(\mu, U))$.

We remark that since the renaming substitutions $R_1, \ldots, R_p$ are not unique, the expansion of μ in U is defined up to a renaming. Moreover, we make no hypothesis on p. If $p = 0$, the expansion $E_{(p,\mu)}$ removes all the occurrences of μ and of the elements of its closure.

Example 2. $E_{(2,\gamma)}([[\gamma] \to [\beta] \to \delta, [\alpha] \to \beta, \alpha, \gamma] \Rightarrow \delta) = [[\gamma_1, \gamma_2] \to [\beta] \to \delta, [\alpha] \to \beta, \alpha, \gamma_1, \gamma_2] \Rightarrow \delta$

Since our work is essentially based on the structure of types, we want to prove that expansions do not change this structure. So we prove that the set of ground pairs is closed under expansion.

Lemma 14. *Let U be a ground B-type, $\mu \in \mathcal{T}_g$, and p an integer. Then $E_{(p,\mu)}(U)$ is a ground B-type.*

4 Principal typing of normalizable λ-terms

This section states the existence of principal types for all normalizable λ-terms in corollary 17. The interest of this section is not the result itself, but its proof which is conceptually much simpler than the proofs in [2, 4, 7, 8]. Here we are only interested in normalizable λ-terms. Thus, thanks to the stability of typing under β-conversion [6], it is enough to use normal forms. We do not need to introduce approximants which significantly simplifies the proofs.

The substitution and expansion operations are both necessary to find a possible pair for a normal form from its principal pair. However these operations must be applied in an appropriate order.

Definition 15. We name *chain* a composition of substitutions, expansions and renaming substitutions, of the form $S_n \circ \ldots \circ S_1 \circ O_m \circ \ldots \circ O_1$ where S_i is a substitution for $i = 1, \ldots, n$ and O_j is either a renaming substitution or an expansion for $j = 1, \ldots, m$.

Theorem 16. *Let N be a term in normal form such that $\vdash N : \mu ; A$. If $Infer(N) = (\mu_p, A_p)$ then there exists a chain C such that $C(\overline{A_p} \Rightarrow \mu_p) = \overline{A} \Rightarrow \mu$.*

Proof. By induction on the structure of N.

- If $N = x$, by definition of *Infer* and the chain is only a substituion of a type variable by μ.
- If $N = \lambda x.N_1$, the induction hypothesis gives the chain C.
- If $N = x\ N_1 \ldots N_n$, we can compose the chains given by the induction hypothesis for each N_i.

Corollary 17. *Let e be a normalizable term such that $\vdash e : \mu : A$, N its normal form and $(\mu_p, A_p) = Infer(N)$. Then there exists a chain C such that $C(A_p \Rightarrow \mu_p) = A \Rightarrow \mu$.*

5 Related work

The authors of [2, 4, 7, 8] introduce a notion of *approximants*, also named *λ-Ω-normal forms*, and define principal typing for these extended normal forms before generalizing to λ-terms using an approximation property, i.e. $B \vdash e : \mu$ if and only if there exists an approximant a of e such that $B \vdash a : \mu$. S. Ronchi della Rocca proposed a semi-algorithm for type inference in [3]. These results give important theoretical benefits, but unfortunately, they provide a good understanding neither of the structure of principal types nor of their characteristic properties. Furthermore, the semi-algorithm proposed in [3] is not practical because of its conceptual complexity.

As far as we know, the work of S. van Bakel [7, 8] is the first real advance in the simplication of the presentation of the intersection type discipline since the initial presentations. Furthermore, in [7], S. van Bakel defines an intersection type system close to the one introduced in [1] with the same partial order relation on types. He studies the existence of principal types for this system. He was induced to define several sub-sets of the set of pairs of a type and a basis, ordered by inclusion. His *set of ground pairs* is equivalent to the set of ground B-types that we define and his *relevant* intersection type system, presented in [8], is pretty closed to the one presented here. In this work, we concentrate on practical aspects of intersection types, leaving out filter λ-models. This led us to study the relationship between principal intersection types and λ-normal forms [5] and to propose an operational definition of the expansion operation, while preserving the fundamental properties of intersection type systems: *all normalizable λ-terms have a principal type*.

6 Conclusion

In this article, we take advantage of the structural properties of intersection types to provide simple proofs of the principal type property for normalizable λ-terms.

We propose an intersection type system in which expansions and substitutions are enough to find all possible types of a normalizable λ-term from its principal type.

These results shed a new light on intersection typing , which can be presented through principal types isomorphic to λ-terms in normal form.

References

1. Henk P. Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
2. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ-calculus semantics. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda-calculus and Formalism*, pages 536–560. Academic Press, London, 1980.
3. Simona Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.
4. Simona Ronchi Della Rocca and Betti Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.
5. Émilie Sayag and Michel Mauny. Characterization of principal types of normal forms in an intersection type system. In V. Chandru and V. Vinay, editors, *Proceedings of Sixteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
6. Émilie Sayag and Michel Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical Report RR-2998, INRIA, 1996. available on our web site http://pauillac.inria.fr/~sayag/rr.html.
7. Steffen van Bakel. Principal type schemes for strict type assignment system. *Logic and Computation*, 3(6):643–670, 1993.
8. Steffen van Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151:385–435, 1995.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 177–181

The completeness theorem for a temporal logic with probabilistic operators

Zoran Ognjanović[1] and Miodrag Rašković[2]

[1] Matematički institut SANU
Kneza Mihaila 35, 11000 Beograd, Jugoslavija
e-mail: zorano@mi.sanu.ac.yu
[2] Prirodno-matematički fakultet
R. Domanovića 12, 34000 Kragujevac, Jugoslavija
e-mail: miodragr@mi.sanu.ac.yu

Abstract. A temporal logic augmented with a countable set of probabilistic operators is presented. The corresponding temporal models are described. An axiomatization of this logic is given. The corresponding completeness theorem is proved.

1 Introduction

Temporal logic is often used with a fixed semantics or "flow of time" (see [1] for further references). A common choice is the natural numbers. Then, a completeness theorem is of the following form "given this axiom system every theory that is consistent with it has a model with flow of time isomorphic to the natural numbers". Temporal logics also have different syntax. It may be varied by allowing different temporal connectives. Probabilistic logics (see [2], [3], [5], [4]) are conservative extensions of the classical logics that are suitable for explicit reasoning about probabilities.

In this paper we will discuss a propositional logic which language consists of temporal and probabilistic operators. In this logic we can analyse how probabilities of some events will change in the future. We will consider a class of models, with the time-line isomorphic to the natural numbers, where every instant of time in every model has a positive probability measure. We will give an axiom system and prove the corresponding completeness theorem.

2 Syntax and Semantics

Let $\phi = \{p, q, r, \ldots\}$ be a set of primitive propositions, and S the set of rational numbers from $[0, 1]$. The well-formed formulas are built up from the primitive

propositions using negation ($\neg$), conjunction ($\wedge$), nexttime-opertator ($\bigcirc$) and a probabilistic operator ($P_{\geq s}$) for every $s \in S$. If $T = \{\alpha_1, \ldots, \alpha_n\}$ is a finite set of formulas $\wedge T$ denotes $\alpha_1 \wedge \ldots \wedge \alpha_n$.

A model is a tuple $M = \langle W, v, \mu \rangle$, where $W = \{w_0, w_1, \ldots\}$ is an infinite sequence of worlds (or moments), $v : W \times \phi \mapsto \{\top, \bot\}$ is a propositional valuation, and μ associates with every $w \in W$ a finitely additive probabilistic measure $\mu(w) : 2^W \mapsto [0, 1]$ which satisfies $\mu(w_i)(w_j) > 0$ iff $j > i$. We will alternatively write $w + i$ to denote w_i. It follows from the above definition that from a particular world all worlds in the future and only they have positive probabilistic measures.

The satisfiability of a formula α in a world w from a model M is defined by:

a) if $\alpha \in \phi$, $w \| \!\!- \alpha$ iff $v(w)(\alpha) = \top$,
b) $w \| \!\!- P_{\geq s}\alpha$ iff $\mu(w)(\{w + i, i > 0 : w + i \| \!\!- \alpha\}) \geq s$,
c) $w \| \!\!- \bigcirc\alpha$ iff $w + 1 \| \!\!- \alpha$,
d) $w \| \!\!- \neg\alpha$ iff it is not $w \| \!\!- \alpha$ and
e) $w \| \!\!- \alpha \wedge \beta$ iff $w \| \!\!- \alpha$ and $w \| \!\!- \beta$.

It is easy to see that $\bigcirc\alpha$ holds in a w if α will hold in the next moment. The intuitive meaning of $P_{\geq s}\alpha$ is that the probability of α is greater or equal to s. In our logic, the well known temporal operator G, 'it is always going to be the case that', can be defined by $P_{\geq 1}$. Some other probabilistic operators can also be defined: $P_{<s}\alpha$ by $\neg P_{\geq s}\alpha$, $P_{\leq s}\alpha$ by $P_{\geq 1-s}\neg\alpha$, $P_{>s}\alpha$ by $\neg P_{\leq s}\alpha$, and $P_{=s}\alpha$ by $P_{\geq s}\alpha \wedge \neg P_{>s}\alpha$. We abbreviate $\bigcirc^0\alpha$ by α, and $\bigcirc^{n+1}\alpha$ by $\bigcirc \bigcirc^n \alpha$.

3 An axiom system

The above class of models is characterized by the following set of axioms:

1. all instances of propositional theorems
2. $\bigcirc(\alpha \rightarrow \beta) \rightarrow (\bigcirc\alpha \rightarrow \bigcirc\beta)$
3. $\neg \bigcirc \alpha \leftrightarrow \bigcirc\neg\alpha$
4. $P_{\geq 1}\alpha \rightarrow \bigcirc\alpha \wedge \bigcirc P_{\geq 1}\alpha$
5. $P_{\geq 0}\alpha$
6. $P_{\geq t}\alpha \rightarrow P_{\geq s}\alpha$, $t > s$
7. $P_{\leq s}\alpha \rightarrow P_{<t}\alpha$, $t > s$
8. $(P_{\geq s}\alpha \wedge P_{\geq r}\beta \wedge P_{\geq 1}(\neg\alpha \vee \neg\beta)) \rightarrow P_{\geq min(1, s+r)}(\alpha \vee \beta)$
9. $(P_{\leq s}\alpha \wedge P_{<r}\beta) \rightarrow P_{<min(1,(s+r))}(\alpha \vee \beta)$
10. $P_{<s}\alpha \rightarrow P_{\leq s}\alpha$
11. $P_{\geq 1}(\alpha \rightarrow \beta) \rightarrow (P_{\geq s}\alpha \rightarrow P_{\geq s}\beta)$
12. $\bigcirc\alpha \rightarrow P_{>0}\alpha$
13. $\bigcirc P_{>0}\alpha \rightarrow P_{>0}\alpha$

and rules:

1. from $\{\alpha, \alpha \rightarrow \beta\}$ infer β
2. from α infer $\bigcirc\alpha$

3. from $\{\beta \rightarrow \bigcirc^{n+m}\alpha$ for an arbitrary $n \geq 0$ and every $m > 0\}$ infer $\beta \rightarrow \bigcirc^n P_{\geq 1}\alpha$
4. from $\{\beta \rightarrow \bigcirc^n P_{\geq s-\frac{1}{k}}\alpha$, for every $n \geq 0$ and every $k \geq \frac{1}{s}\}$ infer $\beta \rightarrow \bigcirc^n P_{\geq s}\alpha$
5. from $\alpha \rightarrow \neg(\bigcirc\neg p \wedge \ldots \wedge \bigcirc^{n-1}\neg p \wedge \bigcirc^n p \wedge \bigcirc^n P_{\geq 1}\neg p)$ where $p \in \phi$ is not in α, $n > 0$ infer $\neg\alpha$.

A finite set of formulas T is consistent if $\not\vdash \neg(\wedge T)$. An infinite set of formulas is consistent if each of its finite subsets is consistent.

4 The completeness theorem

Theorem 1 (Soundness). *Every theorem of the above axiom system is valid in the described class of models.*

Proof. It is easy to see that every axiom is valid. The inference rules preserve validity. For instance, let

$$\alpha \rightarrow \neg(\bigcirc\neg p \wedge \ldots \wedge \bigcirc^{n-1}\neg p \wedge \bigcirc^n p \wedge \bigcirc^n P_{\geq 1}\neg p)$$

be a valid formula, where the primitive proposition p does not appear in α. Let $M = \langle W, v, \mu\rangle$ be a model, and $w \in W$. Let $M' = \langle W, v', \mu\rangle$, where $v'(u)(p) = \top$ iff $u = w + n$, and $v' = v$ for the other primitive propositions. Then, $\mu(u)(\{u + i, i > 0 : u + i \Vdash \neg p\}) = 0$, and $\mu(u)(\{u + i, i > 0 : u + i \Vdash \neg p\}) = 1$. It follows that

$$w \Vdash_{M'} \bigcirc\neg p \wedge \ldots \wedge \bigcirc^{n-1}\neg p \wedge \bigcirc^n p \wedge \bigcirc^n P_{\geq 1}\neg p$$

and consequently that $w \Vdash_{M'} \neg\alpha$. But, v' and v agree on all primitive propositions in α, so we have $w \Vdash_M \neg\alpha$. Since $\neg\alpha$ holds in an arbitrary world in an arbitrary model, it is valid.

Theorem 2 (Completeness). *Every consistent formula α has a model.*

Proof. Let $r_1, \ldots, r_n$ are all primitive propositions in α, and $q_1, q_2, \ldots$ an enumeration of all primitive propositions except $r_1, \ldots, r_n$. Let $\alpha_0, \alpha_1, \ldots$ be an enumeration of all formulas so that for every n, $\bigcirc\neg q_n \wedge \ldots \wedge \bigcirc^{n-1}\neg q_n \wedge \bigcirc^n q_n \wedge \bigcirc^n P_{\geq 1}\neg q_n$ appears in the sequence before any other formula that contains q_n.

We define a sequence of sets of formulas in the following way:

1. $T_0 = \alpha$
2. For every $i \geq 0$
   (a) if $T_i \cup \{\alpha_i\}$ is consistent, then $T_{i+1} = T_i \cup \{\alpha_i\}$, otherwise
   (b) if $\alpha = \gamma \rightarrow \bigcirc^n P_{\geq 1}\beta$ and $T_i \cup \{\alpha_i\}$ is not consistent, then $T_{i+1} = T_i \cup \{\gamma \rightarrow \bigcirc^n\neg\bigcirc^m\beta\}$, for an $m > 0$, so that T_{i+1} is consistent, otherwise
   (c) if $\alpha = \gamma \rightarrow \bigcirc^n P_{\geq s}\beta$ and $T_i \cup \{\alpha_i\}$ is not consistent, then $T_{i+1} = T_i \cup \{\gamma \rightarrow \bigcirc^n\neg P_{\geq s-\frac{1}{k}}\beta\}$, for a $k > \frac{1}{s}$, so that T_{i+1} is consistent, otherwise
   (d) $T_{i+1} = T_i$
3. $T = \cup_i T_i$

Every T_i is a consistent set. It is easy to see that, if T_i is obtained by the rules, $2.a$, $2.b$, and $2.d$. The same holds for the other rules as well. Suppose that there is no formula $\bigcirc^n\neg\bigcirc^m\beta$ so that $T_{i-1}\cup\{\bigcirc^n\neg\bigcirc^m\beta\}$ is consistent, then:

1. $\vdash\neg((\wedge T_{i-1})\wedge(\gamma\rightarrow\bigcirc^n P_{\geq 1}\beta))$, by hypothesis
2. $\vdash\neg((\wedge T_{i-1})\wedge(\gamma\rightarrow\bigcirc^n\neg\bigcirc^m\beta))$, for every $m>0$
3. $\vdash(\wedge T_{i-1})\rightarrow\neg(\gamma\rightarrow\bigcirc^n P_{\geq 1}\beta)$, from 1.
4. $\vdash(\wedge T_{i-1})\rightarrow\neg(\gamma\rightarrow\bigcirc^n\neg\bigcirc^m\beta)$, from 2., for every $m>0$
5. $\vdash(\wedge T_{i-1})\rightarrow(\gamma\rightarrow\neg\bigcirc^n\neg\bigcirc^m\beta)$, from 4., for every $m>0$
6. $\vdash(\wedge T_{i-1})\rightarrow(\gamma\rightarrow\bigcirc^n\bigcirc^m\beta)$, from 5. by the axiom 2, for every $m>0$
7. $\vdash((\wedge T_{i-1})\wedge\gamma)\rightarrow\bigcirc^n\bigcirc^m\beta$, from 6., for every $m>0$
8. $\vdash((\wedge T_{i-1})\wedge\gamma)\rightarrow\bigcirc^n P_{\geq 1}\beta$, from 7. by the inference rule 3
9. $\vdash(\wedge T_{i-1})\rightarrow(\gamma\rightarrow\bigcirc^n P_{\geq 1}\beta)$, from 8.
10. $\vdash\neg(\wedge T_{i-1})$ from 3. and 9.

a contradiction. The similar holds for $2.c$. Finally, if T is not consistent, there is a finite $T'\subset T$ so that $\vdash\neg(\wedge T')$. But, then there is at least a $T_i\supset T'$ that is also inconsistent.

T is a maximal set. For an arbitrary formula β, either β, or $\neg\beta$ is in T. If it does not hold, suppose $\beta=\alpha_m$, $\beta=\alpha_n$, and $k=max\{m,n\}$. Then $\vdash\neg((\wedge T_k)\wedge\beta)$ and $\vdash\neg((\wedge T_k)\wedge\neg\beta)$. But, then $\vdash\neg(\wedge T_k)$, a contradiction. For every n $\bigcirc\neg q_n\wedge\ldots\wedge\bigcirc^{n-1}\neg q_n\wedge\bigcirc^n q_n\wedge\bigcirc^n P_{\geq 1}\neg q_n$ belongs to T. It follows from

$\vdash(\wedge T_i)\rightarrow(\bigcirc\neg q_n\wedge\ldots\wedge\bigcirc^{n-1}\neg q_n\wedge\bigcirc^n q_n\wedge\bigcirc^n P_{\geq 1}\neg q_n)$, for some i
$\vdash\neg(\wedge T_i)$, by the inference rule 5

Similarly, it can be shown that T contains every theorem, and if $\bigcirc^n P_{\geq s-\frac{1}{n}}\beta\in T$, then $\bigcirc^n P_{\geq s}\beta\in T$.

Now, we will define the canonical model $M=\langle W,v,\mu\rangle$. $W=\{w_0,w_1,\ldots\}$, $w_0=T$, $w_{i+1}=\{\beta:\bigcirc\beta\in w_i\}$, $v(w)(p)=\top$ iff $p\in w$, $\mu(w)(\{w+i:i>0,\beta\in w+i\})=sup_r\{P_{\geq r}\beta\in w\}$. Every $w\in W$ is consistent. Otherwise, we have the smallest k so that w_k is not consistent:

$\vdash\neg(\wedge_i\beta_i)$, $\{\beta_1,\beta_2,\ldots\}\subset w_k$
$\vdash\bigcirc\neg(\wedge_i\beta_i)$
$\vdash\neg(\wedge_i\bigcirc\beta_i)$

and w_{k-1} is not consistent, a contradiction. Every $w\in W$ is maximal, as well. Otherwise, there are the smallest k and β so that $\beta,\neg\beta\notin w_k$. But, then $\bigcirc\beta,\neg\bigcirc\beta\notin w_{k-1}$, and w_{k-1} is not a maximal set.

The axioms and rules guarantee that μ is a finitely additive probabilistic measure, that $\mu(w)(w+i)>0$, and that M is a model. For example, from the axiom 4, it follows that μ is a nonnegative mapping. If $\vdash\beta$ is a theorem, then by the inference rules 2, $\vdash\bigcirc\beta$, $\vdash\bigcirc^2\beta,\ldots$. By the inference rules 3 we have $\vdash P_{\geq 1}\beta$. Then, by the construction of the set T and the definition of the measure μ, $\mu(w)(\{w+i,i>0\})=1$. The other properties of μ follows similarly. From the construction of the set T we conclude that for every i there is a primitive

proposition denoted q_i, so that $q_i \in w_0 + i$, and $q \notin w_0 + j$, for $j \neq i$. By the axioms 11 and 12, it follows that for every $w \in W$ $\mu(w)(w + i) > 0$.

Finally, we can prove by the induction on the complexity of the formulas that $\beta \in w$ iff $w||\!\!- \beta$, and particularly that α is satisfiable. For the primitive propositions, it follows from the construction. If $\beta = \neg\gamma$, we have $w||\!\!- \beta$ iff it is not $w||\!\!- \gamma$ iff $\gamma \notin w$. If $\beta = \gamma \wedge \delta$, we have $w||\!\!- \beta$ iff $w||\!\!- \gamma$ and $w||\!\!- \delta$ iff $\gamma \in w$ and $\delta \in w$ iff $\gamma \wedge \delta \in w$. If $\beta = \bigcirc\gamma$, we have $w||\!\!- \beta$ iff $w + 1||\!\!- \gamma$ iff $\gamma \in w + 1$ iff $\beta \in w$. If $\beta = P_{\geq s}\gamma$, let $w||\!\!- \beta$. Then $sup_r\{P_{\geq r}\gamma \in w\} = \mu(w)(\{w + i : i > 0, \beta \in w + i\}) \geq s$. From the construction of the set T, and by the axiom 5, we conclude $\beta \in w$. On the other hand, if $\beta \in w$, then $sup_r\{P_{\geq r}\gamma \in w\} = \mu(w)(\{w + i : i > 0, \beta \in w + i\}) \geq s$, and $w||\!\!- \beta$.

5 Some comments

In this paper we demand that from every world every its successor has a positive, but arbitrary measure. We can give complete axiomatization of logics with another requirements, for example: $\mu(w)(w + i) = c_i$, for an arbitrary sequence $\{c_1, c_2, \ldots\}$, so that $\sum_i c_i = 1$, or even stronger $\mu(w)(w + i) = \frac{1}{2^i}$.

Our choice of type of the measures imply that temporal operator G can be defined as $P_{\geq 1}$. On the other hand, we can allow a different situation, where $\mu(w)(w+i)$ can be 0. Then, G is an independent operator and we have to change the axiom system to obtain a complete axiomatization. For example, we have to add the following axiom:

$$G\alpha \rightarrow P_{\geq 1}\alpha$$

and the rule:

from $\beta \rightarrow \bigcirc^{n+m}\alpha$ for an arbitrary n and every $m > 0$ infer $\beta \rightarrow \bigcirc^n G\alpha$

References

1. Gabbay, D. M., Hodkinson, I. M., An axiomatization of the temporal logic with until and since over the real numbers, Journal of logic and computation, **2** (1990) 229–259
2. Fagin, R., Halper, J. Y., and Megiddo, N. (1988), A logic for reasoning about probabilities, Information and Computation **87** (1990) 78–128
3. Fagin, R., Halper, J. Y., Reasoning About Knowledge and Probability, Journal of the ACM, **2** (1994) 340–367
4. Ognjanović, Z., Rašković, M., A logic with higer order probabilities, Publication de l'Institut Math. (NS) vol **60** (74) (1996) 1–4
5. Rašković, M., Classical logic with some probability operators, Publication de l'Institut Math. (NS) vol **53** (67) (1993) 1–3

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 183–189

Generating the products, candidates for dividing the polynomial expressions

Miloš Racković

Faculty of Sciences, Institute of Mathematics,
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia

Abstract. Dividing the polynomial expressions into the products is a basic part of the algorithm for reducing the number of calculating operations in the analytical expressions of the mathematical models of different systems. On the basis of the mathematical background the data structures and the algorithm for dividing the polynomial expressions is developed. The central topic of paper is the algorithm for generating all products which are the candidates for dividing.

1 Introduction

Mathematical modelling of the complex systems generates the analytical expressions with great calculating complexity which should be reduced in order to obtain the efficient model. The process of reducing calculating complexity of the analytical expressions is one of the basic processes in the generation of complete mathematical model. In [1], the complete procedure for generating the symbolic models of the dynamics of complex robotic mechanisms is described and in [2] is given the structural system analysis of the reducing calculating complexity process.

All of the analytical expressions obtained in [1] are in polynomial form

$$Y = \sum_{i=1}^{I} k_i \cdot \prod_{l=1}^{L} x_l^{e_{il}} \tag{1}$$

where:

Y - is the variable to be calculated;

k_i - is a constant coefficient related to the i-th addend;

x_l - is one of the basic variables of the robotic system model represented by its name ($q, \dot{q}, \ddot{q}, \sin q, \cos q$, where q represents the degree of freedom). For each addend the same sequence of variables x_j, $j = 1, ..., L$ is used.

e_{il} - the exponent of the l-th variable of the i-th addend. The algorithm for forming the mathematical model ensures that each of the exponents is a nonnegative integer number.

The main task is to form the calculation graph for the chosen analytical expressions of the type (1), with the least number of mathematical operations. To obtain a maximal reduction in the number of mathematical operations in [1-5] is proposed the dividing of the expressions in the following form

$$Y = \sum_{w=1}^{W} (Y_{w1} \cdot Y_{w2}) + Y_{W+1} \tag{2}$$

where Y_{w1}, Y_{w2}, $w = 1, ..., W$ and Y_{W+1} are also the expressions of the type (1).

The expressions Y_{l1}, Y_{l2}, $l = 1, ..., W$ have two addends at least, and are determined in a way which maximises reduction of the number of mathematical operations. Y_{W+1} represents the remainder of the expression Y which can not be divide into products any more. After dividing the expressions into the products the monomial extraction algorithm is applied which forms the calculating graph for chosen expressions [1].

This paper gives mathematical basis for developing the algorithm for dividing the expressions into the products. The data structures and the algorithm for generating all possible products, candidates for dividing are described too.

¿From all candidates for dividing, these products are chosen which give the largest reduction in the number of calculating operations. Then, the dividing on chosen products is performed. This part of expressions dividing process is not the topic of this paper.

2 Mathematical Background

The concept of structural matrices was introduced in [6] to represent the analytical expressions of the robotic quantities.

Structural matrix S of the expression Y is represented with the vector of coefficients $K_S = \left[k_1^S, ..., k_I^S\right]^\top$, the vector of variables $X_S = \left[x_1^S, ..., x_L^S\right]^\top$ and the matrix of exponents

$$E_S = \begin{bmatrix} e_{11}^S & e_{12}^S & \dots & e_{1L}^S \\ e_{21}^S & e_{22}^S & \dots & e_{2L}^S \\ \dots & \dots & \dots & \dots \\ e_{I1}^S & e_{I2}^S & \dots & e_{IL}^S \end{bmatrix}.$$

The problem of dividing some expression into the products is analogous to the problem of finding the structural matrices A and B which satisfy the equation

$$A \cdot B = C \tag{3}$$

for given structural matrix C. This problem can be written by equations

$$E_A \cdot E_B = E_C \ ; \ K_A \cdot K_B = K_C \tag{4}$$

where E_A, E_B i E_C are the exponent matrices of the structural matrices A, B and C, and their dimensions are $I \times L$, $J \times L$ and $M \times L$ $(M = I \cdot J)$ respectively. K_A, K_B and K_C are the vectors of coefficients of the structural matrices A, B and C, and their dimensions are I, J and M respectively. The matrix E_C represents the exponents of the expression of the type (1) which we want to write in the form of product of the expressions of the same type. The matrices E_A and E_B are the unknowns in this equation. If the solution of the system exists, then the matrices E_A and E_B which satisfy the equation will represent the exponent matrix and vectors K_A and K_B will represent the vector of coefficients of the expressions which form the product.

Let us denote vectors $\left[e_{i1}^A, ..., e_{iL}^A\right]^\top$, $\left[e_{j1}^B, ..., e_{jL}^B\right]^\top$ and $\left[e_{m1}^C, ..., e_{mL}^C\right]^\top$ (rows of the matrices E_A, E_B and E_C) with e_i^A, e_j^B and e_m^C respectively. Now, the equations (4) can be written as the system in a vector form

$$e_i^A + e_j^B = e_m^C \;;\; k_i^A \cdot k_j^B = k_m^C \tag{5}$$
$$m = (i-1) \cdot J + j, \quad i = 1, ..., I; j = 1, ..., J,$$

where the addition of the vectors is defined in the usual way.

In [1, 4] has been shown that satisfying equations (6) for all different $m1$, $m2$, $m3$, $m4 \in \{1, ..., M\}$ gives the necessary and sufficient condition for existing the solution of the system (5).

$$e_{m1}^C - e_{m3}^C = e_{m2}^C - e_{m4}^C \;;\; \frac{k_{m1}^C}{k_{m3}^C} = \frac{k_{m2}^C}{k_{m4}^C} \tag{6}$$

When the conditions (6) are derived two rows from the matrix E_A and E_B, i.e. two elements of the vectors K_A and K_B, are chosen. Also, by choosing four rows from the matrix E_C, four elements of the vector K_C are chosen. This means that these four addends from the expression described by the exponent matrix E_C and the vector of coefficients K_C, are obtained by multiplying two expressions each containing two addends which are described by the chosen parts of the matrices E_A and E_B and vectors K_A and K_B. Thus every condition of the type (6) represent a "2×2" product.

If in the matrix E_C and in the vector of coefficients K_C exist N couples $(e_{n_1}^C, e_{n_2}^C)$, and $(k_{n_1}^C, k_{n_2}^C)$, $n_1, n_2 \in \{1, ..., M\}$, $n = 1, ..., N$ respectively, and the following equations hold

$$e_{1_1}^C - e_{1_2}^C = e_{2_1}^C - e_{2_2}^C = \ \ldots = e_{N_1}^C - e_{N_2}^C \;;\; \frac{k_{1_1}^C}{k_{1_2}^C} = \frac{k_{2_1}^C}{k_{2_2}^C} = \ \ldots = \frac{k_{N_1}^C}{k_{N_2}^C} \tag{7}$$

then this couples represent the "$2 \times N$" product in analogous way as in 2×2 case.

Let us take the two products with dimensions $2 \times N^1$ and $2 \times N^2$ respectively, which are described by the equations analogous to (7). By this, $2 \times N^1$ elements are chosen in the matrix E_C and in the vector K_C in the first case and $2 \times N^2$ elements in the second case. These elements represent the addends of the expression given by the structural matrix C. The sets of all indexes belonging to

the first elements in the couples of the first and the second product are denoted by

$$I_{1,N^1} = \{n_1^1 | n^1 = 1, ..., N^1\} \quad ; \quad I_{1,N^2} = \{n_1^2 | n^2 = 1, ..., N^2\}$$

respectively. The intersection of these two sets is calculated.

$$I_{1,N^1} \bigcap I_{1,N^2} = I_{1,N} = \{n_1 | n = 1, ..., N\}$$

The sets of these indexes, belonging to the second elements in the couples of products, which correspond to the first elements whose indexes are in set $I_{1,N}$ are denoted by

$$I_{2,N^1,I_{1,N}} = \{n_2^1 | n_1^1 \in I_{1,N}\} \quad ; \quad I_{2,N^2,I_{1,N}} = \{n_2^2 | n_1^2 \in I_{1,N}\}$$

respectively. If $I_{2,N^1,I_{1,N}} \bigcap I_{2,N^2,I_{1,N}} = \emptyset$ it is possible to construct the new product with dimension $3 \times N$ by joining together corresponding elements of two existing products. The new product is described by the N ordered triples $(e_{n_1}^C, e_{n_2}^C, e_{n_3}^C)$, and $(k_{n_1}^C, k_{n_2}^C, k_{n_3}^C)$ where $n_1 \in I_{1,N}$, $n_2 \in I_{2,N^1,I_{1,N}}$ and $n_3 \in I_{2,N^2,I_{1,N}}$. The products with greater dimensions are constructed in the analogous way.

This mathematical analysis serves as basis for constructing the algorithm for generating the products, candidates for dividing the expressions. The first step is to represent the expression by the structural matrix. The next step is generating the equations of the form (7) for different combinations of the addends and for N great as possible. Every equation represents the $2 \times N$ product. On the basis of starting products, we then construct all possible products with greater dimensions. By this procedure all products, candidates for dividing are generated.

3 Generating the Products, Candidates for Dividing

After representing the expression, which have to be divide into the products, by the structural matrix, we need to construct all possible products, candidates for dividing. The basic idea of the procedure for generating the candidates is in following. The couple of addends is chosen, for which the difference between the exponents and the quotient between the constant coefficients are calculated. Then, for all other couples of addends these quantities are checked and if both are the same (equation (7)) this couple is added in the product which already contains the starting couple. When we check the conditions for all couples, the new product is formed if it contains the two couples at least. This procedure is repeated for all possible couples of addends. We do not form the product for these couples which are already member of some other product because the transitivity law for equation tells us that the same product is to be generated. When we repeat the procedure for all couples, all possible products with dimension $2 \times N$ for different values $N \geq 2$ are generated and stored in the list of products. The data structure for representing the product is given in the pseudocode.

```
struct product {
  int m, n, matadd[m][n];
  struct product * nextpr; }
```

Dimension of the product is represented with m and n and equals $m \times n$. The matrix $matadd[m][n]$ represent indexes of the addends which forms the given product. In $nextpr$, the pointer on the next product in the list of products is given. Procedure $FormPro$ which performs generating of the products and their storing in the list of products is given in the pseudocode.

$FormPro(S)$ {
  for ($i1 = 1$; $i1 < I$; $i1++$) {
    for ($i2 = i1 + 1$; $i2 \leq I$; $i2++$) {
      if ($NotMemL(i1, i2, lispro)$) {

$MemPair(i1, i2, pr)$; $s = c^S_{i1} - c^S_{i2}$; $q = \dfrac{k^S_{i1}}{k^S_{i2}}$;

for ($i3 = 1$; $i3 < I$; $i3++$) {
  for ($i4 = i3 + 1$; $i4 \leq I$; $i4++$) {

if ($s == c^S_{i3} - c^S_{i4}$ && $q == \dfrac{k^S_{i3}}{k^S_{i4}}$) $MemPair(i3, i4, pr)$;

else if ($s == c^S_{i4} - c^S_{i3}$ && $q == \dfrac{k^S_{i4}}{k^S_{i3}}$) $MemPair(i4, i3, pr)$; } } }

$MemPro(pr, lispro)$; } } }

The procedure $NotMemL(i1, i2)$ checks if the given couple $(i1, i2)$ is already stored in the some product from the list $lispro$. The procedure $MemPair(i1, i2, pr)$ stores this couple in variable pr in which the complete product is stored. By procedure $MemPro(pr, lispro)$ product pr is stored in the list of products $lispro$.

The next step is generating the new products with greater dimensions by joining existing products in a way described in the previous section. The procedure $GenAllPro(lispro, arrpp, dap)$ performs generating of all possible products with dimension $m \times n$ for different m and n based on the products candidates which are stored in the list of products $lispro$. An array of pointers $arrpp$ with dimension dap is introduced, where every member of this array points on the first element in the list of products which has corresponding value for m. The procedure $GenAllPro$ is given in pseudocode.

```
GenAllPro(lispro, arrpp, dap) {
  p1 = lispro; arrpp[0] = lispro; dap = 1;
  while (p1- > nextpr! =NULL) {
    for (i = 0;i < dap;i++) {
      if (i == 0) p2 = p1- > nextpr;
      else p2 = arrpp[i];
      while (p2! =NULL) {
        if (NotMem(p1, p2) && ConPoss(p1, p2)) {
```

$p3 = GenPro(p1, p2, m)$; $InsPro(p3, m, arrpp, dap)$; }
$p2 = p2- > nextpr$; } }
$p1 = p1- > nextpr$; }
$JoinLis(lispro, arrpp, dap)$; }

The idea of the procedure is to match the different products, and if it's possible the new product is constructed by joining the two matched products. This new product is stored in the list of products on the corresponding place. The pointer $p1$ passes through the starting list of products, while the pointer $p2$ beside through the starting list passes through all new generated products. By this procedure, finding and performing all possible products joining is provided. The procedure $NotMem(p1, p2)$ checks if the product $p1$ is the member of the product $p2$ if the $p2$ is some of the new generated products. The procedure $ConPoss(p1, p2)$ checks when is possible to join these two products and procedure $GenPro(p1, p2, m)$ performs the joining if it's possible. The joining is performed by applying procedure described in section 2. The variable m returns the number of rows in the matrix of addends for the newly-formed product. The procedure $InsPro(p3, m, arrpp, dap)$ inserts the newly-formed product $p3$ in the list of products on which points the corresponding pointer from the array *arrpp*, depending on the value m. If $p3$ is the first product for this value of m, the new pointer is formed in the array *arrpp* which points on the new list of products. The dimension of the array, *dap* is incremented too. When we pass through all product from the starting list it is necessary to join together all list of products, pointed by pointers from the *arrpp*, in the one list of products *lispro*. The pointers from the array *arrpp* still point on the same products as before joining. This joining is performed by the procedure $JoinLis(lispro, arrpp, dap)$. In this way all products, candidates for dividing the expression are stored in the list of products *lispro*.

4 Conclusion

The reduction of calculating complexity of the analytical expressions, participating in the mathematical model of system, is necessary part of the mathematical modelling process in symbolic form. The aim of reducing the model calculating complexity is the need for calculating numerical values of the model quantities in the real-time regime.

In this paper, the first part of the algorithm for dividing polynomial expressions into the products is described, which is consisted of generating all the products, candidates for dividing the expressions. The generating procedure is obtained on the basis of detail mathematical analysis which gives the necessary and sufficient conditions for dividing the expressions into the products.

The basic characteristic of the generating procedure is that instead of searching among the products and checking the conditions for all products, this procedure implements given mathematical analysis in the two-step algorithm. First step comprises generating all products with dimension $2 \times n$, while the second

step performs the constructing of the new products by joining of the existing ones. In this way, the algorithm is obtained which has practical value for complex analytical expressions too.

References

1. Racković, M.: Generation of the Mathematical Models of Complex tic Mechanisms in Symbolic Form. Ph. D. thesis, University of Novi Sad, (1996)
2. Racković, M., Surla, D.: Reduction of the Computational Complexity of the Mathematical Models of the Dynamics of Robotic Mechanisms. 10^{th} International Conference on Industrial Systems IS'96, October 01-03, (1996), Novi Sad, (In Serbian) 217–222
3. Racković, M., Surla, D.: Data Structures and the Algorithms for Automatic Generating of the Robotic System Models. Journal for informatics, computer sciences and telecommunications of Yugoslav Informatic Society INFO, **3/95** (1995) (In Serbian) 4–6
4. Racković, M., Surla, D.: The Algorithms and Data Structures for Forming Symbolic Models of the Robotic Systems. FILOMAT (Niš) **9:3** (1995), 887–897
5. Surla, D., Racković, M.: Forming a Calculation Graph for the Polynomial Expressions. X Conference on Applied Mathematics PRIM'95, Budva (1995), 275–282
6. Vukobratović, M., Kirćanski, N.: Real-Time Dynamics of Manipulation Robots. Springer-Verlag, (1985)

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 191–197

A Characterization of Ellipses by Discrete Moments

Nataša Sladoje

University of Novi Sad, Faculty of Engineering,
Trg D. Obradovića 6, 21000 Novi Sad, Yugoslavia

Abstract. In this paper we prove that four integers are enough for the unique coding of ellipses with axes parallel to coordinate axes. This efficient and easily computable representation needs an asymptotically minimal number of bits and enables a constant time approximate reconstruction of digital ellipses. It is shown that the errors in estimating the half-axes and the center position of digitized ellipse tend to zero while the number of pixels per unit tends to infinity, which corresponds to the situation when the digital picture resolution increases.

1 Introduction

Among the most important problems considered in computer vision and image processing related to digital pictures analysis, there are recognition of the studied object, its efficient representing and finding the algorithm for recovering the object from its representation. Digital shapes which appear the most often in practice are digital straight lines and conic sections (in the Euclidean plane) and so-called surfaces of the second order (in the Euclidean space). Since the least squares method, which gives efficient representations for digital straight line segments ([2]), digital parabola segments ([3]) and digital plane segments ([1]), is not appropriate for representing ellipses, where it leads to nonlinear problems which require complex numerical techniques for the solution, in this paper we develop the idea of separating sets for proving one-to-one correspondence between digital shapes and their representations by a constant number of integers, introduced in [1].

In Section 2 we give an asymptotically optimal representation of digital ellipses, corresponding to the ellipses of the form $\left(\frac{x-a}{Ar}\right)^2 + \left(\frac{y-b}{Br}\right)^2 \leq 1$, $A, B \in R$, r is the number of pixels per unit (i.e. resolution), by four integers.

An efficient estimation of the original ellipse half-axes, as well as the coordinates of its center, is given in Section 3. It is shown that errors of that estimation tend to zero when the resolution of digital picture increases.

The numerical data (Table 1.) strongly confirm the theoretical results. For the ellipse (at random position) with the half-axis exceeding 1000, relative deviation in estimating the half-axes and the center position is less then 0.0003%.

2 Representation of Digital Ellipses

Consider an ellipse E in the Euclidean plane, defined by $\left(\frac{x-a}{Ar}\right)^2 + \left(\frac{y-b}{Br}\right)^2 \leq 1$. The ellipse E can be digitized by using digitizing method in which all the digital points (points with integer coordinates) in the ellipse are taken. In this way, we get the set of digital points, defined by $D(E) = \{(i,j) \mid B^2(i-a)^2 + A^2(j-b)^2 \leq A^2B^2r^2,\ i,j \text{ are integers}\}$, which will be considered as digital ellipse. $D(E)$ can be efficiently coded by four integer parameters:

- the number of points of $D(E)$, denoted by $R(E)$;
- the sum of x-coordinates of the points of $D(E)$, denoted by $X(E)$;
- the sum of y-coordinates of the points of $D(E)$, denoted by $Y(E)$;
- the sum of squares of the x-coordinates of the points of $D(E)$, denoted by $XX(E)$.

These parameters can easily be computed for any ellipse. Very important property of this code is that it provides one-to-one correspondence between the set of digital ellipses and the set of their representations. That is the main result of the paper and it is presented by the following statement:

Theorem 1. *Let $D(E_1)$ and $D(E_2)$ be two digital ellipses and let $(R(E_1), X(E_1), Y(E_1), XX(E_1))$ and $(R(E_2), X(E_2), Y(E_2), XX(E_2))$ be their representations, respectively. Then*

$$R(E_1) = R(E_2) \wedge X(E_1) = X(E_2) \wedge Y(E_1) = Y(E_2) \wedge XX(E_1) = XX(E_2)$$

is equivalent to

$$D(E_1) = D(E_2).$$

Proof. It is obvious that all the parameters $R(E)$, $X(E)$, $Y(E)$, $XX(E)$ are uniquely determined for a certain ellipse E, so $(D(E_1) = D(E_2)) \Rightarrow R(E_1) = R(E_2) \wedge X(E_1) = X(E_2) \wedge Y(E_1) = Y(E_2) \wedge XX(E_1) = XX(E_2)$ follows from the definitions.

The opposite direction of the statement will be proved by a contradiction. Let's assume that the relations $D(E_1) \neq D(E_2)$ and $R(E_1) = R(E_2)$, $X(E_1) = X(E_2)$, $Y(E_1) = Y(E_2)$, $XX(E_1) = XX(E_2)$ are satisfied. Then we have $\#(D(E_1) \setminus D(E_2)) = \#(D(E_2) \setminus D(E_1)) \neq 0$, and

$$\sum_{(x,y)\in D(E_1)\setminus D(E_2)} x = \sum_{(x,y)\in D(E_2)\setminus D(E_1)} x, \tag{1}$$

$$\sum_{(x,y)\in D(E_1)\setminus D(E_2)} y = \sum_{(x,y)\in D(E_2)\setminus D(E_1)} y, \tag{2}$$

$$\sum_{(x,y)\in D(E_1)\setminus D(E_2)} x^2 = \sum_{(x,y)\in D(E_2)\setminus D(E_1)} x^2. \tag{3}$$

Two ellipses, satisfying $\#(D(E_1) \setminus D(E_2)) = \#(D(E_2) \setminus D(E_1)) \neq 0$, can have zero, one, two, three or four intersection points. If they have less then four intersecting points, the sets $D(E_1) \setminus D(E_2)$ and $D(E_2) \setminus D(E_1)$ can be separated by the straight line $ax + by = c$, in such way that

$$\begin{aligned} &ax + by < c \text{ for } (x, y) \in D(E_1) \setminus D(E_2) \text{ and} \\ &ax + by > c \text{ for } (x, y) \in D(E_2) \setminus D(E_1) \end{aligned}$$

is satisfied. This gives:

$$\begin{aligned} c \cdot \#(D(E_1) \setminus D(E_2)) &= \sum_{(x,y) \in D(E_1) \setminus D(E_2)} c \\ &> a \cdot \left(\sum_{(x,y) \in D(E_1) \setminus D(E_2)} x \right) + b \cdot \left(\sum_{(x,y) \in D(E_1) \setminus D(E_2)} y \right) \\ &= a \cdot \left(\sum_{(x,y) \in D(E_2) \setminus D(E_1)} x \right) + b \cdot \left(\sum_{(x,y) \in D(E_2) \setminus D(E_1)} y \right) \\ &> \sum_{(x,y) \in D(E_2) \setminus D(E_1)} c = c \cdot \#(D(E_2) \setminus D(E_1)). \end{aligned}$$

The contradiction $c \cdot \#(D(E_1) \setminus D(E_2)) > c \cdot \#(D(E_2) \setminus D(E_1))$ finishes the proof in this case.

Let us suppose that E_1 and E_2 have four intersection points. Obviously, for two ellipses E_1 and E_2, given by the equations $E_1(x, y) = 0$ and $E_2(x, y) = 0$ respectively, the set of all conics $F_\lambda(x, y)$, $\lambda \in R$, containing all the intersection points of E_1 and E_2 is defined by the equation

$$F_\lambda(x, y) = E_1(x, y) + \lambda E_2(x, y) = ax^2 + by^2 + cx + dy + e = 0.$$

If parameter λ is chosen in such way that the coeficient of y^2 in the equation $F_\lambda(x, y) = 0$ is annulled, (the existance of four intersection points implies that the coeficient of x^2 will not be annulled then), we get:

1. if horizontal axes of E_1 and E_2 belong to different straight lines, the equation of the form $F_\lambda(x, y) = ax^2 + cx + dy + e = 0$, which is the equation of parabola with the axis parallel to y-axis. The fact that any two conics may have at most four intersection points (in other words, none of the existing four intersection points can be the point of contact of F_λ and E_1, or F_λ and E_2), implies that F_λ separates the sets $D(E_1) \setminus D(E_2)$ and $D(E_2) \setminus D(E_1)$. The conclusions follow analogously as in the case when the separator for the sets $D(E_1) \setminus D(E_2)$ and $D(E_2) \setminus D(E_1)$ is a straight line.
2. if horizontal axes of E_1 and E_2 belong to the same straight line, the equation of the form $F_\lambda(x, y) = ax^2 + cx + e = 0$, which is the equation of two straight lines of the form $x = x_1$ and $x = x_2$. Since these lines separate sets $D(E_1) \setminus D(E_2)$ and $D(E_2) \setminus D(E_1)$ in such way that the relation $ax^2 + cx + e < 0$ is satisfied for the points from one of the sets $D(E_1) \setminus D(E_2)$ and $D(E_2) \setminus D(E_1)$,

while the relation $ax^2 + cx + e > 0$ is satisfied for the points from another, the conclusions follow as in the previous case.

Obviously, $YY(E)$ can be used instead of $XX(E)$ for the representation of the ellipse E, where $YY(E)$ denotes the sum of squares of y-coordinates of the points of $D(E)$.

For convenience and without loss of generality, in the rest of the paper it will be assumed that all the digital points that appear have positive coordinates.

3 Estimation of the Original Ellipse from its Code

An important question is how efficiently an ellipse $E : \left(\frac{x-a}{Ar}\right)^2 + \left(\frac{y-b}{Br}\right)^2 \leq 1$ can be recovered from $(R(E), X(E), Y(E), XX(E))$-code of its digitization $D(E)$. In this section it will be shown that the proposed coding scheme enables an approximate reconstruction of the parameters a, b, $A \cdot r$ and $B \cdot r$ with errors tending to zero while $r \to \infty$.

A (k,l)-moment, denoted by $m_{k,l}(S)$ for a continuous shape S, in $2D$-space is defined by:

$$m_{k,l}(S) = \iint_S x^k y^l \, dx \, dy.$$

If S is a continuous ellipse E, given by the inequality $\left(\frac{x-a}{Ar}\right)^2 + \left(\frac{y-b}{Br}\right)^2 \leq 1$, then

$$m_{0,0}(E) = \pi \cdot r^2 \cdot A \cdot B, \quad m_{1,0}(E) = \pi \cdot a \cdot r^2 \cdot A \cdot B,$$

$$m_{0,1}(E) = \pi \cdot b \cdot r^2 \cdot A \cdot B \quad \text{and} \quad m_{2,0}(E) = A \cdot B \cdot r^2 \cdot \pi \left(\frac{A^2 \cdot r^2}{4} + a^2\right).$$

Thus, if the moments of the continuous ellipse E are known, E can be reconstructed easily. Namely,

$$a = \frac{m_{1,0}(E)}{m_{0,0}(E)}, \quad \text{and} \quad b = \frac{m_{0,1}(E)}{m_{0,0}(E)}, \quad \text{while}$$

$$A \cdot r = \frac{2}{m_{0,0}(E)} \cdot \sqrt{m_{2,0}(E) \cdot m_{0,0}(E) - (m_{1,0}(E))^2} \quad \text{and}$$

$$B \cdot r = \frac{(m_{0,0}(E))^2}{2 \cdot \pi \cdot \sqrt{m_{2,0}(E) \cdot m_{0,0}(E) - (m_{1,0}(E))^2}}.$$

Four integers, $R(E)$, $X(E)$, $Y(E)$ and $XX(E)$, appearing in the proposed characterization of digital ellipse $D(E)$, can be understood as discrete moments of that discrete shape. So, it is natural to expect that the digitization of the ellipse, defined by

$$\left(\frac{x - \frac{X(E)}{R(E)}}{\frac{2}{R(E)} \cdot \sqrt{XX(E) \cdot R(E) - (X(E))^2}}\right)^2 + \left(\frac{y - \frac{Y(E)}{R(E)}}{\frac{(R(E))^2}{2 \cdot \pi \cdot \sqrt{XX(E) \cdot R(E) - (X(E))^2}}}\right)^2 \leq 1,$$

can be a good approximation for the represented digital ellipse $D(E)$.

In order to estimate the half-axes and the center position of the original ellipse, we need asymptotical expressions for $R(E)$, $X(E)$, $Y(E)$ and $XX(E)$:

Theorem 2. *Let the digital ellipse $D(E)$ be the digitization of an ellipse E, given by the equation $\left(\frac{x-a}{A\,r}\right)^2 + \left(\frac{y-b}{B\,r}\right)^2 \leq 1$, where $a \geq A \cdot r$ and $b \geq B \cdot r$ is satisfied. Then the following asymptotical expresions hold:*

$$R(E) = A \cdot B \cdot r^2 \cdot \pi + \mathcal{O}(r); \tag{4}$$

$$X(E) = a \cdot A \cdot B \cdot r^2 \cdot \pi + \mathcal{O}(a \cdot r); \tag{5}$$

$$Y(E) = b \cdot A \cdot B \cdot r^2 \cdot \pi + \mathcal{O}(b \cdot r); \tag{6}$$

$$XX(E) = A \cdot B \cdot r^2 \cdot \pi \cdot \left(\frac{A^2 \cdot r^2}{4} + a^2\right) + \mathcal{O}(a^2 \cdot r). \tag{7}$$

Proof. For the asymptotical expresion (5), we have

$$\begin{aligned} X(E) &= \sum_{\substack{(i,j) \\ (\frac{i-a}{Ar})^2 + (\frac{j-b}{Br})^2 \leq 1}} i \;=\; \sum_{i=\lceil a-A\cdot r\rceil}^{\lfloor a+A\cdot r\rfloor} \sum_{j=\lceil b-\frac{B}{A}\sqrt{(A\cdot r)^2-(i-a)^2}\rceil}^{\lfloor b+\frac{B}{A}\sqrt{(A\cdot r)^2-(i-a)^2}\rfloor} i \\ &= 2\,\frac{B}{A} \sum_{i=\lceil a-A\cdot r\rceil}^{\lfloor a+A\cdot r\rfloor} i\,\sqrt{(A\cdot r)^2-(i-a)^2} + \sum_{i=\lceil a-A\cdot r\rceil}^{\lfloor a+A\cdot r\rfloor} \mathcal{O}(i) \\ &= 2\,\frac{B}{A} \int_{a-A\cdot r}^{a+A\cdot r} x\,\sqrt{(A\cdot r)^2-(x-a)^2}\,d(\lfloor x\rfloor) + \mathcal{O}\left(\sum_{i=\lceil a-A\cdot r\rceil}^{\lfloor a+A\cdot r\rfloor} i\right) \\ &= a \cdot A \cdot B \cdot r^2 \cdot \pi + \mathcal{O}(a \cdot r). \end{aligned}$$

The relations (4), (6) and (7) can be proved analogously.

The optimality of the proposed code is a consequence of the above theorem:

Corollary 3. *The proposed $(R(E), X(E), Y(E), XX(E))$-code requires an asymptotically optimal (minimal) number of bits.*

Proof. Under the assumptions of Theorem 2, the number of bits required for the numbers $R(E)$, $X(E)$, $Y(E)$ and $XX(E)$ is

$$\mathcal{O}(\log(r^2) + \log(a \cdot r^2) + \log(b \cdot r^2) + \log(a^2 \cdot r^2)) = \mathcal{O}(\log(\max\{a, b\})).$$

On the other side, a trivial lower bound on the number of different digital ellipses which can be inscribed into an integer grid of size $(a + A \cdot r) \times (b + B \cdot r)$ is $(a + A \cdot r) \cdot (b + B \cdot r)$. So, the number of bits, required for the unique coding of digital ellipses, is at least $\log((a + A \cdot r) \cdot (b + B \cdot r)) = \mathcal{O}(\log(\max\{a, b\}))$. Since the lower bound is reached, the proposed code is optimal.

Now, we can give an upper bound on the precision in estimating the half-axes and the center of an original ellipse from its code.

Theorem 4. *Let the digital ellipse $D(E)$ be the digitization of an ellipse E, given by the equation $\left(\frac{x-a}{Ar}\right)^2+\left(\frac{y-b}{Br}\right)^2 \leq 1$. Then the following error estimations hold:*

$$\frac{\frac{X(E)}{R(E)}}{a}-1=\mathcal{O}\left(\frac{1}{r}\right), \quad \frac{\frac{Y(E)}{R(E)}}{b}-1=\mathcal{O}\left(\frac{1}{r}\right),$$

$$\frac{\frac{2}{R(E)}\cdot\sqrt{XX(E)\cdot R(E)-(X(E))^2}}{A\cdot r}-1=\mathcal{O}\left(\frac{1}{r}\right), \tag{8}$$

$$\frac{\frac{(R(E))^2}{2\cdot\pi\cdot\sqrt{XX(E)\cdot R(E)-(X(E))^2}}}{B\cdot r}-1=\mathcal{O}\left(\frac{1}{r}\right). \tag{9}$$

Proof. The first two relations follow by applying Theorem 2. For proving the relations (8) and (9), let's notice that the proposed approximate value for the length of corresponding half-axes of E end E', where E' is obtained by translating E for $-M_x(E)$ in horizontal direction and $M_x(E)$ denotes the minimal abscissa of the points belonging to $D(E)$, is the same. Namely,

$$R(E)=R(E'), \quad X(E)=\lceil a-A\cdot r\rceil\cdot R(E')+X(E')$$

and $XX(E)=(\lceil a-A\cdot r\rceil)^2\cdot R(E')+2\cdot\lceil a-A\cdot r\rceil\cdot X(E')+XX(E')$,

which implies that the equalities

$$\frac{2}{R(E)}\cdot\sqrt{XX(E)\cdot R(E)-(X(E))^2}=\frac{2}{R(E')}\cdot\sqrt{XX(E')\cdot R(E')-(X(E'))^2}$$

and

$$\frac{(R(E))^2}{2\cdot\pi\cdot\sqrt{XX(E)\cdot R(E)-(X(E))^2}}=\frac{(R(E'))^2}{2\cdot\pi\cdot\sqrt{XX(E')\cdot R(E')-(X(E'))^2}}$$

are satisfied. So, we can assume that $(A\cdot r-1)<a\leq A\cdot r$. By applying Theorem 2, the relations (8) and (9) follow.

The previous theorem shows that the errors in estimating the half-axes and the center position of the original ellipse from the corresponding digital data tend to zero while $r\to\infty$. The experimental results are in accordance with the theoretical result; errors in estimating the coordinates of the center, $\tilde{a}$ and $\tilde{b}$, and the half-axes, $\widetilde{Ar}$ and $\widetilde{Br}$, reconstructed from the code of the ellipse $\left(\frac{x-a}{A\,r}\right)^2+\left(\frac{y-b}{B\,r}\right)^2\leq 1$, are presented in Table 1:

Ar	Br	a	b	$\frac{\widetilde{Ar}}{Ar}-1$	$\frac{\widetilde{Br}}{Br}-1$	$\left\|\frac{\tilde{a}}{a}-1\right\|$	$\left\|\frac{\tilde{b}}{b}-1\right\|$
46.5	7.8	73.6	19.8	0.0050508	0.0106625	0.0002302	0.0035037
		7733.7	199.9	0.0016194	0.0107781	0.0000051	0.0001901
	37.8	733.7	99.6	0.0001912	0.0011011	0.0000414	0.0000494
		77337.3	1991.1	0.0004201	0.0007779	0.0000001	0.0000066
445.6	27.7	3388.8	91.1	0.0001253	0.0004362	0.0000077	0.0001822
		3888.3	991.1	0.0001532	0.0004898	0.0000116	0.0000164
	227.7	8883.3	911.9	0.0000154	0.0000305	0.0000011	0.0000044
		888338.8	11999.9	0.0000389	0.0000288	0.0000001	0.0000003
6455.4	772.7	9393.3	686.6	0.0000027	0.0000085	0.0000005	0.0000031
		38388.8	9119.1	0.0000016	0.0000036	0.0000002	0.0000003
	2722.2	9399.3	4888.8	0.0000001	0.0000012	0.0000002	0.0000011
		393393.3	68886.6	0.0000001	0.0000007	0.0000001	0.0000001

Table 1.

where $\tilde{a}=\frac{X(E)}{R(E)}$, $\tilde{b}=\frac{Y(E)}{R(E)}$,

$\widetilde{Ar}=\frac{2}{R(E)}\cdot\sqrt{XX(E)\cdot R(E)-(X(E))^2}$, $\widetilde{Br}=\frac{(R(E))^2}{2\cdot\pi\cdot\sqrt{XX(E)\cdot R(E)-(X(E))^2}}$.

4 Comments and Conclusion

In this paper our studies are focused on the digital ellipses and problems of their representation and reconstruction. In the previous sections representation by constant number of integers, requireing optimal number of bits, is presented. One-to-one correspondence between the digital ellipses and their proposed codes is proved. That enables an approximate, constant time reconstruction of the digital ellipse from its proposed code. The efficiency of the reconstruction is analysed and it is shown that the errors in estimating the half-axes of the ellipse and the coordinates of its center tend to zero while the nunber of pixels per unit (i.e. resolution) tends to infinity. The illustration by the experimental results is given.

References

1. R. Klette, I. Stojmenović and J. Žunić, "A parametrization of digital planes by least square fits and generalizations", *Graphical Models and Image Processing*, vol. 58, no. 3, pp. 295-300, 1996.
2. R. A. Melter, I. Stojmenović and J. Žunić, "A new characterization of digital lines by least square fits", *Pattern Recognition Letters*, vol. 14, pp. 83-88, 1993.
3. J. Žunić and J. Koplowitz, "A representation of digital parabolas by least square fits", *SPIE Proc.*, vol. 2356, pp. 71-78, 1994.

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 199–205

Algorithm for reducing the calculating complexity of the polynomial expressions

Dušan Surla and Miloš Racković

Faculty of Sciences, Institute of Mathematics
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia

Abstract. Dividing the polynomial expressions into the products is a basic part of the algorithm for reducing the number of calculating operations in the analytical expressions of the robotic mechanisms models. On the basis of the mathematical background the data structures and the algorithm for dividing the polynomial expressions is developed. The central topic of paper is the algorithm for choosing the products which gives the largest reduction in the number of calculating operations. The obtained results are shown in the illustrative example.

1 Introduction

Forming the mathematical models of the robotic mechanisms in symbolic form is one example for the modelling of complex systems. The analytical expressions which exists in the model have great calculating complexity which should be reduced in order of obtaining the efficient models. In [1], the complete procedure for generating the symbolic models of the dynamics of complex robotic mechanisms is described and in [2] is given the structural system analysis of the reducing calculating complexity process.

The model analytical expressions which are in the polynomial form

$$Y = \sum_{i=1}^{I} k_i \cdot \prod_{l=1}^{L} x_l^{c_{il}} \tag{1}$$

should be transformed in the form

$$Y = \sum_{w=1}^{W} (Y_{w1} \cdot Y_{w2}) + Y_{W+1} \tag{2}$$

where Y_{w1}, Y_{w2}, $w = 1, ..., W$ and Y_{W+1} are also the expressions of the type (1). This is achieved by dividing the expressions into the products [1-5]. First

part of the procedure for dividing the expressions comprises the generating of all products, candidates for dividing, and its description is given in [6].

This paper gives the mathematical basis for dividing the expressions into the products. Paper also describes the data structures and the algorithms for choosing these products which gives the "maximal" reduction in the number of calculating operations, and for dividing into the chosen products. The complete procedure is tested on the illustrative example.

2 Background

The problem of dividing some expression into the products is analogous to the problem of finding the structural matrices A and B which satisfy the equation

$$A \cdot B = C \tag{3}$$

for given structural matrix C, and can be represented by the following system of equations

$$e_i^A + e_j^B = e_m^C \ ; \ k_i^A \cdot k_j^B = k_m^C \tag{4}$$
$$m = (i-1) \cdot J + j, \quad i = 1, ..., I; j = 1, ..., J.$$

In [1, 4] has been shown that satisfying equations (5) for all different $m1$, $m2$, $m3$, $m4 \in \{1, ..., M\}$ gives the necessary and sufficient condition for existing the solution of the system (4).

$$e_{m1}^C - e_{m3}^C = e_{m2}^C - e_{m4}^C \ ; \ \frac{k_{m1}^C}{k_{m3}^C} = \frac{k_{m2}^C}{k_{m4}^C} \tag{5}$$

This claim was proved by deriving one solution which satisfies the system (4). This proof is used for developing the algorithm which effectively divides the expression into the given products. The idea of the algorithm follows.

For e_1^A we obtain each of the components so that $e_{1l}^A = min(e_{1l}^C, ..., e_{Jl}^C)$ for $l = 1, ..., L$. From J equations in which e_1^A participates we have that

$$e_j^B = e_j^C - e_1^A \ ; \ j = 1, ..., J. \tag{6}$$

Now for each $e_i^A, i \neq 1$ remains J equations from which follows that

$$e_i^A = e_{m_1}^C - e_1^C + e_1^A = ... = e_{m_J}^C - e_J^C + e_1^A \tag{7}$$
$$m_j = (i-1) \cdot J + j \ ; \ i = 2, ..., I.$$

The analogous procedure is applied for the vector of coefficients. For k_1^A we take the value 1.00. From J equations in which k_1^A participates we have that

$$k_j^B = k_j^C \ ; \ j = 1, ..., J. \tag{8}$$

Now, for each $k_i^A, i \neq 1$ remains J equations from which follows that

$$k_i^A = \frac{k_{m_1}^C}{k_1^C} = ... = \frac{k_{m_J}^C}{k_J^C} \tag{9}$$

$$m_j = (i-1) \cdot J + j \ ; \ i = 2, ..., I.$$

That solution, derived by this algorithm, satisfies the system (4) has been shown in [1, 4].

Before dividing the expressions, the products which gives the "maximal" reduction in the number of calculating operations should be chosen. The quotation marks is used because for this choice the suboptimal algorithm is applied. The idea of the algorithm follows.

¿From all generated products that one is chosen which gives maximal reduction in the number of operations. Then from all other products, the chosen addends are eliminated and the same procedure is repeated. The procedure stops when there is no product left to be choose. It is obvious that this greedy algorithm which in every step takes the best solution does not guarantee the globally best solution. Still, this algorithm can be efficiently implemented and gives satisfactory results in the practical applications.

3 Choice of the Products

All products candidates for dividing the expressions were generated by algorithm described in [6]. These products are stored in the list of products *lispro*. This is a list of structures which represents the products. The data structure which describes one product is given in the pseudocede.

struct *product* { int m, n, $matadd[m][n]$; struct *product* $*$ *nextpr*; }

Dimension of the product is represented with m and n and equals $m \times n$. The matrix $matadd[m][n]$ represent indexes of the addends which forms the given product. In *nextpr* the pointer on the next product in the list of products is given. An array of pointers *arrpp* with dimension *dap* is introduced, where every member of this array points on the first element in the list of products which has corresponding value for m. ¿From the list of products *listpro* the choice of those products which gives the "maximal" reduction in the number of calculating operations is performed. The procedure *ChoBestPro* is given in the pseudocode.

```
ChoBestPro(lispro, arrpp, dap) {
  p = lispro;
  while (p! =NULL) {
    while (p! = arrpp[dap − 1]) p = p− > nextpr;
    MaxDim(p); p3 = ReconPro(lispro, p, arrpp, dap);
    InsLisPro(nlispro, p3); DelProLis(p3, lispro, narrpp, dap);
    p = lispro; }
  lispro = nlispro; }
```

The first step of this procedure is the positioning of the variable p on the first product with the maximal value for m. This is the product pointed by the pointer $arrpp[dap-1]$. Then among the products with the same values for m,

the variable p is positioned, by the procedure $MaxDim(p)$, on that product which has maximal dimension $m \times n$. This product is eliminated from the list of the products *lispro* by the procedure $ReconPro(lispro, p, arrpp, dap)$, which reconnects the list and if it's necessary modifies the array of pointers *arrpp* and its dimension *dap*. Then the chosen product $p3$ (one with the maximal dimension) is inserted in the new list of products *nlispro* by the procedure $InsLisPro(nlispro, p3)$. In this new list those products are stored, which are chosen for dividing the expression.

The next step is modifying all the products that remain in the list *lispro*, which is achieved by applying the procedure $DelProLis(p3, lispro, arrpp, dap)$. By this procedure those addends which participates in the chosen product $p3$ are eliminated from all other products, because in the process of dividing the expression every expression addend can be used only ones. In this way some of the products are completely eliminated from the list of products and some of them have the new, smaller dimension. In the case of eliminating the product from the list, the modifying of array of pointers *arrpp* and its dimension *dap* is performed if it's necessary. In the case of changing the dimension, the value for m can be smaller then before and the product must be transferred to the corresponding place in the list *lispro*. Then the variable p is again positioned on the beginning of the list *lispro*, and the same procedure is repeated while the list of products *lispro* isn't empty. On the end, the pointer is set on the new list *nlispro* in which all the chosen products are stored.

4 Dividing the Expression into the Chosen Products

When all the products for dividing are chosen, the effectively dividing the expression into the chosen candidates is performed on the basis of algorithm described in the section 2. The dividing is performed by applying the procedure $DivExpPro(Y, lispro, M, arrproY, remexpY)$ which divides the expression Y into the products from the list *lispro*. The array of couples of the expressions *arrproY* with dimension M, and the remainder expression *remexpY* represent the result of dividing according to equation

$$Y = \sum_{l=1}^{M} (arrproY[l][1] \cdot arrproY[l][2]) + remexpY \tag{10}$$

The procedure *DivExpPro* is given in the pseudocode.

```
DivExpPro(Y, lispro, M, arrproY, remexpY) {
  for (l = 1;l ≤ M ;l++) DivPro(Y, lispro, l, arrproY[l]);
  remexpY = FindRem(Y, M, arrproY); }
```

The procedure $DivPro(Y, lispro, l, arrproY[l])$ generates the expressions $arrproY[l][1]$ and $arrproY[l][2]$ for l-th member of the list of products *lispro* according to algorithm for deriving the solution described in the section 2. The

procedure $FindRem(Y, M, arrproY)$ obtains the part of expression Y which remains when all the addends participating in chosen products are eliminated. This expression $remexpY$ is the expression which can not be divided into the products any more. By this the algorithm for dividing the expressions into the products is completed and the monomial extraction algorithm [1], which is the next step in the reducing the calculating complexity of the analytical expressions, can be applied.

5 Example

The algorithm is implemented in the programming language C under the SCO UNIX operating system. Algorithm for dividing the expression into the products is applied on the expression Y, which is represented by the structural matrix $S = (K_S, X_S, E_S)$ given in the Table 1.

Table 1. Structural matrix of the expression Y

i	k_i	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
0	0.1574	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	-0.0105	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
2	0.3306	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
3	-0.3306	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0
4	-0.0315	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0
5	0.0428	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
6	0.0079	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
7	-0.0026	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
8	-0.0139	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	-0.0394	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
10	0.0132	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
11	-0.0157	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
12	-0.0787	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
13	-0.0292	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
14	-0.1653	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
15	0.0553	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
16	0.214	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
17	0.0394	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
18	-0.0132	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
19	0.0028	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
20	0.0079	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
21	-0.0026	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0
22	0.8987	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
23	0.0292	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
24	0.1653	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
25	-0.0553	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
26	0.0014	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
27	0.007	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The rows represent the addends, columns represent the variables, and in the intersection of the row and the column is the exponent of the corresponding variable in the corresponding addend. By applying the algorithm described in [6] the 11 products $p_1, ..., p_9, p_{11}$ were generated, which are shown by the n-tuples of indexes of the chosen addends participating in the products.

$$p_1 = [(5,7),(16,18),(22,25)] \; ; \; p_2 = [(5,6),(16,17),(22,24)]$$
$$p_3 = [(9,10),(14,15),(20,21)] \; ; \; p_4 = [(16,22),(17,24),(18,25)]$$
$$p_5 = [(6,7),(17,18),(24,25)] \; ; \; p_6 = [(5,22),(6,24),(7,25)]$$
$$p_7 = [(5,16),(6,17),(7,18)] \; ; \; p_8 = [(0,4),(8,19),(9,20),(10,21)]$$
$$p_9 = [(0,8),(2,13),(3,23),(4,19),(11,26),(12,27)]$$
$$p_{10} = [(5,7,6),(16,18,17),(22,25,24)] \; ; \; p_{11} = [(5,22,16),(6,24,17),(7,25,18)]$$

The first 9 products are obtained by generating the all $2 \times N$ products, and the remaining two are obtained by constructing the products of the greater dimensions [6].

By applying the procedure *ChoBestPro* 3 products which gives the "maximal" reduction in the number of calculating expressions are chosen. In this case this is the actual maximal reduction. These are the products p_{11}, p_9 and p_3. By applying the procedure *DivExpPro* dividing of the expression Y is performed according to

$$Y = \sum_{w=1}^{3} (Y_{2 \cdot w-1} \cdot Y_{2 \cdot w}) + Y_7 \tag{11}$$

where the expressions Y_1,...,Y_7 are represented by the structural matrices S_1,...,S_7 given in Table 2.

6 Conclusion

The reduction of calculating complexity of the analytical expressions, participating in the mathematical model of system, is necessary part of the mathematical modelling process in symbolic form. The aim of reducing the model calculating complexity is the need for calculating numerical values of the model quantities in the real-time regime.

In this paper, the second part of the algorithm for dividing the polynomial expressions into the products is described. This part of algorithm contains choosing of the products which gives the "maximal" reduction in the number of calculating operations and effectively dividing the expression into the chosen product. For the choice, the suboptimal algorithm is applied, which in efficient fashion determines which products are to be choose. The complete algorithm is implemented and the results are shown on the example which illustrates the effectiveness of the algorithm.

Table 2. Structural matrices of the expressions $Y_1,...,Y_7$

S	k_i	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
S_1	1.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	5.0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	21.0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
S_2	0.0428	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	0.0079	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	-0.0026	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
S_3	1.0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	-0.0884	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S_4	0.1574	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	0.3306	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
	-0.0315	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
	-0.3306	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
	-0.0157	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	-0.0787	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
S_5	1.0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	-0.3345	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
S_6	-0.0394	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	-0.1653	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	0.0079	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
S_7	-0.0105	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0

References

1. Racković, M.: Generation of the Mathematical Models of Complex tic Mechanisms in Symbolic Form. Ph. D. thesis, University of Novi Sad, (1996)
2. Racković, M., Surla, D.: Reduction of the Computational Complexity of the Mathematical Models of the Dynamics of Robotic Mechanisms. 10^{th} International Conference on Industrial Systems IS'96, October 01-03, (1996), Novi Sad, (In Serbian) 217–222
3. Racković, M., Surla, D.: Data Structures and the Algorithms for Automatic Generating of the Robotic System Models. Journal for informatics, computer sciences and telecommunications of Yugoslav Informatic Society INFO, **3/95** (1995) (In Serbian) 4–6
4. Racković, M., Surla, D.: The Algorithms and Data Structures for Forming Symbolic Models of the Robotic Systems. FILOMAT (Niš) **9:3** (1995), 887–897
5. Surla, D., Racković, M.: "Forming a Calculation Graph for the Polynomial Expressions. X Conference on Applied Mathematics PRIM'95, Budva (1995), 275–282
6. Racković, M.: Generating the Products, Candidates for Dividing the Polynomial Expressions. VIII Conference on Logic and Computer Science "LIRA '97", Novi Sad, (1997), (to appear)

This article was processed using the LaTeX macro package with LLNCS style

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '97
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1 - 4, 1997., PP. 207-214

Optimizing Abstract SECD Machine Code

Lehel Szarapka
Faculty of Civil Engineering, Dept. of Mathematics and Physics
24000 Subotica, Kozaračka 2/a, Yugoslavia
e-mail: lehel@gf.su.ac.yu

Mirjana Ivanović
Faculty of Science, Institute of Mathematics,
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: zjb@unsim.ns.ac.yu

Abstract

This paper presents a new algorithm for optimizing abstract SECD machine code. The algorithm has two key features: the more effective usage of the top of the stack S and the handling of constant declarations. The optimization techniques are simple, straightforward, and based on a common sense. Therefore, they are easy to understand, implement, and port to other abstract machines. The optimization techniques are implemented via a novel set of instructions that are built into a SECD to Modula-2 translator. It is shown that by optimizing the abstract SECD machine code, usually a 25% - 39% speedup can be achieved.

1 Introduction

The standard way to implement functional languages is:

a) to transform the higher level functional language to an intermediate language,
b) to compile this intermediate language into a machine language of any of the known abstract machines, and either
c_1) to simulate the chosen abstract machine on a real machine, or
c_2) to translate the language of the chosen abstract machine into a language of a real machine.

This is called the "implementation chain" and every of its three steps (transformation, compilation, simulation/translation) offers a broad spectrum of possibilities to implement optimization. Optimization can also be implemented on several levels:

- on the original functional language,
- on the intermediate functional language,
- on the language of the abstract machine,
- on the language of the real machine.

As we move down along the implementation chain, the optimization become more general, because they affect everything which is "up the chain." For example, by implementing an optimization technique on the level of an intermediate language, it is immediately available to every higher level functional language, which is implemented via transformation to the intermediate language.

In this paper two optimization techniques for the abstract SECD machine are considered. Every program, written in any of the known functional languages, which are implemented via the SECD machine could use the benefits of these optimization techniques. It is also shown, that by using these optimization techniques, the execution efficiency of the code can be increased by about 25% - 39%. The both techniques are simple and straightforward yet effective and efficient. They can easily be applied to every abstract stack based machine.

The rest of the paper is organized as follows. Section two gives a short overview of the SECD machine. In the third and fourth section the two new optimization techniques are introduced. The fifth section presents some test results. The last section concludes the paper.

2 SECD Machine

The SECD machine [3] was the first abstract machine designed for implementation of functional programming languages. It is a machine for evaluating λ-expressions or programs of those programming languages that are based on λ-calculus. The SECD machine consists of four stacks:

- the stack S (Stack) - which is used to store the results obtained during the evaluation,
- the stack E (Environment) - which is used to store identifiers and their values,
- the stack C (Control) - which contains the SECD machine code,
- the stack D (Dump) - which is used to store the contents of the other three stacks, when necessary.

As all other automata, the SECD machine's operation can be represented by a set of states and transition rules. In the initial state, register `S` contains the program arguments, register `C` contains the SECD machine code, and the other two registers are empty. All four registers (stacks) can be represented as s-expressions (a data structure best known from LISP and LISP-like languages.) The empty stack is denoted by NIL (a special, atomic s-expression that denotes an empty s-expression.) The stack X ($X \in \{$`S`, `E`, `C`, `D`$\}$) with its top elements x_i, i=1, ..., n is denoted as $(x_1\ x_2 \ldots x_n\ .\ X)$.

The SECD machine has a set of commands. A program of the machine consists of a sequence of SECD machine commands. Every command *comm* can be described with an appropriate transition rule of the form:

comm: `S E C D` → `S' E' C' D'`

where S, E, C, and D are register contents before execution of the command *comm*, and S', E', C', and D' are register contents after the execution of *comm*. After the whole program is executed, the result of a program (i.e., the result of an evaluation carried out by a machine) is on the top of register (stack) S.

Henderson's version of the SECD machine has a basic set of 21 commands [2]. It can be extended by new commands to support new functionality of a machine [1]. As an illustration we cite some transition rules of an extended SECD machine [1, p. 92-93].

LDC:	S E (LDC *x*.C) D	→ (*x*.S) E C D
LD:	S E (LD *i*.C) D	→ (*loc(i,E)*.S) E C D
LDF:	S E (LDF *c'*.C) D	→ ((*c'*.E).S) E C D
LDE:	S E (LDE *c'*.C) D	→ ([F(*c'*.E)].S) E C D
RTN:	(*x*) E (RTN) (*s' e' c'*.D)	→ (*x*.*s'*) *e' c'* D
DUM:	S E (DUM.C) D	→ S (Ω.E) C D
		(Ω is so-called dummy environment)
AP:	((*c'*.*e'*) *v*.S) E (AP.C) D	→ nil (*v*.*e'*) *c'* (S E C.D)
RAP:	((*c'*.*e'*) *v*.S) (Ω.E) (RAP.C) D	→ nil *rh*(*e'*,*v*) *c'* (S E C.D)
SEL:	(*x*.S) E (SEL *y z*.C) D	→ S E (y) (C.D) *(if x is TRUE)*
SEL:	(*x*.S) E (SEL *y z*.C) D	→ S E (z) (C.D) *(If x is FALSE)*
JOIN:	S E (JOIN) (C.D)	→ S E C D
EQ:	(*x y*.S) E (EQ.C) D	→ ((*x* = *y*).S) E C D
	(similarly for LEQ, LE, EQN, LEQN, LESN, EQS, LEQS i LESS)	
ADD:	(*x y*.S) E (ADD.C) D	→ ((*x* + *y*).S) E C D
	(similarly for SUB, MUL, QUO, DIV i MOD)	
SIN:	(*x*.S) E (SIN.C) D	→ ((*sin x*).S) E C D
	(similarly for COS, EXP, SINH, COSH, LOG, ATAN i ATANH)	
CONS:	(*x y*.S) E (CONS.C) D	→ ((*x*.*y*).S) E C D
CAR:	((*x*.*y*).S) E (CAR.C) D	→ (*x*.S) E C D
CDR:	((*x*.*y*).S) E (CDR.C) D	→ (*y*.S) E C D
APND:	(*x y*.S) E (APND.C) D	→ ((*x* + + *y*).S) E C D
	(similarly for MEMB, NTH and REST)	
LEN:	(*x*.S) E (LEN.C) D	→ ((*len x*).S) E C D
	(similarly for ATOM, NUL, CHR and ORD)	
STOP:	S E (STOP) nil	→ S E (STOP) D
	(*x*) E (STOP . C) (*s' e' c'*.D)	→ (*x*.*s'*) *e' c'* D

After the evaluation ends, the result is on top of the stack S.

Although the SECD machine was the first machine capable of evaluating λ-expressions, even today it is still a good basis for many other (and more modern) abstract machines (for example: G machine) and an excellent test-bed for implementations of functional languages. It implements strict semantics, i.e., eager evaluation.

2.1 The SECD to Modula-2 Translator

Instead of simulating SECD machine, the translator from SECD machine language to Modula-2

was developed [4]. By translating the SECD program into Modula-2, a directly executable code is made from any functional program implemented via the SECD machine. The execution efficiency of those programs were significantly improved (20% - 28%). The translator also enabled the better understanding of the operation of the SECD machine and opened a door for possible optimization techniques. The following two techniques are now a regular part of the translator.

3 The Handling of Constants

The first of the two techniques deals with predefined program constants. The SECD machine does not make any difference between constants and identifiers representing functions or other values. Every time a constant is used, its value is just pushed onto the stack `S`, for example, `ldc 1`, `ldc NIL`, etc.

Since all values (including constants) in SECD machine are "boxed", every time a constant is used:

a) a new space is created in the main computer storage,
b) the constant value is packed into it (i.e., "boxed"), and
c) pushed onto the stack S.

Because the creation and boxing uses space and time, it would be better if we could create in advance a list of all constants used in the program. They could be then just taken from the list of prepared constants when needed. Of course, this list must be initialized before the evaluation of the main function takes place.

Our first optimization technique does just that. When the translator in its first pass scans the program looking for functions [4], it gathers all program constants, create a memory space for them, box them into that space, and puts them into an array. Now the constants are ready-made and can be used without any additional overheads.

4 An Effective Usage of the Top of the Stack `S`

The second optimization that is discussed, involves the more effective usage of the top of the stack S. Many SECD machine commands take their arguments from the top of the stack `S` and leave their result there. Based on this observation we can divide the SECD commands into four groups:

- commands that take their arguments from the top of the stack `S`, but do not leave any result behind (`ap`, `rap`, `sel`, `rtn`, etc.,)
- commands that leave their result on the top of the stack `S`, but need no arguments (`ld`, `ldc`, `ldf`, etc.,)
- commands that take at least one argument from the top of the stack `S` and leave their result also on the top of `S` (`car`, `cdr`, `eq`, `add`, `mul`, `len`, `sin`, `neg`, etc.,)
- commands that do not have any arguments (`join`, `dum`, `stop`, etc.)

In cases when a command that leave its result on the top of the stack `S` is followed by a

command that takes its argument from S, an optimization can be applied. The first command's result will not be pushed onto S at all, but just passed to the second command via the intermediate register TopOfS. This way a one push and one pop operation is avoided.

To implement this optimization, a new set of SECD commands has been developed. They can be recognized by their suffix, which is:

- "In" - the command takes its arguments from the TopOfS, but either leaves nothing behind or leaves its result on the top of S.
- "Out" - the command leaves its result in the TopOfS, but either needs no arguments or takes its arguments from S.
- "InOut" - the command takes its arguments from the TopOfS and leaves its result also in TopOfS.
- none - the command does not use TopOfS.

The algorithm that generates the optimized SECD code from the original one is fairly simple and will not be discussed here. To demonstrate the proposed optimization, observe the following SECD code translated into equivalent Modula-2 code (taken from the factorial function [4],) without optimization.

```
PROCEDURE Function2;
BEGIN
    DoLDC(0);                          (* ldc 0 *)
    DoLD(0, 0);                        (* ld (0 . 0) *)
    DoEQ;                              (* eq *)
    dummy := Top(S); Pop(S);
    IF IsTrue (dummy) THEN             (* sel *)
       DoLDC(1);                       (* ldc 1 *)
       (* joined *)                    (* join *)
    ELSE
      DoLD(0, 0);                      (* ld (0 . 0) *)
      DoLDC(NIL);                      (* ldc nil *)
      DoLDC(1);                        (* ldc 1 *)
      DoLD(0, 0);                      (* ld (0 . 0) *)
      DoSUB;                           (* sub *)
      DoCONS;                          (* cons *)
      DoLD(1, 0);                      (* ld (1 . 0) *)
      DoAP;
      (* load procedure variable Fun - prepare application *)
      Fun;                             (* ap *)
      DoMUL;                           (* mul *)
      (* joined *)                    (* join *)
    END;
    DoRTN;                             (* rtn *)
END Function2;
```

The optimized version is as follows (the array Constants contains previously initialized constants:)

```
PROCEDURE Function2;
BEGIN
  TopOfS := Constants[6];    (* 0 *)
  DoLDC;

  DoLDOut(0,0);
  DoEQInOut;
  IF IsTrue (TopOfS) THEN
    TopOfS := Constants[5]; (* 1 *)
    DoLDC;
    (* joined *)
    ELSE
  ELSE
    SECDcommands.DoLD(0,0);
    TopOfS := Constants[4]; (* NIL *)
    DoLDC;
    TopOfS := Constants[5]; (* 1 *)
    DoLDC;
    DoLDOut(0,0);
    DoSUBInOut;
    DoCONSIn;
    DoLDOut(1,0);
    DoAPIn;
    Fun;
    DoMUL;
    (* joined *)
  END;
  DoRTN;
END Function2;
```

Note that after a command with an `Out` suffix there is always a command with an `In` suffix, and vice versa. Before a command with an `In` suffix there is always a command with an `Out` suffix. The commands with the `InOut` suffix can play the role of commands with `In` or `Out` suffixes.

5 Results

The execution efficiency of seven benchmark programs were measured on the SECD machine translator. The benchmark programs were divided into two groups:

- programs involving arithmetics (Fibonacci numbers in two different ways, matrix operations and the Takeuchi function),
- programs involving symbolic computation and no numerical operations (eight queens on a chessboard, and two manipulations with lists).

The results are displayed in the following two tables, where the speedup ratio of optimized programs with respect to programs that are not optimized is given. The first table presents the measurements for the programs with arithmetic operations, while the second table shows the results for the programs with symbolic computations.

Program	Speedup ratio
Fibonacci numbers - 1	32% - 34%
Fibonacci numbers - 2	31% - 34%
Matrix operations	28% - 31%
Takeuchi function	25% - 26%

Program	Speedup ratio
8 Queens	36% - 39%
List - 1	25% - 36%
List - 2	26% - 32%

As expected, the execution efficiency is significantly improved. Note, that there is very little difference between the speedup ratio of programs with or without numerical computations.

6 Conclusion

In this paper two optimization techniques for the abstract SECD machine are proposed. The mentioned optimization techniques (the handling of constant declaration and the more effective usage of the top of the stack S) have been built into the SECD to Modula-2 translator. The results, obtained by measuring the execution efficiency of seven benchmark programs show that it is worth implementing these techniques. The speedup ratio is 25% - 39%.

The paper also shows how two simple and straightforward techniques can significantly improve the efficiency of resulting programs. These two techniques can be applied to every stack-based abstract machine in a similar way, even without deep understanding of machine's underlying structure.

References

[1] Budimac, Z.,*The Contribution to the Theory and Implementation of Functional Programming Languages*, Ph.D. thesis, University of Novi Sad, Novi Sad, Yugoslavia, 1994.

[2] Henderson, P., *Functional Programming - Application and Implementation*, Prentice Hall, New York, 1980.

[3] Landin, P.J., *The Mechanical Evaluation of Expressions*, Computer J. vol. 6, num 4, 1964, 308-320.

[4] Szarapka, L., Budimac, Z., Ivanović, M., *Translating an Abstract SECD Machine Code into Modula-2*, Bull. Appl. Math., Budapest, in print, 1997.

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 215–222

A note on L-fuzzy relations

Goran Trajkovski[1] and Biljana Janeva[2]

[1] Faculty of Electrical Engineering, Institute of Mathematics and Physics
PO Box 574, 91000 Skopje, Republic of Macedonia
e-mail: goranpt@cerera.etf.ukim.edu.mk
[2] Faculty of Natural Sciences and Mathematics, Institute of Informatics
PO Box 162, 91000 Skopje, Republic of Macedonia
e-mail: biljana@robig.pmf.ukim.edu.mk

Abstract. In the paper some alternative definitions of reflexivity, antisymmetry, and transitivity of L-valued binary fuzzy relations are given, as well as the decomposition and synthesis theorems, in order to state the connection between the fuzzy relation with particular properties and its crisp level (cut) relations, and to investigate the properties that are inherited in the process of their decomposition and synthesis. A taxonomy system for fuzzy relations is also proposed.

1 Introduction

A crisp relation represents the presence or absence of association, interaction or interconnectedness between the elements of two or more sets [4]. The concept of fuzzy relation is introduced naturally, as generalization of crisp relations in fuzzy set theory [14]. It models situations where interactions between elements are more or less strong. Crisp relation can thus be viewed as a special, restricted case of fuzzy relation.

Ordinary fuzzy relations (having the unit interval [0, 1] for its valuation set) can be extended to L-fuzzy relation (whose valuation set is a lattice), in the same way as ordinary fuzzy sets are extended to L-fuzzy sets [2].

In [15], Zadeh gives definitions of the four elementary properties of relations in the fuzzy case: reflexivity, antisymmetry, symmetry, and transitivity, and proves the decomposition and synthesis theorem in the case of fuzzy relations, fuzzy equivalence (similarity), and fuzzy orderings. Those theorems give the connection between the fuzzy relation and its cuts, which represent crisp relations.

Valuable attempts are also made to define other fuzzy analogues to the definitions of special relation properties. Most of them try to make generalizations that will coincide with the crisp, well-known property definitions when applied to a crisp relation.

Our intention is to extend the existing theory considering the decomposition and synthesis to binary lattice-valued fuzzy relations with alternatively defined special properties. The paper deals with the theorems of decomposition and synthesis of several types of L-fuzzy relations, considering alternative definitions of reflexivity found in [3, 13], antisymmetry [5, 11], transitivity in the min-max case, and few others.

The paper is organized in the following manner. In Section 2 we give the standard definitions of elementary properties of relations, as well as the decomposition and synthesis theorem as in [15]. In section 3 definitions of aletrnative properties are stated, and a taxonomy system for those relation is proposed. In the following section the decomposition and synthesis of fuzzy relations with alternative property(ies) are investigated. In the last section we conclude our work.

2 Preliminaries

Zadeh defines fuzzy binary relation $\overline{S}$ of the non-empty sets X and Y to be a mapping from the Cartesian product $X \times Y$ to the real unit interval $[0, 1]$ ([14]), as a generalization of the notion of a characteristic function of relation, observed as set. Goguen [2] considered L-valued fuzzy sets, replacing the $[0, 1]$ interval by a set of particular structure (e.g. lattice, group structure etc.).

We shall be considering L-valued fuzzy relations as mappings $\overline{S} : X \times X \to L$, where $\mathcal{L} = (L, \wedge, \vee)$ is complete lattice ([7]) with the least element 0 and the greatest element 1 [11,12,13].

For every $p \in L$, p-cut of $\overline{S}$ [6] is the mapping $\overline{S}_p : X^2 \to \{0, 1\}$, such that for $x, y \in X$, $\overline{S}_p(x, y) = 1$ iff $\overline{S}(x, y) \geq p$. It is obvious that $\overline{S}_p$ represents a crisp relation for every $p \in L$.

We denote the p-cuts familly of a L-fuzzy relation $\overline{S}$ by $\overline{S}_L$, i.e. $S_L = (\overline{S}_p | p \in L)$. It is well known that all the p-cuts synthesize the relation [23], for $x, y \in X$,

$$\overline{S}(x, y) = \bigvee_{p \in L} p \cdot \overline{S}_p(x, y),$$

where $\bigvee$ is the supremum in L, and $\cdot$ is defined with $p \cdot 0 = 0$ and $p \cdot 1 = p$.

The usual definitions of elementary properties (reflexivity, symmetry, antisymmetry, and transitivity) of fuzzy relations are as given below [1].

The L-fuzzy relation $\overline{S} : X^2 \to L$ is

1. reflexive if $\overline{S}(x, x) = 1$, for all $x \in X$;
2. symmetric if $\overline{S}(x, y) = \overline{S}(y, x)$, for all $x, y \in X$;
3. antisymmetric if $\overline{S}(x, y) \wedge \overline{S}(y, x) = 0$, for all $x, y \in X$ such that $x \neq y$;
4. transitive if $\overline{S}(x, y) \wedge \overline{S}(y, z) \leq \overline{S}(x, z)$, for all $x, y, z \in X$.

We denote $\overline{S} \in \mathcal{SR}_1$, $\overline{S} \in \mathcal{S}$, $\overline{S} \in \mathcal{SA}_1$, and $\overline{S} \in \mathcal{T}_{max-min}$ respectively in the cases 1-4 of the previous definition.

The relation $\overline{S}$ is relation of fuzzy equivalence (similarity relation) if $\overline{S} \in \mathcal{SR}_1 \cap \mathcal{S} \cap \mathcal{T}_{max-min}$. If $\overline{S} \in \mathcal{SR}_1 \cap \mathcal{SA}_1 \cap \mathcal{T}_{max-min}$, then the relation represents fuzzy ordering. The term fuzzy quasi-similarity is used when $\overline{S} \in \mathcal{S} \cap \mathcal{T}_{max-min}$, and compatibility relation (tolerance relation) when $\overline{S} \in \mathcal{SR}_1 \cap \mathcal{S}$. $\overline{S}$ is preorder (quasi-ordering relation), if $\overline{S} \in \mathcal{SR}_1 \cap \mathcal{T}_{max-min}$.

It can be noticed that the above definitions are only natural extensions of the corresponding definitions in the theory of crisp relations. Applied to crisp relations, viewed as special cases of fuzzy relations, we get corresponding crisp relations. One must here notice that they are not the only extensions of the crisp definitions.

The above list do not intend to be exhaustive listing of relations posessing variants of elementary properties, since it can get on length by extending definitions from the crisp to the fuzzy case.

Proposition 1. *[12, 23] All the p-cuts of a similarity relation $\overline{S}$ are (ordinary) equivalence relations on X.*

Proof. Straightforward. □

We will give the proof of the following proposition since its concepts will be useful in proving some of the propositions below (theorems of synthesis).

Proposition 2. *[12] Let $\mathcal{F} = \{\rho_i | i \in I\}$ be a family of equivalence relations on X, closed under intersection, containing X^2. Let also L be a lattice antiisomorphic with $(\mathcal{F}, \subseteq)$. We define the relation $\overline{S} : X^2 \to L$, so that for every $x, y \in X$*

$$\overline{S}(x, y) := \bigvee \{\rho_i | (x, y) \in \rho_i\},$$

where $\bigvee$ is supremum in L (intersection in $\mathcal{F}$), and the corresponding elements in $\mathcal{F}$ and L are denoted identically. Now, $\overline{S}$ is a similarity relation on X, $\mathcal{F}$ is a family of its p-cuts, and, moreover, $\overline{S}_{\rho_i} = \rho_i$, for every $i \in I$.

Proof. The fuzzy relation $\overline{S}$ is well defined as a mapping, that is, for $x, y \in X$, due to the intersection closure property, the family $\{\rho_i | (x, y) \in \rho_i\}$ is unanimously defined.

It is obvious that $\overline{S}$ is L (lattice)–valued fuzzy relation. What is to prove now is that for every $i \in I$, $\rho_i = \overline{S}_{\rho_i}$.

Let $x, y \in X$. Then $(x, y) \in \overline{S}_{\rho_i}$

iff $\overline{S}(x, y) \geq \rho_i$ (because of the definition of p-cut),

iff $\bigvee_{j \in I} \{\rho_j | (x, y) \in \rho_j\} \geq \rho_i$ (for such is the definition of $\overline{S}$),

iff $\bigcap_{j \in I} \{\rho_j | (x, y) \in \rho_j\} \subseteq \rho_i$ (because of the antiisomorphism)

iff $(x, y) \in \rho_i$.

We have proven thus that $\mathcal{F}$ is the family of p-cuts for $\overline{S}$. The defining equality for $\overline{S}$ gives the resolution equality for it.

It is easy now to prove that $\overline{S}$ is similarity relation in X. □

Proposition 3. *[12, 23] All the p-cuts of a fuzzy ordering relation $\overline{S}$ for $p \neq 0$ are (ordinary) orderings on X and $\overline{S}_0 = X^2$.*

Proof. Straighforward. □

Proposition 4. *[13] Let $\mathcal{F} = \{\rho_i | i \in I\}$ be a family of ordering relations on X, closed under arbitrary intersection. Consider $\mathcal{F} \cup \{X^2\}$ and let L be a lattice antiisomorphic with $(\mathcal{F} \cup \{X^2\}, \subseteq)$. Define the relation $\overline{S} : X^2 \to L$, so that for $x, y \in X$*

$$\overline{S}(x, y) := \bigvee \{\rho_i \in \mathcal{F} \cup \{X^2\} | (x, y) \in \rho_i\},$$

where $\bigvee$ is supremum in L (intersection in $\mathcal{F} \cup \{X^2\}$), and the corresponding elements in $\mathcal{F} \cup \{X^2\}$ and L are denoted identically. Now, $\overline{S}$ is a partially ordered fuzzy relation on X, $\mathcal{F} \cup \{X^2\}$ is a family of its p-cuts, and, moreover, $\overline{S}_\rho = \rho$. for every $\rho \in \mathcal{F} \cup \{X^2\}$.

Proof. The proof is similar to the one of Proposition 2. □

3 Some Alternative Definitions

3.1 (Anti)reflexivity

The L-fuzzy relation $\overline{S} : X^2 \to L$ is

1. ε–Reflexive [21] ($\overline{S} \in \mathcal{SR}_\varepsilon$) if

$$\overline{S}(x, x) \geq \varepsilon, \quad x \in X, \varepsilon \in L, \varepsilon > 0.$$

   Note that for $\varepsilon = 1$ we get the usual definition of reflexivity in fuzzy relations, and that that definition applied to crisp relations, for $\varepsilon > 0$, give the usual definition of reflexivity in crisp relations.
2. G-Reflexive [3] ($\overline{S} \in \mathcal{GR}$) if
   (a) $\overline{S}(x, x) > 0$,
   (b) $\overline{S}(x, y) \leq \bigwedge_{z \in L} \overline{S}(z, z), \quad x, y \in X, \ x \neq y, \bigwedge_{z \in L} \overline{S}(z, z) > 0.$
3. Weakly reflexive ($\overline{S} \in \mathcal{WR}$) if $\overline{S}(x, x) \geq \overline{S}(x, y), x, y \in X$.
   If we apply the definition of weak reflexivity in the case when $L = \{0, 1\}$, we can obtain "weak reflexivity" when ordinary (crisp) relations are considered. Thus, we can say that a relation $\alpha \subseteq X^2$ is *weakly reflexive* iff

$$(x, y) \in \alpha \Rightarrow (x, x) \in \alpha, \ \textit{for all } x, y \in X.$$

4. Antireflexive [6] ($\overline{S} \in \mathcal{AR}_0$) if $\overline{S}(x, x) = 0$, $x \in X$.
5. ε-Antireflexive ($\overline{S} \in \mathcal{AR}_\varepsilon$) if $\overline{S}(x, x) < \varepsilon$, $x \in X, \varepsilon \in L$.
   It is obvious that 4. is special case of 5, and that when applied to crisp relations, for $\varepsilon < 1$, we get the definition of standard (crisp) antireflexive relation.
6. Weakly antireflexive ($\overline{S} \in \mathcal{WAR}$) if $\overline{S}(x, x) \leq \overline{S}(x, y)$, $x, y \in X$.

   Note here that $\mathcal{AR}_0 \subseteq \mathcal{WAR}$.

The above properties together with the standard reflexivity given in the previous section will be refereed as to REF-type of properties of fuzzy relation.

3.2 Antisymmetry

Let us consider the L-fuzzy relation $\overline{S} : X^2 \to L$. We say that it is

1. Weakly antisymmetric [1] ($\overline{S} \in \mathcal{WAS}$), if for $x, y \in X$, such that $x \neq y$ one of the following cases holds: $\overline{S}(x,y) \neq \overline{S}(y,x)$, or $\overline{S}(x,y) = \overline{S}(y,x) = 0$.
2. Contrasymmetric [14], ($\overline{S} \in \mathcal{CS}$), if for $x, y \in X$, $x \neq y$, and $\overline{S}(x,y) \neq 0 \neq \overline{S}(y,x)$, it holds that $\overline{S}(x,y) \neq \overline{S}(y,x)$.
3. ε–Antisymmetric ($\varepsilon \in L, \varepsilon > 0$) $\overline{S} \in \mathcal{SAS}_\varepsilon$, if for all $x, y \in X$, $x \neq y$, one of the following cases holds: $\overline{S}(x,y) \geq \varepsilon$, $\overline{S}(y,x) < \varepsilon$, or $\overline{S}(y,x) \geq \varepsilon$, $\overline{S}(x,y) < \varepsilon$.

The above properties together with the standard symmetry and antisymmetry given in the previous section consist the SYM-type of properties of fuzzy relation.

3.3 Transitivity

The L-fuzzy relation $\overline{S} : X^2 \to L$ is refereed as to min-max transitive relation [14] $\overline{S} \in \mathcal{T}_{min-max}$ if for all $x, y, z \in L$, it holds that

$$\overline{S}(x,z) \leq \overline{S}(x,y) \vee \overline{S}(y,z).$$

Min-max and max-min transitivity consist the TRA-type of properties of fuzzy relations.

3.4 Multiple feature relations – Taxonomy

A fuzzy relation $\overline{S}$ can be characterized by the triplet (R, S, T), where R is the symbol of the REF family to which $\overline{S}$ belongs (or *, if irelevant), and S and T are the symbols of the SYM and TRA families respectively, or * if irrelevant. For example, if the fuzzy relation is assigned the triplet $(\mathcal{SR}_1, S, *)$, than it is a fuzzy compatibility relation (relation of tolerance) [6], and if it is assigned the triplet $(\mathcal{GR}, *, \mathcal{T}_{min-max})$, we deal with a G-reflexive min-max transitive preorder.

4 Some Decomposition and Synthesis Theorems

Proposition 5. *All p-cuts of $(\mathcal{SR}_\varepsilon, *, *)$ relation are ordinary reflexive relations for $p \leq \varepsilon$.*

Proof. Straightforward, from the definition of p-cut of a relation and ε– reflexivity of fuzzy relations that are lattice valued. □

Proposition 6. *The p-cuts of ε– reflexive similarity relations represent ordinary equivalence relations for $p \leq \varepsilon$.*

Proof. The properties of the p-cuts regarding ε–reflexivity are considered in Proposition 5. The proof of symmetry and transitivity in the standard sense (as well as of standard reflexivity and antisymmetry) can be found in [1, 6, 14, 23] in the proofs of Propositions 1 and 3. □

Proposition 7. *Let $\mathcal{F} = \{\rho_i | i \in I\}$ be a family of equivalence relations and quasi-equivalencies closed under intersection, and containing X^2 and at least one equivalence relation different from X^2. Let also L be a lattice antiisomorphic with $(\mathcal{F}, \subseteq)$. We define the relation $\overline{S} : X^2 \to L$, so that for every $x, y \in X$*

$$\overline{S}(x, y) := \bigvee \{\rho_i | (x, y) \in \rho_i\}.$$

Then there exists an $\varepsilon \in L$, such that $\overline{S}$ is an ε–reflexive similarity relation on X, (i.e. $(\mathcal{SR}_\varepsilon, \mathcal{S}, \mathcal{T}_{max-min})$ relation) $\mathcal{F}$ is a family of its p-cuts, and, moreover, $\overline{S}_{\rho_i} = \rho_i$, for every $i \in I$.

Proof. Let us choose

$$\varepsilon := \bigvee \{\rho_i | \rho_i \text{ is an eqivalence relation in } \mathcal{F}\}.$$

The following holds

$$\overline{S}(x, x) = \bigvee_{(x,x) \in \rho_i} \rho_i \geq \varepsilon,$$

by the choice of ε. The rest of the proof is the same as the proof of Proposition 2. □

Proposition 8. *The p-cuts of $(\mathcal{WR}, *, *)$ relation are weak reflexive crisp relations.*

Proof. Let $p \in L$. Let $x, y \in X$, and $(x, y) \in S_p$, for $p \in L$, that is by definition of p-cut, $\overline{S}(x, y) \geq p$. By the definition of weak reflexivity, we have $\overline{S}(x, x) \geq \overline{S}(x, y)$, and by the assumption, $\overline{S}(x, y) \geq p$. Thus, $\overline{S}(x, x) \geq p$, i.e. $(x, x) \in S_p$. □

Proposition 9. *Let $\mathcal{F} = \{\rho_i | i \in I\}$ be a family of crisp equivalence relations and weakly reflexive quasi-equivalencies closed under intersection, and containing X^2. Let also L be a lattice antiisomorphic with $(\mathcal{F}, \subseteq)$. We define the relation $\overline{S} : X^2 \to L$, so that for every $x, y \in X$*

$$\overline{S}(x, y) := \bigvee \{\rho_i | (x, y) \in \rho_i\}.$$

Then, $\overline{S}$ is a weakly–reflexive similarity relation on X, (i.e. $(\mathcal{WR}, \mathcal{S}, \mathcal{T}_{max-min})$ relation) $\mathcal{F}$ is a family of its p-cuts, and, moreover, $\overline{S}_{\rho_i} = \rho_i$, for every $i \in I$.

Proof. Because of the proofs given above, it remains to prove that $\overline{S}$ is weakly reflexive. Let $x \in X$. Then

$$\overline{S}(x, x) = \bigvee_{(x,x) \in \rho_i} \rho_i \geq \bigvee_{(x,y) \in \rho_i} \rho_i = \overline{S}(x, y), \quad y \in X.$$

□

The proofs of the following propositions are obvious, due to the definitions from Section 3.

Proposition 10. *If $\overline{S}$ is a G-reflexive fuzzy relation, than it cuts are reflexive for*

$$p \leq \bigwedge_{z \in L} \overline{S}(z, z).$$

Proposition 11. *The p-cuts of (*, $\mathcal{SAS}_\varepsilon$, *) relations are crisp antisymmetric relations for $p \geq \varepsilon$.*

One can easily construct an example that will be sufficient to prove the following proposition.

Proposition 12. *The p-cuts of weakly antisymmteric L–fuzzy relations are not always crisp antisymmetric relations. The p-cuts of (*, $\mathcal{TRA}_{min-max}$, *) relations are not always transitive in the crisp sence.*

5 Conclusion

In this paper we first surveyed pervious works in the field of decomposition and synthesis of lattice-valued fuzzy relations. In the sequel we surveyed some alternative definitions of the notions of reflexivity, antisymmetry and transitivity, and proposed a taxonomy triplet for fuzzy relations. In the fourth section, we gave propositions on decomposition and synthesis of fuzzy relations regarding some of the given alternative definitions of the special properties defined as in section 3.

The research presented in the paper reperesents a contribution to theory of fuzzy sets, and can be applied in solving various problems and generalizing existing solutions in several various areas, such as: theory of fuzzy lattices [15], theory of information [13, 20], medicine [16, 17], student evaluation [18, 19], robotics [8, 9, 10], and other areas out of the narrowest scope of our own research interests.

References

1. Dubois D, Prade H (1980) Fuzzy sets and systems: Theory and applications., Academic Press, San Diego.
2. Goguen J A (1967) "L-fuzzy sets", J. Math. Appl. 18, 145-174.
3. Gupta G C, Gupta R K (1996) " Fuzzy equivalence relation redefined", Fuzzy sets and systems, 79, 227-233.
4. Janeva B (1996) Introduction to set Theory and Mathematical Logic. Faculty of Nat.Sci. And Math., Skopje (in Macedonian).
5. Kaufman A (1975) Introduction to the Theory of Fuzzy Subsets, Vol. 1: Fundamental Theoretical Elements. Academic Press, New York.

6. Klir G J, Yuan B (1995) Fuzzy sets and fuzzy logic: Theory and applications. Prentice Hall, Upper Saddle River.
7. MacLane S, Birkhoff G (1979) Algebra, 2nd. ed. Coller Macmillan, New York.
8. Stojanov G, Božinovski S, Trajkovski G (1997) "The Status of Representation in Behavior Based Robotic Systems: The Problem and a Solution," Proc. SMC'97, Orlando, USA (to appear).
9. Stojanov G, Božinovski S, Trajkovski G (1997) "Interactionist-Expectative View on Agency and Learning," IMACS J. Math. and Computers in Simulation, North-Holland, Amsterdam, The Netherlands.
10. Stojanov G, Trajkovski G, Božinovski S (1997) "Representation vs Context: A False Dichotomy," European Congres on Cognitive Science – Workshop on Context, Manchester, United Kingdom, 227-230.
11. Šešelja B and Tepavčević A (1994) "Partially ordered and relational valued fuzzy relations II". Review of Research, Mathematical Series, Faculty of Science in Novi Sad, 24(1), 245-260.
12. Šešelja B and Tepavčević A (1995) "Partially ordered and relational valued fuzzy relations I". Fuzzy Sets and Systems, 72, 205-213.
13. Šešelja B and Tepavčević A (1990) "On a construction of codes by P–fuzzy sets". Review of Research, Mathematical Series, Faculty of Science in Novi Sad, 20(2), 71-80.
14. Trajkovski G (1997) Fuzzy Relations Decomposition and Synthesis Theorems. Faculty of Nat.Sci. and Math., Skopje (seminar work, in Macedonian).
15. Trajkovski G (to appear) Fuzzy Relations and Fuzzy Lattices, MSc Thesis, Faculty of Nat.Sci. and Math., Skopje (in Macedonian).
16. Trajkovski G, Stojanov G, Božinovski S, Božinovska L, Janeva B (1997) Fuzzy Sets and Neural Networks in CNV Detection, Proc. ITI'97, Pula, Croatia.
17. Trajkovski G, Stojanov G, Božinovski S (1997) "Application of Fuzzy Sets in CNV Detection During the DCNV Experiment," Proc. 7th IFSA Congress, Prague, Czech Republic.
18. Trajkovski G, Janeva B (1997) "Towards a Standardized Personal Fuzzy Criterion for Student Evaluation," Proc. 7th IFSA Congress, Prague, Czech Republic.
19. Trajkovski G (1996) "Towards a Standard Fuzzy Criterion," Proc. Int. Summer School and Workshop: Intelligent Technologies and Soft Computing, Mangalia, Romania (to appear).
20. Vojvodić G (1994) "Some remarks on errors in the class of block-codes which correspond to L-valued fuzzy set". Review of Research, Mathematical Series, Faculty of Science in Novi Sad, 24(2), 89-93.
21. Yeh R T (1973) "Toward an algebraic theory of fuzzy relational systems". Proc. Int. Congr. Cybern., Namur, 205-223.
22. Zadeh L A (1965) "Fuzzy sets". Information and control, 8(3), 338-353.
23. Zadeh L A (1971) "Simmilarity relations and fuzzy orderings". Information sciences, 3(2), 159-176.

This article was processed using the LaTeX macro package with LLNCS style

Late Papers

PROCEEDINGS OF THE VIII CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '97
NOVI SAD, YUGOSLAVIA, SEPTEMBER 1–4, 1997, PP. 225–230

On construction of maximal coequality relation and its applications

Daniel Abraham Romano

University of Banja Luka, Faculty of Sciences
Department of Mathematics and Informatics
78000 Banja Luka, Mladena Stojanovića 2
Republic of Srpska - Bosnia and Herzegovina
e-mail: daniel@urcbl.bl.ac.yu

Abstract. Relation q is a coequality relation on a set R with apartness if and only if it is consistent, symmetric and cotransitive. If e is an equality relation on R, then e and q are compatible if and only if $(x,y) \in e \wedge (y,z) \in q \Rightarrow (x,z) \in q$. In this case is $e \subseteq \neg q$. In the set theory of constructive mathematics there is an important problem: If e is an equality relation on set R, is there a maximal coequality relation q on R compatible with e? We give an affirmative answer to this question. Besides, we give some applications of that result in constructiv algebraic theories.

AMS Subject Classification (1991): 03F55, 04A05

Key Words and Phrases: constructive mathematics, coequality relation, compatibility, classes of coequality relation, strongly extensional subset, maximal coequality relation

Introduction and preliminaries

This investigation is in constructive mathematics [1, 4, 8, 11, 12].

Coideals of commutative ring $R = (R, =, \neq, +, 0, \cdot, 1)$ with apartness were first defined and studied by W. Ruitenburg in 1982 [11]. After that, coideals (anti-ideals) were studied by A. S. Troelstra and D. van Dalen in their monograph [12]. The author proved, in his paper [5], if S is a coideal of a ring R, then the relation q on R defined by $(x,y) \in q \Leftrightarrow x - y \in S$, satisfies the following properties:

(1) $(\forall x \in R)((x,x) \notin q)$ (consistent [4])
(2) $(\forall x,y \in R)((x,y) \in q \Rightarrow (y,x) \in q)$ (symmetric)
(3) $(\forall x,y,z \in R)((x,z) \in q \Rightarrow (x,y) \in q \vee (y,z) \in q)$ (cotransitive [4])
(4) $(\forall x,y,u,v \in R)((x+u, y+v) \in q \Rightarrow (x,y) \in q \vee (u,v) \in q)$
(5) $(\forall x,y,u,v \in R)((xu, yv) \in q \Rightarrow (x,y) \in q \vee (u,x) \in q)$

A relation q on R which satisfies properties (1)–(5) is called *cocongruence* on R [5]. A relation q on a set $(R, =, \neq)$ which satisfies (1)–(3), i.e. which is consistent, symmetric and cotransitive, is called *coequality relation* on R [8]. Coequality relations were first defined and studied by M. Božić and D. A. Romano (1985). After that, coequality relations were studied by the author in several papers (see for example [5, 6, 7, 8]). Let J be an ideal and S be a coideal of the ring R. W. Ruitenburg in his dissertation ([11], page 33) first stated a demand that $J \subseteq \neg S$. This condition is equivalent to the followig condition:

$$(\forall x, y \in R)(x \in J \wedge y \in S \Rightarrow x + y \in S).$$

In this case we say that J and S are *compatible* [5]. W. Ruitenburg, in his dissertation, first stated a question of existence of a coideal compatible with given ideal J. If e is a congruence on R which is determined by J, and if q is a cocongruence on R which is determined by S, then J and S are compatible if and only if

$$(6) \qquad (\forall x, y, z \in R)((x, y) \in e \wedge (y, z) \in q \Rightarrow (x, z) \in q).$$

In general, if e is an equality relation and if q is a coequality relation on a set R we say that they are *compatible* if and only if they satisfy the condition (6). If q is a coequality relation on R, then $\neg q$ is an equality relation on R compatible with q such that $= \subseteq \neg q$ [11]. Opposite, we have a question:

If e is an equality relation on a set R with apartness, does there exist a maximal coequaltiy relation q on R compatible with e?

It is clear that $\emptyset$ is a coequality relation on R complatible with e.

Let f and g be relations on a set $R = (R, =, \neq)$. By $g * f$ we denote the *filled product* [7, 8, 9] of f and g defined by

$$g * f = \{(x, z) \in R \times R : (\forall y \in R)((x, y) \in f \vee (y, z) \in g)\}.$$

This product is nonempty if and only if $D(g) \cup R(f) = R$. The filled product is associative. For $n \geq 2$ by ${}^n f$ we denote the n-th filled powed of f. Put ${}^1 f = f$. The next theorems are main results of this notion.

Theorem 1. **[7], [8, Theorem 2.4]** *Let f be a relation on a set R. Then the relation $c(f) = \bigcap_{n \in \mathbf{N}} {}^n f$ is a cotransitive relation on R.*

Thus follows the main result on construction of coequality relation:

Theorem 2. **[7], [8, Corollary 2.4.1]** *Let f be a relation on a set R with apartness. Then the relation $q(f) = c((f \cup f^{-1}) \cap \neq)$ is a coequality relation on R.*

Now we have a partial answer to the above question of the existance of coequality relation compatible with the given coequality relation e on a set R with apartness:

Theorem 3. [8, Corollary 2.4.2] *Let e be an equality relation on a set R with apartness. Then the relation $q(\overline{e})$ is a coequality relation on R compatible with e.*

In this short note we shall give an answer to the above question. For undefined notions and notations we refer to books [1, 2, 4, 11, 12] and papers [3, 5, 8].

1 A construction of maximal coequality relation

For our answer we need some lemmas:

Lemma 4. [8, Theorem 1.1] *Let q be a coequality relation on a set R with apartness. Then there exists the subfamily $\mathcal{V}(R,q) = \{xq : x \in R\}$ of $\mathcal{P}(R)$ such that:*

$$(\forall y \in R)(\exists Y \in \mathcal{V}(R,q))(y \notin Y),$$
$$(\forall Z \in \mathcal{V}(R,q))(\exists z \in R)(z \notin Z),$$
$$(\forall Y, Z \in \mathcal{V}(R,q))(Y \neq Z \Rightarrow Y \cup Z = R).$$

Corollary 5. *Let q be a coequality relation on a set R with apartness and let $a \in R$. Then the set $aq = \{x \in R : (a,x) \in q\}$ is a strongly extensional subset of R such that $a \notin aq$.*

Proof. It is clear that $a \notin aq$. Let $x \in aq$, i.e. let $(a,x) \in q$ and let y be an arbitraty element of R. Then $(a,y) \in q$ or $(y,x) \in q$. Thus $y \in aq \vee y \neq x$. So, the set aq is strongly extensional subset of R.

Lemma 6. [6], [8, Theorem 2.2] *Let e and q be an equality and a coequality relations on a set $R = (R, =, \neq)$ and let $\mathcal{A}(X,e) = \{xe : x \in R\}$ and $\mathcal{V}(R,q) = \{yq : y \in R\}$ be families of classes of e and q, respectively. Then e and q are compatible if and only if*

$$(\forall x, y \in R)(x \neq y \wedge xe \cap yq \neq \emptyset \Rightarrow xe \subseteq yq).$$

Lemma 7. [6], [8, Theorem 2.3] *Let q_1 and q_2 be two coequality relations on a set R and let $\mathcal{V}(R,q_1)$ and $\mathcal{V}(R,q_2)$ be their families of classes, respectively. Then $q_1 \subseteq q_2$ if and only if*

$$(\forall Y' \in \mathcal{V}(R,q_1))(\exists Y'' \in \mathcal{V}(R,q_2))(Y' \subseteq Y'').$$

For an element a of a set R and for $n \in \mathbf{N}$ we introduce the following notation: $A_n(a) = \{x \in R : (a,x) \in {}^n\overline{e}\}$, $A(a) = \{x \in R : (a,x) \in q(\overline{e})\}$. By the following results we will present some basic characteristics of these sets.

Theorem 8. *Let a be an element of a set R and let $n \in \mathbf{N}$. Then:*

(6.1) $A_1(a) = \{x \in R : (a,x) \notin e\}$,
(6.2) $A_{n+1}(a) \subseteq A_n(a)$,
(6.3) $A_{n+1}(a) = \{x \in R : R = A_n(a) \cup A_1(x)\}$,
(6.4) $A(a) = \bigcap_{n \in \mathbf{N}} A_n(a)$,

(6.5) The set $A(a)$ is a maximal strongly extensional subset of R such that $a \notin A(a)$.

Proof. (2) Let $x \in A_{n+1}(a)$ i.e. let $(a,x) \in {}^{n+1}\overline{e}$. Then $(\forall s \in R)((a,s) \in {}^{n}\overline{e} \vee (s,x) \in \overline{e})$. Thus $(a,x) \in {}^{n}\overline{e}$ because $(x,x) \notin e$. So, $x \in A_n(a)$.

(3) $x \in A_{n+1}(a) \Leftrightarrow (a,x) \in {}^{n+1}\overline{e} \Leftrightarrow (\forall s \in R)((a,s) \in {}^{n}\overline{e} \vee (s,x) \in \overline{e}) \Leftrightarrow (\forall s \in R)(s \in A_n(a) \vee s \in A_1(x)) \Leftrightarrow R = A_n(a) \cup A_1(x)$.

(5) Let $x \in A(a)$ i.e. let $(a,x) \in q(e)$ and let y be an arbitrary element of R. Then $a \neq x$ and $(a,y) \in q(e)$ or $(y,x) \in q(e)$. Thus $y \in A(a)$ or $y \neq x$. So, the set $A(a)$ is strongly extensional subset of R such that $a \notin A(a)$.

Let T be a strongly extensional subset of R such that $a \notin T$. Let $t \in T$ and let (u,v) be an arbitrary element of e. Then $t \neq v \vee v \in T$. Thus $t \neq v \vee v \neq a$, i.e. $(a,t) \neq (u,v) \in e \vee (a,t) \neq (v,u) \in e$. So, $t \in A_1(a)$. Therefore, $T \subseteq A_1(a)$. Assume that $T \subseteq A_n(a)$. Let t be an arbitrary element of T, let z be an arbitrary element of R and let (u,v) be arbitrary element of e. Then $t \neq z \vee z \in T$. If $t \neq z$ then $t \neq u \vee u \neq z$ i.e. $(t,z) \neq (u,v) \in e \vee (t,z) \neq (v,u) \in e$. Therefore we have that $z \in A_1(t) \vee z \in T$. So, $S = A_1(t) \cup A_n(a)$ i.e. $t \in A_{n+1}(a)$. This means that $T \subseteq A_{n+1}(a)$. Thus, by induction, we obtain that $A_n(a) \supseteq T$ for every $n \in \mathbf{N}$. Whence $A(a) \supseteq T$. Hence, $A(a)$ is a maximal strongly extensional subset of R such that $a \notin A(a)$.

We shall conclude the section with the following result:

Theorem 9. *Let e be an equality relation on a set R with apartness. Then the relation $q(\overline{e}) = \bigcap_{n \in \mathbf{N}} {}^{n}\overline{e}$ is a maximal coequality relation on R compatible with e.*

Proof. Let e be an equality relation on a set R with apartness. Then $q(\overline{e})$ is a coequality relation on R compatible with e, by theorem 3. Let q be a coequality relation on R compatible with e and let a be an arbitrary element of R. Then by corollary 5, aq is a strongly extensional subset of R such that $a \notin aq$. Thus $aq \subseteq A(a)$ because $A(a)$ is a maximal strongly extensional subset of R such that $a \notin A(a)$. By lemma 7 we have that $q \subseteq q(e)$. Therefore $q(e)$ is maximal coequality relation on R compatible with given equality relation e.

2 Applications

1. Let $G = (G, =, \neq, +, 0)$ be an Abelian group with apartness where the group operation is strongly extensional in the next sense:

$$(**) \qquad (\forall a,b,x,y \in G)(a + x \neq b + y \Rightarrow a \neq b \vee x \neq y).$$

Firstly we have:

Theorem 10. *Let q be a coequality relation on an Abelian group G with apartness. Then the relation $q^* = \{(x,y) \in G \times G : (\exists a \in G)((x + a, y + a) \in q)\}$ is a cocongruence relation on G which is a minimal extension of q.*

As an application of theorem 9 we have:

Theorem 11. *Let X be a subgroup of G. Then there exists a maximal cosubgroup $S(X)$ compatible with X.*

Proof. Let X be a subgroup of G. Then the relation e on G defined by $(x, y) \in e \Leftrightarrow x - y \in X$ is a congruence on G. Thus, by theorem 9, the relation $q(\overline{e})$ is a maximal coequality on G compatible with e. On the other hand, the relation $q(\overline{e})^*$ is a cocongruence on G which is extension of $q(\overline{e})$. Hence $q(\overline{e})^* = q(\overline{e})$. So, the set $S(X) = \{x \in G : (x, 0) \in q(\overline{e})\}$ is a maximal cosubgroup of G compatible with X.

2. Let $R = (R, =, \neq, +, 0, \cdot, 1)$ be a commutative ring with apartness $(0 \neq 1)$ where the ring operations are strongly extensional in the next senses:

$$(\forall a, b, x, y \in G)(a + x \neq b + y \Rightarrow a \neq b \vee x \neq y),$$
$$(\forall a, b, x, y \in G)(ax \neq by \Rightarrow a \neq b \vee x \neq y).$$

In the next two theorems we will give a construction of maximal coideal of a commutative ring R compatible with given ideal J.

Theorem 12. [7] *Let q be a coequality relation on R. Then the relation $q^* = \{(x, x) \in R \times R : (\exists a, b \in R)((ax + b, ay + b) \in q)\}$ is a cocongruence on R such that $q \subseteq q^*$. If s is a cocongruence on R such that $q \subseteq s$ then $q^* \subseteq q$.*

Theorem 13. *Let J be an ideal of a commutative ring with apartness. Then there exists a maximal coideal $S(J)$ compatible with J.*

Proof. Let J be an ideal of R. Then the relation $e = \{(x, y) \in R \times R : x - y \in J\}$ is a congruence on R. Thus the relation $q(\overline{e})$ is a maximal coequality relation on R compatible with e. So, $q(\overline{e})^* = q(\overline{e})$. Therefore the set $S(J) = \{a \in R : (a, 0) \in q(\overline{e})\}$ is a maximal coideal of R compatible with J.

References

1. E. Bishop, *Foundations of Constructive Analysis*, McGraw-Hill, New York 1967
2. S. Bogdanović, M. Ćirić, *Polugrupe*, Prosveta, Niš 1993
3. M. Ćirić, S. Bogdanović, *Theory of greatist decompositions of semigroups (a survey)*, Filomat, 9:3 (1995), 385–426
4. R. Mines, F. Richman, W. Ruitenburg, *A Course of Constructive Algebra*, Springer, New York 1988
5. D. A. Romano, *Rings and fields, a constructive view*, Z. Math. Logik Grundl. Math., 34(1)(1988), 25–40
6. D. A. Romano, *Some remarks on coequality relations on sets*, In Proc. of the 6th conf. "Algebra and Logic", Maribor 1989, D. Pagon ed., Znanstv. rev., 2(1)(1990), 111-116
7. D. A. Romano, *Note on two problems in commutative coideal theory*, Scientific Review, Ser. Math., 19-20 (1996), 91–96

8. D. A. Romano, *Coequality relations, a survey*, Bull. Soc. Math. Banja Luka, 3(1996), 1–35
9. R. Milošević, D. A. Romano, M. Vinčić, *Note on two problems in constructive set theory*, In Proc. 2nd math. conf. Priština 1996, Lj. Kočinac ed., Univ. of Priština, Faculty of Sciences, Dept. of Mathematics, Priština 1997, 49-54
10. D. A. Romano, *A construction of maximal coequality relation and its applications*, (to appear)
11. W. Ruitenburg, *Intuicionistic algebra*, Ph. D. Thesis, Univ. of Utrecht, Utrecht 1982
12. A. S. Troelstra, D. van Dalen, *Constructivism in mathematics, an introduction*, North-Holland, Amsterdam 1988

This article was processed using the LaTeX macro package with LLNCS style

Proceedings of the VIII Conference on
Logic and Computer Science LIRA '97
Novi Sad, Yugoslavia, September 1–4, 1997, pp. 231–238

Repetitive applications of function as argument in programming languages

Predrag Stanimirović, Svetozar Rančić, Milan Tasić

Faculty of Philosophy, Department of Mathematics
Ćirila i Metodija 2, 18000 Niš, Yugoslavia

Abstract. We investigate possibility of repetitive applications of a function, given as argument, in different programming languages: MATHEMATICA, LISP, APL, HASKELL. Different approaches used in these presented languages are compared and classified. A few generalizations are stated.

AMS Subject Classification (1991): 68N15

1 Introduction

Perhaps the most important example of repetitive applications of functional argument is the so-called function mapping.

Also, we investigate repetitive applications of an argument of type function to parts of lists or expressions: applications to specified or unknown parts, and applications to specified levels of the expression or list.

The next case is iterative applications of a function as argument, determined by a given scalar argument, and based on the fixed-point style of computation.

Moreover, implicit repetitive calls of a selected function, f, given as argument, is assumed in making tables of values of f. The arguments to the function f can be supplied on many different ways, and resulting arrays are of different dimensions.

Reduction of a vector by an operation is to apply the operation "across" the vector. The reduction operator repeatedly applies dyadic function which performs the reduction.

Our main aim is to revise different approaches to repetitive applications of a function, given as argument to an applicative operator, in different programming languages: MATHEMATICA [7], [12], LISP [1–3], [9–11], APL [8] and HASKELL [4, 5]. We classify these approach using a few adequate criteria, firstly introduced in this paper. Finally, several generalizations are proposed.

2 Mappers

In general, a mapping function, or a mapper for short, is a function that has ability to sequentially apply another function f to several lists of the actual arguments for f. Each argument list corresponds to one parameter of f. There should be as many of these lists as the function argument f requires. The mapper works by the following algorithm: it picks one item from each argument, and supplies these items to f and somehow processes the result of the call. The processing method is specific for each particular mapper. The mapper then advances each argument list to its *cdr*, and the process repeats until the ends of all argument lists are reached.

COMMON LISP [3], [6], [11] and WALTZ LISP [2] support two common types of mapping functions: element mappers and tail mappers. These mappers differs only in the way in which the arguments to the function argument are supplied. Element mappers pick one element at a time from each of the element lists, while the tail mappers supply the entire remainder of each list. The mapping functions in LISP occur in pairs having the same processing function, one being an element mapper and the other a tail mapper.

In MATHEMATICA and HASKELL are built-in only the element mappers.

2.1 Element mappers

LISP contains heterogeneous set of element mappers. Element mapper *mapcar* expects at least two arguments - a function and at least one list - and it apply the function to each element of the list, returning a list of the results. *mapcan* repeatedly applies a function to successive elements of one or more argument lists, but rather than creating a list of all the results, it destructively appends them all together using *nconc*. *mapcar* is useful for collecting a list of results, *mapcan* is useful for filtering out specific elements of a list. Sometimes, the values returned by *mapcar* are less interesting than the side-effect that transpire: the mapper *mapc* returns the value of the last application of the used function.

The generic definition of element mappers is [2]

```
(defun ELEMENT-MAPPER (f arg1 ... argN)
   (process-func
    '( (funcall f (car (nth 1 arg1))...(car (nth 1 argN)))
                  ... ... ...
      (funcall f (car (nth k arg1))...(car (nth k argN))))))
```

where $arg1, \ldots, argN$ denote arbitrary lists of the length k, which contain arguments of the functional argument f.

mapcar is an element mapper with $process-func = list$ in generic mapper definition. *mapc* and *mapcan* are the element mappers with $process-func = progn$ and $process-func = nconc$, respectively.

Although the function *subset* is not a true mapper, it uses similar mechanisms [2]. The result of the expression `(subset arg1)` is the list of those elements from $arg1$ for which f returned a non-*nil* result.

SCHEME [9] provides two mapping functions: *map* and *for−each*, applicable in the form: `(map function list)` and `(for-each function list)`

The mapper *map* is an equivalent to *mapcar*. *for − each* is more useful when primary intent is to iterate over a list applying some function, given as argument, for its side effect only. The result is $\#!TRUE$ if all applications of this function return non–*nil* values, and *nil* otherwise.

MATHEMATICA.

`Map[f, {a, b, ... }]` applies monadic function f to each element in a list, giving $\{f[a], f[b], ...\}$. You can use *Map* on any expression, not just a list, because of a list in MATHEMATICA is treated as a particular expression whose head is *List*. Using $head[x, y, ...]$ as a prototype of MATHEMATICA expression, we can write

$$Map[f,\ head[x, y, ...]] = head[f[x], f[y], ..].$$

`MapThread[f, {{`a_1`, `a_2`, ...}, {`b_1`, `b_2`, ... }, ...}]` gives the resulting list $\{f[a_1, b_1, ...], f[a_2, b_2, ...], ...\}$. It is analogous to the LISP's mapper *mapcar*.

`Select[list, f]` applies f as a criterion to each element of *list* in turn, and keeps only those for which the result is *True*. In this form `select` is equivalent to `subset`. The object *list* can have any head, not necessarily *List*, so that `Select[expr, f]` selects the elements in *expr* for which the function f gives *True*. The function *Select* is analogous to the LISP's function *subset*, for monadic functions.

`Scan[f, expr]` applies f to each part of *expr*, but does not construct a new expression.

HASKELL.

`map f xs` applies f to each element of xs: `map f xs == [f x|x← xs]`. It is an equivalent of the function `Map` in MATHEMATICA.

filter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate, i.e. `filter p xs == [x|x← xs, p x]`. The expression `filter p list` is equivalent to `Select[list, p]`

2.2 Tail mappers

There is also a set of mapping functions that apply a function to the whole list given as an argument to the mapping function, and then to successive *cdrs* of these arguments.

The generic definition of tail mappers is [2]

```
(define (TAIL-MAPPER f arg1 ... argN)
   (process-func
    '( (funcall f (nth 1 arg1)...(nth 1 argN))
       ...  ...  ...
       (funcall f (nth k arg1)...(nth k argN)))))
```

The mapping functions in LISP occur in pairs having the same processing function, one being an element mapper and the other a tail mapper.

In [11] the tail mappers are treated as more primitive. However, in [1] the element mapper *mapcar* is defined in terms of *maplist*.

3 Applications to parts of lists and expressions

One problem with the various mapping functions is that there is no way to stop them before they have run to completion. Sometimes we want to combine flow of control and function mapping. In this way, a function as argument can be applied to parts of lists and expressions.

3.1 Applications to selected elements

MATHEMATICA. The parts of list or expression, used as arguments, can be selected by indices.

`MapAt[f, expr, {{`i_1`, `j_1`, ...}, {`i_2`, `j_2`, ...}, ...}]` applies f to parts of $expr$ at several positions $expr[[i_1, j_1, ...]]$, $expr[[i_2, j_2, ...]]$, ...

Also, the parts of list or expression, where the function is applied, can be specified by any property.

`Thread[f[args], h]` threads f over any objects with head h that appear in $args$.

`Thread[f[args], h, n]` threads f over any objects with head h that appear in the first n $args$ with head h.

`Thread[f[args], h, -n]` threads f over the last n $args$ with head h.

`Thread[f[args], h, {m,n}]` threads f over arguments m through n.

3.2 Applications to selected levels

MATHEMATICA. Level specifications allow you to tell Map to which levels of parts in an expression you want a function applied:

n	levels 1 through n
$Infinity$	all levels
$\{n\}$	level n only
$\{n_1, n_2\}$	levels n_1 through n_2
$Heads->True$	include heads of expressions
$Heads->False$	do not include heads of expressions.

`Map[f, expr, level]` applies f to parts of $expr$ specified by $level$. The default value for $level$ is $\{1\}$.

`MapThread[f, {expr1, ...}, level]` applies f to the parts of $expr1$, ... specified by $level$.

`MapAll[f, expr]` or `f // @ expr` applies f to every subexpression in *expr*. It is equivalent to `Map[f, expr,{0,Infinity}]`

`MapAll[f, expr, Heads->True]` applies f inside the heads of the parts of *expr*.

`MapIndexed[f, expr, level]` apply f to parts at specified levels, giving the list of indices for each part as successive second arguments to f.

`Scan[f, expr, level]` applies f to parts of *expr* on levels specified by *level*, but does not construct a new expression.

3.3 Applying function to unknown part of list or expression

Functional argument of these functionals is applied as long as it is necessary to do so.

MATHEMATICA.

`Select[expr, f, n]` select the first n elements in *expr* for which the function f gives *True*.

LISP. The mapper *some* applies the function to successive elements of the list(s) until the function returns non-*nil*. Then it returns this non-*nil* value. It returns *nil* otherwise [11].

The function *every* stops as soon as one of the function applications returns *nil*, in which case *every* returns *nil* as its value [11]. If al the applications return non-*nil*, *every* returns the last non-*nil* value. The generic definition of these functions is given by the generic function for element mappers, with the $process-func = and$ for *every*, and $process-func = or$, for *some* [2].

4 Fixed point style of computation

In this case applicative operators apply arguments of type function repeatedly, using the previous result as argument.

MATHEMATICA. The functional operations *Nest* and *Nestlist* take a function f of one argument, and apply it repeatedly. At each step, they use the result of the previous step as the new argument of f.

`Nest[f, expr, n]` gives an expression with f applied n times to *expr*.

`NestList[f, expr, n]` gives a list of the results of applying f to *expr* n times.

Computations which repeatedly apply a function or process to its own output until the input equals the output are called the fixed-point computations.

`FixedPoint[f, expr]` starts with *expr*, then applies f repeatedly until the result no longer changes.

`FixedPointList[f, expr]` generates a list giving the result of applying f repeatedly, starting with *expr*, until the result no longer changes.

`FixedPoint[f, expr, SameTest->comp]` stop when the function *comp* applied to two successive results yields *True*.

HASKELL.

`iterate f x` returns an infinite list of repeated applications of f to x: `iterate f x == [x, f x, f (f x), ...]`.

LISP. In [10] is defined the procedure implementing fixed-point computations in LISP.:

```
(defun fixedpoint(x f)
   (sub-fixedpoint x (f x)) )
(defun sub-fixedpoint (x y)
   (print x)
   (if (equal x y) x (sub-fixedpoint y (f y)) ) )
```

5 Making tables of values

MATHEMATICA. Arguments for given function argument in multidimensional tables are specified using the standard iterator notation in MATHEMATICA.

`Array[f,n]` generates a length n list of the form $\{f[1], ...f[n]\}$.

`Array[f, dims, origin]` generate a list using the specified index *origin*.

`Array[f, dims, origin, h]` uses head h rather than *List*, for each level of the array.

`Array [f,{`n_1`, `n_2`, ... }]` generate an $n_1 \times n_2 \times ...$ nested list with elements $f[[i_1, i_2, ...]]$.

`Table[f, {imax}]` generates a list of *imax* values of f.

`Table[f,{i,imax}]` generates a list of the values of f as i runs from 1 to *imax*.

`Table[f, {i,imin,imax}]` starts with $i = imin$.

`Table[f, {i,imin,imax,di}]` uses steps *di*.

`Table[f,{i,imin,imax,di},{j,jmin,jmax,dj},...]` generates a multidimensional table.

`Inner[f, list1, list2, g]` generalized inner product

`Outer[f, list1, list2, ... ]` generalized outer product. Takes all possible combinations of elements from *listi*, and combines them with f.

`FoldList[f, x, {`a_1`, `a_2`, ...}]` gives $\{x,\ f[x, a_1],\ f[f[x, a_1],\ a_2], ...\}$.

`Fold[f, x, {`a_1`, `a_2`, ... }]` gives the last element of the expression $FoldList[f,\ x,\ \{a_1,\ a_2,\ ...\}]$.

`MapIndexed[f, expr]` apply f to the elements of the expression *expr*, giving the part specification of each element as a second argument to f.

LISP. Tables of values in LISP can be formed only using composition of applicative operators [6].

6 Reduction

Reduction of a given vector by a dyadic operation is transformation of this vector into a scalar by means of this operation.

APL. In APL we consider the following monadic reduction operators: +/, -/, */, //, *max/*, *or/*, *and/*. For a vector argument, if $\oplus/$ is applied to vector $v = (v_1, \ldots, v_n)$ the result is the scalar [8]

$$(\oplus\ v_1\ (\oplus\ v_2\ (\ldots\ (\oplus\ v_{n-1}\ v_n)\ \ldots\ )))$$

Note that in application of any reduction operator $\oplus/$ the function $\oplus$ is assumed. For a matrix argument, each row is returned as a vector, and the result is the vector of the resulting values.

LISP. The reduction performed by means of the functional *reduce* in COMMON-LISP requires multiplicative applications of the functional argument. In evaluation of the expression `(reduce fun sequence)` the function *fun* is applied to the first two arguments of *sequence*. It is then applied to the resulting value and the next argument, and so on, until all the arguments have been combined by *fun* [11].

HASKELL.

foldl , applied to a binary operator, a starting value, and a list, reduces the list using the binary operator, from the left to right:

```
foldl f z [x1, x2,...,xn] == (...((z 'f' x1) 'f'...) 'f' xn
```

foldl1 is a variant that has no starting value argument.

scanl is similar to *foldl*, but returns a list of successive reduced values from the left:

```
scanl f z [x1,x2,...,xn]==[z, z 'f' x1, (z 'f' x1) 'f' x2,...]
```

scanl1 is similar, again without starting element:

7 Conclusion

Our central task is the systematization of homogeneous approaches to repetitive applications of a function as argument. The systematization is performed applying several useful criteria through the following programming languages: MATHEMATICA [7], [12], LISP [1–3], [10, 11], SCHEME - version of LISP [9], APL [8] and HASKELL [4, 5]. Such a kind of systematization is firstly used in this paper. Main cases, considered in this revision are: A: element mappers and tail mappers, B: repetitive applications to a part of list or expression, C: fixed-point style of computation, D: making tables of values and E: reduction. The goal of this systematization is to make easy and efficient use of the repetitive applications of a function, supplied to an operator. This approach is very suitable in investigation of programming languages.

Moreover, we now in a position to propose a few generalizations.

1. Include in a programming language possibility to set and remove properties from the cases A:–E: to any function. Such a mechanism can be implemented using the *selection of attributes* in MATHEMATICA or *property list* in LISP. **2.** In all of the above described repetitive applications of a function f, used as argument, no explicit mention is made of the order in which the arguments for f are supplied, but it is assumed to be from left to right. It is nevertheless possible to supply them in another, possibly unspecified order, perhaps for the sake of efficiency. This order is usually unimportant, but is crucial if the evaluation involves side-effects. Application of f from the right to left, in the case when the function f is monadic and given as argument of the applicative operator *mapcar*, is considered in [1]. Following this idea, we can modify the generic definition of element and tail mappers to be able to force any order of evaluation of arguments. We suggest three general methods for implementation of this idea:

(i) modify the *process-func*, keeping the order of applications of f;

(ii) modify the order of applications of f, keeping the *process-func*;

(iii) modify the order of applications of f, as well as the *process-func*.

3. Define an universal applicative operator, possessing all of the properties contained in the cases A:–E:.

References

1. Danicic, I., *LISP programming*, Blackwell Scientific Publications, Oxford, London, Edinburgh, Boston, Melbourne 1985
2. Foderaro, J., *The Franz LISP manual*, Franz, Inc., Alameda, California, 1985
3. Henessey, L.W., *Common* LISP, McGraw-Hill Book Company, 1989
4. Hudak, P. and Fasel, J.H., *A gentle introduction to Haskell*, ACM SIGPLAN Notices vol 27 No 5, 1992, 1–53
5. Hudak, P. et al., *Report on the programming language Haskell*, ACM SIGPLAN Notices vol 27 No 5, 1992, 1–157
6. Hyvönen, E. and Seppänen, J., *Introduction to LISP and functional programming*, Moskva, "Mir", 1990 (Russian)
7. Krejić, N., Herceg, Dj., *Matematika i* MATHEMATICA, Računari u univerzitetskoj praksi, Novi Sad, 1993 (Serbian)
8. Samuek, K.N., *Programming Languages, an Interpreter-based Approach*, Addison-Wesley Publishing Company, Inc., 1990
9. Smith, J.D., *An Introduction to Scheme*, Prentice Hall, Englewood Cliffs, New Jersey, 1988
10. Stark, W.R., *LISP, Lore and Logic*, Springer-Verlag, New York, Berlin, Heidelberg, London, Paris, Tokyo, 1990
11. Wilensky, R., *Common LISPcraft*, Norton, New York, 1986
12. Wolfram, S., *Mathematica: a system for doing mathematics by computer*, Addison-Wesley Publishing Co, Redwood City, California, 1991

This article was processed using the LaTeX macro package with LLNCS style

NIS - NAFTNA INDUSTRIJA SRBIJE
SA POTPUNOM ODGOVORNOŠĆU
NOVI SAD, SUTJESKA BR. 1

TELEFON: 021/615-144
FAKS: 021/25-037
TELEKS:14-196

НИС
НАФТНА ИНДУСТРИЈА СРБИЈЕ

Zakonom Republike Srbije osnovano je Javno preduzeće za istraživanje, proizvodnju, preradu i promet nafte i prirodnog gasa NIS - NAFTNA INDUSTRIJA SRBIJE. Kapital sa kojim raspolaže Naftna industrija Srbije je državna svojina Republike Srbije.

DELATNOST NAFTNE INDUSTRIJE SRBIJE JE:

- istraživanje i proizvodnja nafte, prirodnog gasa, podzemnih voda i geotermalne energije
- transport i promet prirodnog i tečnog gasa
- proizvodnja derivata
- inženjering u oblasti naftne industrije
- trgovina na veliko i malo
- spoljnotrgovinski promet
- projektovanje i izgradnja objekata
- turističke usluge i dr.

U SASTAVU NAFTNE INDUSTRIJE SRBIJE SU:

- NIS NAFTAGAS, deo preduzeća za istraživanje i proizvodnju nafte i prirodnog gasa, podzemnih voda i geotermalne energije, Novi Sad, Sutjeska 1,
- NIS GAS, deo preduzeća za transport i promet prirodnog i tečnog gasa, Novi Sad, Radnička 20,
- NIS ENERGOGAS, deo preduzeća za transport i promet prirodnog i tečnog gasa, Beograd, Autoput 11,
- NIS RAFINERIJA NAFTE, deo preduzeća za proizvodnju derivata nafte, Pančevo, Spoljnostarčevačka bb,
- NIS RAFINERIJA NAFTE, deo preduzeća za proizvodnju ulja, Beograd, Pančevacki put 83,
- NIS FABRIKA MAZIVA FAM, deo preduzeća za proizvodnju maziva, Kruševac, Jastrebačka 14,
- NIS JUGOPETROL, deo preduzeća za promet nafte i naftnih derivata, i izvoz-uvoz, Beograd, Milentija Popovića 1,
- NIS NAFTAGAS PROMET, deo preduzeća za promet nafte i naftnih derivata, izvoz-uvoz, Novi Sad, Bulevar Oslobodjenja 27,
- NIS INŽENJERING, deo preduzeća za projektovanje i izgradnju investicionih objekata, Novi Sad, Bulevar Oslobodjenja 37.