
UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCE
INSTITUTE OF MATHEMATICS

# Proceedings of the VI Conference on Logic and Computer Science

LIRA '92

Novi Sad
October 29-31, 1992

**Organizer**

Institute of Mathematics
Faculty of Natural Sciences and Mathematics
University of Novi Sad
Trg D. Obradovića 4
21000 Novi Sad
Yugoslavia
**tel.** (+38-21) 58-136, (+38-21) 58-888
**fax.** (+38-21) 350-458

**Organizing Committee**

Dr Djura Paunić, chairman
Dr Mirjana Ivanović, secretary
Dr Dragoslav Herceg
Dr Dušan Surla
mr Zoran Budimac
Zoran Putnik

**Programme Committee**

Dr Dragan Acketa, University of Novi Sad
Dr Branislav Boričić, University of Belgrade
Dr Siniša Crvenković, University of Novi Sad
Dr Milan Grulović, University of Novi Sad
Dr Petar Hotomski, University of Novi Sad
Dr Mirjana Ivanović, University of Novi Sad
Dr Smile Markovski, University of Skopje
Dr Djura Paunić, University of Novi Sad
Dr Slaviša Prešić, University of Belgrade
Dr MioDrag Rašković, University of Kragujevac
Dr Dušan Surla, University of Novi Sad
Dr Dušan Tošić, University of Belgrade
Dr Ratko Tošić, University of Novi Sad
Dr Živko Tošić, University of Niš
Dr Gradimir Vojvodić, University of Novi Sad
Dr Djordje Vukomanović, University of Belgrade

**Other Referees**

Dr Branko Karan, Institute "Mihajlo Pupin", Belgrade
Dr Duško Katić, Institute "Mihajlo Pupin", Belgrade
Dr B. Lazarević, University of Belgrade
Dr Gordana Pavlović-Lažetić, University of Belgrade
Dr Milan Milosavljević, University of Belgrade
Dr Pavle Mogin, University of Novi Sad
Dr Dušan Velašević, University of Belgrade

# Preface

The first Conference of the Seminar for Logic and Computer Science was held in Novi Sad in 1987 and up to now it has been regularly held though there were no proceedings of the conferences.

The VI Conference of the Seminar for Logic and Computer Science *LIRA 92* took place at the Institute of Mathematics in Novi Sad, on 29th, 30th, and 31st of October 1992. The Conference was well attended. There were 33 communications, 44 participants, 18 papers and 15 abstracts in spite of all difficulties which have befallen Yugoslavia at that time.

This volume contains all accepted papers and abstracts of the Conference. All presented papers were reviewed by at least two members of the Programme Committee and/or by other competent specialists.

We use this opportunity to express our thanks to all members of the Programme Committee for their effort and participation in the organization of the Conference.

Special thanks goes to the sponsors of the Conference:

- University of Novi Sad, and its rector Dr D. Herceg,
- Institute of Mathematics in Novi Sad, and its director Dr A. Takači,
- City of Novi Sad, and its major Dr V. Divjaković,
- EPS JP Elektrovojvodina, Novi Sad, and Miss M. Zrnić,
- Elite Computers, Novi Sad, and Mr Z. Nadlački,
- Sojaprotein, Bečej, and Mr B. Bjekić,
- Š Panonija Komerc, Novi Sad, and Mr M. Milićević,
- Efekt, Sremska Kamenica, and Mr Z. Šlavik.

Novi Sad, April 1993.

Đura Paunić, Ratko Tošić, editors

# Papers

# Abstracts

# Author Index

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 1 - 8

# SOME COMMENTS ON PROGRAMMING IN PROLOG

Dragan M. Acketa and Snežana Matić-Kekić

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

**Abstract**

A number of hints concerning writing programs in PROLOG is pointed to and illustrated by examples. Emphasis is put on the (global) stack problems related to programs of a combinatorial nature.

**Key words and phrases:** translation to PROLOG, stack overflow, backtracking

## 1 Introduction

It is well-known that PROLOG is a declarative programming language based on backtracking search. Motivated by our programming experience, we are making some comments concerning the following related questions:

- translation from a procedural programming language (say, PASCAL) to PROLOG
- advantages of declarative approach with problems of a combinatorial nature
- stack limitations and ways of overcoming stack problems
- possibilities of an immediate control over the backtracking process

The presented examples were tested on ARITY PROLOG, version 4.0.

We use the well-known PROLOG predicates: `member`, `append`, `reverse`, `dec`, `inc`, `minimum`, `sum` (members of a list). (see, e.g. [2,3,4,5]).

## 2 Some hints concerning translation to PROLOG

### 2.1 Output files

The following (non-standard) possibility may be used for writing into output files:
The command `create(Out,'outfile.')` creates an output file called "outfile". Commands of the form `write(Out,...)` are further used for writing into "outfile". It is required that the variable `Out` is always unified (directly or indirectly, by using `Out` as an additional predicate argument) with `Out` in the call of `create` .

## 2.2 For-loop

We sketch two possibilities for simulating PASCAL for-loop of the form:
`for i := min to max do action( i );`

a) ( [7], nrev.ari ) `action :- for( Min, Max, I), action(I), fail.` ,
where the predicate `for` is defined by:
```
for( Max, Max, Max) :- !.      for( Min, Max, Min).
for( Min, Max,  I ) :- inc( Min, Min1 ),  for( Min1, Max, I ).
```

b) The call `for( Min, Max, I)` can be substituted by producing a list, from which the indices of for-loop will be chosen: `interval( Min, Max, Interval)`, `choose( I, Interval)` , where `Interval = [ Min, Min+1, ... , Max ].`

The predicate `interval` can be found in [3], pp.120, while the predicate `choose` is very similar to `member`, without cut in `member(X,[X|T]) :- !.`

## 2.3 Some applications related to the predicate fail

It is well-known that the predicate `fail` may force producing all the answers to a question. Each of these answers may correspond to a combinatorial object from a class of objects which is being generated. We proceed with some other related hints.

### 2.3.1 Counting passes through a fail-based loop

The total number of objects generated by applying `fail` can be determined by applying an outer counter within the working memory. Let one clause of the form "c." be added for each generated new object. The value of the counter `Number` can be determined in the following manner:

```
generate :- generate_and_write_down_new_object, assert( c ), fail.
generate :- count( Number ), write( Number ).
count(N) :- retract( c ), !, dec(N,N1), count(N1).      count( 0 ).
```

If the required combinatorial objects are obtained from a broader class of objects (candidates), then it might be interesting to count the number of unsuccessful attempts, i.e. the number of tested candidates before one good candidate is found. In this case the first clause of `generate` can be replaced by the following one:

```
generate :- generate_candidate( Candidate ),  assert( c ),
   good_candidate( Candidate ), count(Number), write(Number), fail.
```

We used this method to count the number of attempts with the PROLOG program for solving the "5 houses" problem by Lewis Baxter ([4] (Problem 85., pp. 151), [7], zebra.ari). Instead of testing the total of $(5!)^5 = 24.883.200.000$ possibilities for colours, drinks, nationalities, cigarettes and pets, which are corresponding to each one of the five houses, it turned out that the program tested only 119 candidates, each one of which satisfied a set of fourteen constraints. Only one of these candidates was good in the sense that it passed the permutations' test (no overlaps and duplicates).

### 2.3.2 Assignment to a "global" variable

A brute simulation of the PASCAL assignment to a global variable can be performed by using the built-in predicates `retract` and `assert` , with possible use of `fail` in the first case.

Thus the assignment `a[i,j] := x` to a global matrix `a` can be replaced by

```
clean( I, J ),  assert( a( I, J, X) ) ,    where:
clean( I, J ) :-  retract( a(I, J, Y) ), fail.    clean( _, _ ).
```

The second clause of the predicate `clean` may be omitted, provided that one of the following two situations occurs:

a) the call `clean( I, J )` is replaced by `not clean( I, J )` (the name "`clean`" would be also reasonable to be replaced by "`non_clean`" in that case)

b) the calls of `clean` and `assert` are separated into two consecutive clauses with the same head

If we are sure that multiple former calls of the form `assert( a( I, J, _ ) )` were not possible, then `fail` need not be used and the call of `clean` can be completely substituted by the call of `retract`.

### 2.3.3 "Nesting" fail-based predicates

The following example (output of an $m \times n$ matrix) demonstrates a possibility of using nested predicates which use `fail` ("`nwM`" and "`nwR`" are abbreviations for "`non_write_matrix`" and "`non_write_row`" respectively):

```
nwM(M,N) :-  for(1,M,I), not nwR(I,N ), nl, fail.
nwR(I,N) :-  for(1,N,J), a(I,J,AIJ), write(AIJ), fail.
```

A shorter way to write down the same thing would be:

```
nwM(M,N) :-  for(1,M,I), nl, for(1,N,J), a(I,J,AIJ), write(AIJ), fail.
```

# 3 Three approaches to translating to PROLOG

Each loop in PROLOG is activated either by recursion or by use of the predicate `fail`. On the other hand, PASCAL is a procedural language, which supports recursive programming. There are a lot of problems in which the use of PASCAL recursion is arbitrary: it can be used, but need not [1]. Suppose that we have both iterative and recursive PASCAL program for a problem. There are three choices for writing a PROLOG program for the same problem:

1. to simulate an iterative PASCAL program; the main PASCAL loop is replaced by a PROLOG recursion

[1] our experience says that in most situations the iterative versions are more effective. For example, when generating random latin squares with an iterative PASCAL program (see also Section 5.), the stack overflow error has appeared at size 23, while this error was present already at size 13 with a recursive program, based on the same algorithm.

2. to simulate a recursive PASCAL program; the PASCAL recursion is directly replaced by a PROLOG recursion

3. to write a PROLOG program independently of the PASCAL programs; advantages of declarative programming are used

We illustrate these approaches on the example of generating partitions of natural numbers.

## 3.1 Simulating iterative partitions

An iteration for generating partitions is explained by the following example:

Let be given `N = 16` and `Partition = [5,4,1,1,1,1,1,1,1]` of `N`. The lexicographically next partition `Next = [5,3,3,3,2]` is obtained by joining lists `First_part = [5,3]` and `Second_part = [3,3,2]`. `First_part` is obtained by deleting all the 1's from `Partition` and by decreasing by 1 its last element greater than 1 (i.e., 4) to the value `Last = 3`. `Sum` of members of `First_part` is determined (this sum is equal to 8). `Second_part` is obtained by adding summands equal to `Last` until `N` is reached (only the last summand of `Second_part` may be smaller than `Last`).

We give the "shell" of a PROLOG predicate `next_partition`, which performs this main step:

```
next_partition( N, Partition, Next ) :-
     make_first_part( Partition, First_part, Last ),
     sum( First_part, Sum ),
     make_second_part( N, Sum, Last, Second_part ),
     append( First_part, Second_part, Next ).
```

The main loop for generating all the partitions of `N` is simulated by calling `p(N)`:

```
p( N ) :-  p( N,[N] ).
p( N, [1|T] ) :-  write( [1|T] ), !. fail.
p( N, Partition ) :-  write( Partition ), nl,
     next_partition( N, Partition, Next ),   p( N, Next ).
```

## 3.2 Simulating recursive PASCAL partitions

The following recursive PASCAL procedure `p` ([6], pp.116, sec. 6.1.6), called by `p( N, 1, 1 )` generates all the partitions of `N` :

```
procedure p( N, Min, K: integer);
   var   I: integer;
   begin    for  I:=  Min to (N div 2) do
               begin  Partition[K]:= I;  p( N-I, I, K+1 )  end;
            Partition[K]: = N;  write_down( Partition )
   end;
```

The corresonding PROLOG program for generating partitions of N has the following outlook:

```
p( N ) :- p( N, 1, [] ), fail.
p( N, Min, Temp ) :-
   Max is N // 2,  interval( Min, Max,Int ), choose( I, Int ),
   N_I is  N - I,  p( N_I, I, [ I | Temp ] ).
p( N, _  , Temp ) :-
   reverse( [ N | Temp ], Partition), write( Partition), nl.
```

The second and the third row of the second clause correspond to the for-loop and to the action( I ) respectively

Note that the recursive call of p is almost directly simulated from the PASCAL version. The main difference between the PASCAL and PROLOG versions is the way of writing chosen summands ( I ) into partitions. In the first case are for that purpose used array indices (third arguments of p ), while in the second case new summand is written as the head of a list (this implies neccesity for reversing).

### 3.3 A direct construction of partitions in PROLOG

The following PROLOG program generates partitions of N as lists of non-increasing summands. Each partition is generated by iterative addition of a new summand, which is not greater then the last chosen summand [2] (Min) and the complement (Rest) of the current partial sum. The added summand (I) becomes the head of the list with the current unknown tail T (advancing along the partition corresponds to decrease of the tail).

```
p(N) :-  p(N, N, 0, Partition), write(Partition), nl, fail.
p(N, _, N, [])  :- !.
p(N, Min, Sum, [I|T] ) :-
     Rest is N - Sum,  minimum( Min, Rest, Current_max),
     interval(1, Current_max, Int),  choose(I, Int),
     Sum1 is Sum + I,   p(N, I, Sum1, T).
```

## 4 Some experiences with stack limitations

In this section will be described a number of tests with simple PROLOG programs. The purpose of these tests was to determine the (global) stack limitations and the influence of small alterations of the programs (such as, e.g., adding or reordering some conditions and clauses) to these limitations.

The main predicates of these programs have only one natural number ( N ) as an argument. Each test should determine the maximum value $M = \max(\ N\ )$ of N, for which a program works (gives the answer "yes"). If this value is increased by 1, then

[2] to preserve the non-increasing ordering

the execution of the program is aborted, due to the stack overflow. The maximum value of N is effectively determined by using the halving technique.

Two main predicates: g ("good") and b ("bad") were used for tests:

```
g(N) :- g(N,0).    g(N,N) :- !.   g(N,K) :- K1 is K+1,   g(N,K1).
b(N) :- b(N,0).    b(N,K) :- N > K, K1 is K+1, b(N,K1).   b(N,N).
```

Both of these predicates are devoted to execution of the same task: to make the required number N of passes through a loop. The cut condition, respectively the condition N > K, are used for breaking the recursive calls.

"Good" and "bad" version differ in the position of the recursive clause. The corresponding maximal values $M$ were found to be equal to 59377 and 524 respectively [3]. It is interesting that $M$ increased to 580 with the predicate b , when the condition K1 is K + 1 was replaced by the more effective built-in predicate inc(K,K1) . It turned out that the value of $M$ with the predicate g is disk-space-dependent [4].

The values of $M$ with composed calls "c1(N) :- b(N),b(N).", "c2(N) :- g(N),b(N)." and "c3(N) :- b(N),g(N)." have an interesting behaviour:

$M$ with c1 is equal to 261, which is almost exactly half of 524 = $M$ with b ; $M$ with c2 is also equal to 524 (it took $\approx$ 10 sec both for success with N = 524 and for failure with N = 525). $M$ with c3 is equal to 523; however, it took approximately 2 sec, 30 sec, 1 min, 6 min for success with N equal to 250, 500, 512 and 523 respectively, and only 10 sec for failure with N = 524. Explanation: the first call of "bad" b occupies a large part of global stack, and leaves little room for the work of g.

The following two predicates, g3 and g2 [5], are obtained from g by introducing a list argument:

```
g3(N) :- g3(N,0,L), write(L).          g3(N,N,L) :- !.
g3( N, K, T ) :- K1 is K + 1, g3( N, K1, [ "H" | T ] ).

g2(N) :- g2(N,L), write(L).            g2(0,[]) :- !.
g2( K, [ "H" | T ] ) :-  K1 is K - 1, g2( K1, T ).
```

Two more predicates, b3 and b2 , are introduced so that the relationship between the predicates, b3 and g3 , (also b2 and g2 ), is the same as the relationship between the predicates, b and g.

The following table contains the values of $M$ obtained with 20 tests, which included lists. "Good" and "bad" predicates are denoted by g and b. Columns are indexed by triples VWH, where "V" from {2,3} denotes whether the "3" or "2" version of the predicate is used, "W" from $\{w,n\}$ denotes whether or not the call of write is activated, "H" from {1, K, N} denotes the variable which replaces " H " in predicates:

| | 3w1 | 3wK | 3wN | 3n1 | 3nK | 3nN | 2w1 | 2wK | 2n1 or 2nK |
|---|---|---|---|---|---|---|---|---|---|
| g | 4064 | 3251 | 3251 | 4065 | 3252 | 3252 | 5418 | 4064 | 32767 |
| b | 451 | 451 | 451 | 451 | 451 | 451 | 5412 | 4059 | 32767 |

[3] The discussion concerning the ancestor predicate in [2], sec. 2.6.2. and [3], sec. 3.4., says that the order of conditions within a recursive clause may have a much higher influence to the stack overflow than the order of clauses themselves. However, this example shows that this second order may also have a high influence.

[4] we give the following explanation: if N > 32767, then the predicate is operates with real numbers. The tests show that Arity Prolog Interpreter uses disk when manipulating these numbers.

[5] numbers "3" and "2" are equal to the maximal number of arguments

The last column is explained by entering real arithmetic at $2^{15}$. "2" versions seem to be more efficient than the corrersponding "3" versions, especially with `b` predicates or when the writing is excluded. There is no much difference between `b` and `g` predicates of "2" versions. It is interesting that "g3K" and "g3N" versions gave the same results, which differed from the values for "g31" versions (similarly like "2w1" and "2wK"). Also note similar results for "2wK", "3w1" and "3n1" versions of `g`.

A "good-good" composed call of the version "3wN" makes a very small reduction of $M$ in comparision with the single call (3249 versus 3251).

# 5 A way for overcoming stack problems

The main "shell" of a backtracking procedure has the following outlook:

```
REPEAT
  IF forward_condition    THEN                                BEGIN
     Step_forwards ;
     IF  output_condition    THEN
        BEGIN  Output( New_object ); Step_backwards  END   END
  ELSE   Step_backwards
UNTIL  end_condition
```

When implementing a backtracking procedure, the built-in backtracking gives the PROLOG user two important advantages:

- he need not write down the details of `Step_forwards` and `Step_backwards`
- moreover, he avoids `IF-THEN-ELSE` branching on `forward_condition`, which enables overcoming the stack problems like those in the following test with only [6] $M = 722$ :

```
g(N) :- g(N,0).,      where:      g(N,N) :- !.
g(N,K) :- K1 is K+1, K mod 2 =:= 0, !, g(N,K1).
g(N,K) :- K1 is K+1, K mod 2 =:= 1, !, g(N,K1).
```

  This leads to a *space optimal* implementation [7].

Let some combinatorial objects be generated by backtracking and let PROLOG be the used programming language. We are going to describe in more detail how to replace `IF-THEN-ELSE` branching, which leads to a controlled backtracking, by the built-in PROLOG backtracking.

The objects should be represented by lists, which are gradually beeing extended. Each new member `H` is added (as a new head) to the former sublist `T` , provided that `forward_condition` is satisfied. Thus we obtain the following PROLOG "shell":

```
objects :-  new_object( [], L ),  output( L ), fail.
new_object( L, L ) :-  output_condition( L ), !.
new_object( T, L ) :-
      forward_condition( H, T ),   new_object( [H|T], L ).
```

[6] compare with $M > 50000$ with the predicate `g` in Section 4.

[7] the point with such an implementation is the *possibility* of the program execution, disregarding the necessary *time*

### 5.1 Generating latin squares

We illustrate above ideas on the example of generating latin squares, $N \times N$ matrices with elements in the set $S(N) = \{1, ..., N\}$, such that no two elements appear in the same row or column.

Each latin square $A$ may be generated by a backtracking procedure, an elementary step of which is an attempt to fill in a field of the square. The field $(i, j)$ may be filled in only if all the fields $(i', j')$ have already been filled in, where $i' < i$ or $((i' = i)$ and $(j' < j))$ .

Each element $A(i, j)$, $1 \leq i, j \leq N$, is chosen from the set $S(i, j) = W(i, j) - C(i, j)$, where:

$$W(\ i,\ j\ ) \ = \ S(N) \ - \ (\ \bigcup_{k=1}^{j-1} A(i,k) \ \cup \ \bigcup_{k=1}^{i-1} A(k,j)\ )\ ,$$

while the set $C(i, j)$ is determined as follows:

– each `Step_forwards` activated on the position $(i, j)$ adds the new chosen element $A(i, j)$ to the set $C(i, j)$, while each `Step_backwards` makes $C(i, j)$ equal to the empty set.

The `forward_condition` on the position $(i, j)$ is that $S(i, j)$ is non-empty.

Both iterative and recursive PASCAL implementations of this procedure were described in [1].

The PROLOG implementation based on the above "shell" uses:

`output_condition(L)` : length of `L` is equal to $N^2$

`forward_condition( H , T )` : `H` $\in W(i, j)$, where

$1 \leq j \leq N$ and the length of the sublist `T` is equal to $(i - 1) \cdot N + j - 1$.

A consequence of using the built-in backtracking is the replacement of the set $S(i, j)$ by the simpler set $W(i, j)$ with `forward_condition` in the PROLOG version. Each choice of `H` corresponds to a new node of the PROLOG search tree, and the set $C(i, j)$ is not neccessary.

## References

[1] Acketa,D.M., Matić-Kekić,S., *An algorithm for generating finite loops, some of their subclasses and parastrophic closures*, Proccedings of the 13th International Conference of ITI-91, 554-558.

[2] Bratko,I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986.

[3] Radovan,M., *Programiranje u Prolog-u*, Informator, Zagreb, 1987.

[4] Coelho,H., Cotta,J.C., *Prolog by Examples*, Springer-Verlag, 1988.

[5] Tošić,D., Protić,R., *Prolog kroz primere*, Tehnička Knjiga, Beograd, 1991.

[6] Tošić,D., Stojković,V., *Programski jezik PASCAL - zbirka rešenih zadataka*, Tehnička Knjiga, Beograd, 1990.

[7] Arity Prolog – a package of demo programs

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 9 - 12

# TOWARDS A SYSTEMATIZATION OF MATHEMATICAL KNOWLEDGE[1]

*Branislav R. Boričić*[2]
*University of Belgrade, Faculty of Economics*
*11000 Beograd, Yugoslavia*

**Abstract.** We propose a revision of the systematization of mathematical knowledge, having in mind some relevant logic principles. The basic requirement of such an approach, which can be called *relativism*, is to relativize each mathematical statement to the minimal mathematical, logical and linguistic context in which the considered statement can be proved. The methodology of the investigation of formal mathematical and logical systems, their complexity, relatioships etc, as well as the different kinds of indepedence results, will be of the particular interest for the realization of such a program.

In order to present our standpoint picturesquely, let us consider an imaginary example. Suppose we have collected the whole existing mathematical texts in one place. The mathematical statements, together with their proofs, are the most important components of those texts. This is the reason why we have to distinguish the proofs and the statements as the notions of particular interest. So, by the *effective mathematical knowledge* we mean the collection of all the written mathematical statements with their proofs. Imagine that one should make a library (or a data basis) of the effective mathematical knowledge, i. e. to make a selection, classification and systematization of the given writings, respecting the following natural claim: to capture as little as possible of the library space, but, nevertheless, to make the whole knowledge accessible. Let us put ourselves in the role of the manager making such a library and having a sufficient number of experts at his

[1] *1991 Mathematics Subject Classification:* 03A05, 00A30

[2] This work was supported in part by Science Fund of Serbia, grant number 0401A.

disposal. Our first problem will be which instructions to give to our cooperators with the aim to realize the following two goals: do not lose anything of our effective knowledge and throw off everything superfluous. Next, the obtained set of statements and proofs should be ordered, classified and systematized so to provide an addequate presenting of the effective mathematical knowledge and its accessibility to the holders.

By all means, the following instructions should not be doubtful: (i) if we have a number of the same statements with the same proofs, then we keep only one statement with its proof; (ii) if we have two statements of the following form: *If A and B, then C.* , and *If A, then C.* , with the same proofs, then we will keep only the second one with its proof.

Note that the first of the above instructions is closely connected to the *contraction rule,* while the second one results from our need to respect one of the basic *relevant logic principles* by which we drop the irrelevant hypotheses of each statement. After using such a procedure we do not expect to use the logical rules of *contraction* and *weakening* any more. It could be an indicator that the kernel of logical principles we use, will present fundamental principles of the *linear logic.*

Our central task is the *systematization* of the effective mathematical knowledge. The goal of systematization is to present explicitly a narrow or, if possible, the minimal *mathematical, logical and linguistic* context in which the considered statement is proved. This means to quote each relevant *mathematical* hypothesis, as well as the whole *logical* and *linguistic* instruments used in formulation and proof of the given statement. Such a kind of information can be obtained by an analysis of the very statement and its proof. For instance, the formal systems containing any part of the mathematical axioms of the Zermelo-Fraenkel set theory, Peano arithmetic, some algebraic structures etc, may be used as possible satisfactory frameworks of the mathematical context. The logical instuments are classified already as formal systems of different variants of constructive, relevant etc logics, as well, while, in connection with the linguistic instruments it would be valuable to know the minimal complexity of the sentences–formulae appearing in the statement under consideration, its proof and the corresponding mathematical and logical context. In such a way we make the necessary *mathematical, logical and linguistic relativization* of the considered

mathematical statement.

Such a kind of systematization will use the methodology developed in the theory of formal mathematical and logical systems, the facts on their complexity, relationships etc. Consequently, the attempt to realize the proposed systematization would stimulate further work in these fields.

A standardization of denotation and definitions of the mathematical and logical systems and the other basic notions would contribute to an easier realization of this systematization. This systematization may cause some modification of the current mathematical subject classification.

It would not be difficult to identify the influences of some traditional or contemporary philosophical and foundational standpoints to our approach, e. g., formalism or reverse mathematics.

And finally, let us say how we imagine a possible partial realization of our program. It would be enough to begin with the claim that each new mathematical result have to be presented in a form containing at least three emphasized terms of reference which relativize precisely the truth of the obtained result to the corresponding mathematical, logical and linguistic context. Note that such a condition cannot be satisfied easily due to the fact that the determination of such a context, in itself, often presents a very complex and valuable mathematical result. Moreover, every mathematical problem can be formulated as a problem of finding the minimal mathematical context in which the corresponding assertions are provable.

## References:

R. Carnap, *The logicist foundations of mathematics,* in P. Benacerraf and H. Putnam, eds., **Philosophy of Mathematics,** *Cambridge Univ. Press,* Cambridge,1964, pp. 41–52.

H. B. Curry, *Remarks on the definition and nature of mathematics,* **Dialectica 8** (1954), pp. 228–233.

S. Feferman, *Working foundations,* **Synthese 62 (1985),** pp. 229–254.

J.—Y. Girard, **Proof Theory and Logical Complexity,** *Bib-*

*liopolis*, Naples, 1987.

G. Kreisel, *Proof theory: some personal recollections*, in G. Takeuti, **Proof Theory**, *(Second edition), North-Holland*, Amsterdam, 1987, pp. 395–405.

J. von Neumann, *The formalist foundation of mathematics*, in P. Benacerraf and H. Putnam, eds., 1964, pp. 61–65.

W. V. Quine, *Truth by convention*, in P. Benacerraf and H. Putnam, eds., 1964, pp. 329–354.

S. G. Simpson, *Reverse mathematics*, in A. Nerode and R. Shore, eds., **Proceedings of the Recursion Theory Summer School**, *Proc. Symp. Pure Math., Amer. Math. Soc.*, 42 (1985), pp. 461–471.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 13 - 18

# Regular Grammar Representation of the Genetic Code

Dragan Bosnacki

Faculty of Mathematics and Natural Sciences,
University "Kiril i Metodij" - Skopje,
Arhimedova 5, 91 000 Skopje, Macedonia

***Abstract:*** The genetic code has a key role in the process of protein synthesis - the very first part of the complex pathway of information flow during the gene expression. A formal description of this process is given by means of formal language theory. An unconventional, more dynamic representation of genetic code is also given, based on the regular grammar. The regular grammar representation is compared with already existing representations and several new aspects of such a representation are considered.

***Keywords:*** Formal language theory, context-free grammar, regular grammar, genetic code, DNA, RNA, protein synthesis

## *1. Introduction*

The protein synthesis represents the first part of an extremely complex pathway of information flow - from the gene to the gene's final effect on the organism as a whole. According to the **central dogma** of molecular genetics we have the well known scheme of information transfer

DNA -> RNA -> protein

which means that, in fact, two processes are involved: transcription - the transfer of information stored in DNA to RNA, and translation - further transfer of genetic information from RNA to protein. The process of translation is governed by the genetic code, which is of main interest in the present article.

The most common representations of genetic code are genetic code table [3, 8] and genetic code circle [8]. A more dynamic representations by means of language recognizer are given in [1]. Continuing in this dynamic manner, a new representation using simple language generative device is proposed here.

The relationship between formal language theory and the phenomena related with informational macromolecules is quite straightforward, particularly with processes of transcription and translation, because of their intrinsic linguistic nature.

In Section 2 the formal description of these two processes is introduced, in

what we follow [1]. In Section 3 we give the representation of genetic code as a context-free grammar that can be reduced to a regular grammar and we discuss several characteristics of representation.

As general references for formal language theory we use [6,7] and as general references for molecular genetics [3].

## *2. A formal description of protein synthesis process*

Let D = {**T, C, A, G**} be the **DNA alphabet**. Then a **DNA language** $D^*$ is the set of all strings of symbols in D having finite length. Similarly, let $R^*$ be a **RNA language** over the **RNA alphabet** R = {**U, C, A, G**}. We denote elements of the sets D and R as **bases**. Transcription is a function

$t : D \rightarrow R$

given by its graph

graph(t) = {(**T,U**), (**C,C**), (**A, A**), (**G,G**)}

The biochemical mechanism of transcription, in particular the process of base-pairing, suggests representation of t as a composition of two functions

$t = t_2 * t_1$

given by their graphs

$graph(t_1)$ = { (T,A) ,(C,G), (A,T), (G,C) }

$graph(t_2)$ = { (T,A) ,(C,G), (A,U), (G,C) }

Function t has a (unique) natural extension to a function

$t : D^* \rightarrow R^*$

over the strings of the DNA language. For each string $x=N_1N_2...N_k$ in $D^*$, where $N_i$ is the symbol in D, we define ( with understanding that the image of empty string is again empty string)

$$t(x) = t(N_1)t(N_2)...t(N_k)$$

i.e. the image of x is a concatenation of images of its components - bases. In an analogous manner the extensions of $t_1$ and $t_2$ may be defined. (For a slightly different approach to the formal definition of the DNA alphabet and DNA language see [4, 5]).

Let the **protein language** $P^*$ be a set of all strings over the **protein alphabet**

P = {Ala, Arg, Asp, AspN, Cys, GluN, Glu, Gly, His, Ileu, Leu, Lys, Met, Phe, Pro, Ser, Thr, Tryp, Tyr, Val}

elements of which we shall call **amino acids**.

**Translation** is a partial function

$g: T \rightarrow P$

with

dom(g) = T - {**UAA, UAG, UGA**}

where T=R is a set of all **triplets**, strings with length 3 over R. We also denote the elements of the set T as **codons**. The set {**UAA, UAG, UGA**} is denoted as punctuation mark set. If we allow the image of each codon in this set to be an empty string, then, analogously as for the translation function, there is an obvious extension of function g over the strings in $T^*$

$g^* : T^* \rightarrow P^*$

and the elements of the punctuation mark set specify amino acid chain termination during the translation performed over a string of $T^*$.

### *3. The genetic code represented as a regular grammar*

The most common textbook representation of the genetic code - the genetic code table - is, in fact, equivalent to the above-defined surjective mapping g: T -> P. We present here a constructive model which express more faithfully the dynamics of the translation process by means of regular grammar.

Namely, let $G = (V_N, V_T, P, S)$ be a grammar, where

$V_N$ = {*S, a, Ala, Arg, Asp, AspN, Cys, GluN, Glu, Gly, His, Ileu, Leu, Lys, Met, Phe, Pro, Ser, Thr, Tryp, Tyr, Val, Stop, Y, Y', U', C', A', G', R*}

is a set of variables (nonterminal symbols) with distinguished start symbol S,

$V_T$ = R = {**U, C, A, G**}

is a set of terminal symbols and finally, *P* is a set of productions which consists of

(1) *S* -> *Y'***UG***a*
(2) *a* -> *Ala* | *Arg* | *Asp* | *AspN* | *Cys* | *GluN* | *Glu* | *Gly* | *His* | *Ileu* | *Leu* | *Lys* | *Met* | *Phe* | *Pro* | *Ser* | *Thr* | *Tryp* | *Tyr* | *Val* | *Stop*

(3) *Met* -> **AUG***a*
(4) *Tryp* -> **UGG***a*
(5) *Ileu* -> **AU***G'a*

(6) *Pro* -> **CC***Ra*
(7) *Thr* -> **AC***Ra*
(8) *Ala* -> **GC***Ra*
(9) *Val* -> **GU***Ra*
(10) *Gly* -> **GG***Ra*

(11) *Phe* -> **UU***Ya*
(12) *Tyr* -> **UA***Ya*
(13) *Cys* -> **UG***Ya*
(14) *His* -> **CA***Ya*
(15) *AspN* -> **AA***Ya*
(16) *Asp* -> **GA***Ya*

(17) *GluN* -> **CA***Y'a*
(18) *Lys* -> **AA***Y'a*
(19) *Glu* -> **GA***Y'a*

(20) *Ser* -> **AG***Ya* | **UG***Ra*
(21) *Leu* -> **UU***Y'a* | **CU***Ra*
(22) *Arg* -> **AG***Y'a* | **GU***Ra*

(23) *Stop* -> **UA***Y'* | **UG***A*

(24) *Y* -> **U** | **C**
(25) *Y'* -> **A** | **G**
(26) *U'* -> **C** | **A** | **G**
(27) *C'* -> **U** | **A** | **G**
(28) *A'* -> **U** | **C** | **G**
(29) *G'* -> **U** | **C** | **A**
(30) *R* -> **U** | **C** | **A** | **G**

Because of their more clear biochemical interpretation we use as a nonterminals symbols consisting of more then one letter and also of a letter with "'", rather then conventionally used capital letters only.

The above-presented context-free grammar can be easily reduced, using the rules (2) and (24)-(30) to eliminate the nonterminals *Y, Y', U', A', C', G',R* and *a*, to the following regular grammar:

$G = (V_N, V_T, P, S)$ - grammar
$V_N = \{S, a\}$ - set of variables with start symbol S
$V_T = R = \{U, C, A, G\}$ - set of terminal symbols
*P* - set of productions

S -> **AUG**a | **GUG**a

| | | |
|---|---|---|
| a -> | {Met} | **AUG**a \| |
| | {Tryp} | **UGG**a \| |
| | {Ileu} | **AUU**a \| **AUC**a \| **AUA**a |
| | {Pro} | **CCU**a \| **CCC**a \| **CCA**a \|**CCG**a \| |
| | {Thr} | **ACU**a \| **ACC**a \| **ACA**a \| **ACG**a \| |
| | {Ala} | **GCU**a \| **GCC**a \| **ACA**a \| **ACG**a \| |
| | {Val} | **GUU**a \| **GUC**a \| **GUA**a \| **GUG**a \| |
| | {Gly} | **GGU**a \| **GGC**a \| **GGA**a \| **GGG**a \| |
| | {Phe} | **UUU**a \| **UUC**a \| |
| | {Tyr} | **UAU**a \| **UAC**a \| |
| | {Cys} | **UGU**a \| **UGC**a \| |
| | {His} | **CAU**a \| **CAC**a \| |
| | {AspN} | **AAU**a \| **AAC**a \| |
| | {Asp} | **GAU**a \| **GAC**a \| |
| | {GluN} | **CAA**a \| **CAG**a \| |
| | {Lys} | **AAA**a \| **AAG**a \| |
| | {Glu} | **GAA**a \| **GAG**a \| |

{Ser} AGUa | AGCa | UGUa | UGCa | UGAa | UGGa |

{Leu} UUAa | UUGa | CUUa | CUCa | CUAa | GUGa |
{Arg} AGAa | AGGa | GUUa | GUCa | GUAa | GUGa |

{Stop} UAA | UAG | UGA

But we rather keep the previous context-free form as a more compact and convenient one for discussing some features of such a kind of representation.

Rules (1) and (25) imply that the well-formed triplet strings always begin with **AUG** or **GUG**. Rules (23) and (25) define punctuation mark set, i.e. the fact that all well-formed triplet strings end with **UAA**, **UAG** or **UGA**. Between the initiation and termination codons, triplets coding for corresponding amino acid can be concatenated according to the production rules (3)-(22) and (24)-(30).

The classic table representation operates with the notion of **degeneracy** of genetic code - the occurrence of more then one codon per amino acid. Although we have somehow reversed the direction representing the genetic code with regular grammar starting with proteins (amino acid strings) and obtaining RNA (codons - base triplets), the degeneracy is reflected in the number of possible derivations starting with a certain amino acid.

From the rules (3) - (22) one can see that codons for the same amino acid differ only in the third base. This is called **wobble** at this site. Only the initiation codon is an exception of this rule, since the wobble appears at the first position, which is reflected in rule (1).

The rules (26) - (28) are included only for the reasons of symmetry, they are never used and, of course, might be omitted without any effect on the derivations.

It is worth to notice that amino acid which have similar chemical properties, also have similar derivations, which, on the other hand, means that degeneracy and the codon scheduling in the genetic code is not at random, but this is beyond the scope of the article.

### *4. Conclusion*

The regular grammar representation of the genetic code is a constructive approach dealing with dynamic aspects of the code and in some way it parallels the results given in [1]. The results presented in this article are only very modest formal achievements and they represent only rough approximation of the processes of information flow during protein biosynthesis. It is almost certain that the answers to many open questions, for example the preprocession of RNA before the translation, will require more powerful language generative devices than context-free grammar, let alone regular grammar. However, it is hoped that our genetic code model will provide the basis for future software development and computer experimentation related with the processes over the informational macromolecules.

*References*

[1] S. Bozinovski, *Scodons and Finite Automaton: New Representations of the Genetic Code*, IEEE Int. Conference on Systems, Man and Cybernetics, Cambridge, USA, pp. 1270-1271, 1989

[2] S. Bozinovski, L. Bozinovska, *Flexible Production Lines in Genetics: A Model of Protein Biosynthesis Process*, Proc. Int. Symposium on The Robotics, Dubrovnik, 1987

[3] E. Gardner, P. Snustand, *Principles of Genetics*, John Wiley and Sons, 1984

[4] T. Head, *Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors*, ?, 1987

[5] T. Head, *Splicing Schemes and DNA*, ?, 1990 ?

[6] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Adison-Wasley, Reading Mass.,1979

[7] D. E. Prather, *Discrete Mathematical Structures for Computer Science*, Houghton Mifflin, Boston, 1976

[8] M.M. Rakocevic, *Geni Molekuli Jezik*, Naucna knjiga, Beograd, 1988

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 19 - 26

# Different Pattern Associator Definitions Affect the Learning of Macedonian Verb Inflexions

Čundeva Katerina
Institut za informatika, 91000 Skopje
Arhimedova, 5, POBox 162
email: pmfketi%nubsk@uni-lj.ac.mail.yu

Abstract

A model that learns Present Tense inflexions according to the person and the number of the subject is presented in parallel with a model that learns to recognize the inflected forms. Activation patterns generated in both patterns are inconvenient for correct training. The amount of uncertainties of the output patterns is observed for different classes of Pattern Associators within training series of different length.

**keywords:** Artificial intelligence, Natural language processing, Neural networks

## 1. Introduction

Slavonic languages have complex morphological structure and are highly inflectional. Unlike the nouns which are morphologically simplified, Macedonian verbs appear in a diversity of forms which represent the syntactical connection of the verb in the sentence (Koneski (1987)). Therefore, the recognition of verb inflexions is crucial moment for the parsing of Macedonian sentence.

Macedonian verb inflexions depend on:

- the person of the subject (first, second, third);
- the number of the subject (singular, plural);
- the final vowels of the third person form in singular and
- the Tense.

These changes can be represented by the network diagrams proposed by Gross and applied for the French language in Silberztein (1989). In Present Tense all inflected forms are generated by nine common elements: *а*, *и*, *е*, *ј*, *ам*, *ш*, *ме*, *те* and *ат* (Fig. 1.). The first three of them are both external and internal, the fourth is embedded, while the last five are only external.

Although the rules for inflexion generation are simple and very strict, it is interesting to observe their acquisition. The most convenient model capable of learning is the Pattern Associator. It takes as input a pattern of activation on its inputs and produces a pattern of outputs based on the modifiable connections linking the input and the output units.

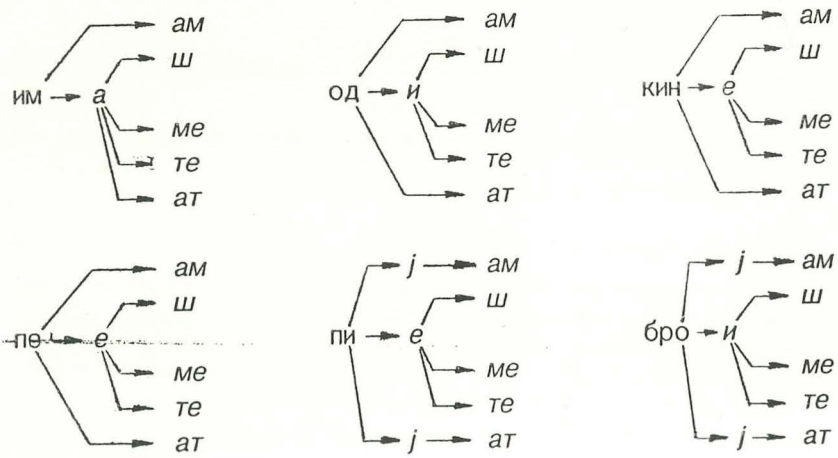



Figure 1. Macedonian verb inflexions in Present Tense

## 2. Definition of the models

The person and the number of the subject determine six personal pronouns: *jas* (I), *ti* (you as singular), *toj* (he), *nie* (we), *vie* (you as plural) and *tie* (they) represented by five binary input units: three for the person, two for the number. Target patterns consist of nine binary units which indicate the presence of the generated inflexion to the corresponding personal pronoun. Here is the pattern file:

| | | |
|---|---|---|
| *jas* | 1 0 0 1 0 | 0 0 0 1 1 0 0 0 0 |
| *ti* | 0 1 0 1 0 | 1 1 1 0 0 1 0 0 0 |
| *toj* | 0 0 1 1 0 | 1 1 1 0 0 0 0 0 0 |
| *nie* | 1 0 0 0 1 | 1 1 1 0 0 0 1 0 0 |
| *vie* | 0 1 0 0 1 | 1 1 1 0 0 0 0 1 0 |
| *tie* | 0 0 1 0 1 | 1 0 0 1 0 0 0 0 1 |

Such definition of the input and target patterns may cause troubles within the process of learning because the input 5-tuples are linearly dependent and the target patterns differ slightly (*toj* vs. *ti*, *nie* or *vie*).

The inflexions themselves are described by seven binary input units. The first six are in fact the vector columns in the first pattern file. They indicate the possibility of connecting the appropriate inflexion to the word kernel defined in Čundeva (1991). The seventh bit is added to point out the difference between the second input pattern *i* and the third, *e*. **Target** patterns are personal pronouns. The pattern file (Fig. 2.) consists of two smaller groups. In the first one three linearly independent 7-tuples generate same target value, while the second consists of six independent 7-tuples and targets with total sum of square differences (further on, tss) not smaller than 2. Macedonian Cyrillic letter *ш* is transliterated into *š* and represented as *{* according to the standard.

| | | |
|---|---|---|
| *a* | 0 1 1 1 1 1 0 | 0 0 1 1 0 |
| *i* | 0 1 1 1 1 0 1 | 0 0 1 1 0 |
| *e* | 0 1 1 1 1 0 0 | 0 0 1 1 0 |
| *j* | 1 0 0 0 0 1 0 | 1 1 0 1 1 |
| *am* | 1 0 0 0 0 0 0 | 1 0 0 1 0 |
| *{* | 0 1 0 0 0 0 0 | 0 1 0 1 0 |
| *me* | 0 0 0 1 0 0 0 | 1 0 0 0 1 |
| *te* | 0 0 0 0 1 0 0 | 0 1 0 0 1 |
| *at* | 0 0 0 0 0 1 0 | 0 0 1 0 1 |

Figure 2. Pattern file for inflexion determination

The **activation** rules determine the output value from the unit according to the value of the net input which comes to it. The Pattern Associator can use:

- *linear* activation when the activation value is same as the net input;

- *linear threshold*

$$o_i = \begin{cases} 1 & \text{net}_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

- *continuous sigmoid* activation according to the function

$$o_i = \frac{1}{1 + e^{-\text{net}_i/T}} \qquad (2)$$

where T is the temperature, usually set to 1 and

- *stochastic* activation with a probability *p* that the output is 1:

$$p(o_i = 1) = \frac{1}{1 + e^{-\text{net}_i/T}} \qquad (3)$$

The learning rules are used to modify the patterns of connectivity. In the very beginning all connections are set to 0, but, after several exposures of the pattern, by obtained experience, they converge to values with minimal tss. The weights $w_{ij}$ specifying the strengths of the connections from input unit $i_j$ to output unit $o_i$ in the Pattern Associator are adjusted by the *Hebbian* algorithm

$$\Delta w_{ij} = \varepsilon o_i i_j \tag{4}$$

or by the *error-correcting delta* rule

$$\Delta w_{ij} = \varepsilon(t_i - o_i)i_j \tag{5}$$

Both network models, the model for inflexion recognition and the other which recognizes personal pronouns according to the inflexions are inconvenient for effective learning by the Pattern Associator. Therefore, it was interesting to compare the results when different activation functions were used with both learning rules. The curriculum patterns were presented in series of different length. The first pattern was exposed 100, 200 and 1000 times, while the second pattern became stabile after 200 learning trials, so the results were obtained from series with 50, 100 and 200 exposures.

## 3. Comparative analyses of the models

The use of the Hebbian rule in all cases produced weight matrices with strictly positive values. When linear, linear threshold and stochastic activation functions were used all matrix series, independently on the learning rates and on the duration of the learning diverged. For any input value, the output value for the inflexion recognition was 111111111. Similarly, in the second model the output values were always 11111.

Continuous sigmoid activation generated convergent matrix series, but the output values were constant: 000000000 for the inflexion recognition and 00000 for the pronoun recognition.

The delta rule was convenient for all activation functions except for the continuous sigmoid. In this case, the output values were constantly zeroes.

### 3.1. Linear activation function

The model was tested with different learning rates. Th rate is a parameter responsible for the changes made to the weights. Its optimal value for the inflexion recognition model with 6 input units is 0.167, but to accelerate the learning the values 0.25, 0.5 and 1 were also used. The last value 1 generated a divergent weight matrix series. In all other cases after 25 exposures weight

matrices got stabilized. Both third person pronouns produced correct output values. The other two person pronouns generated incorrect output values: 011000000 for both singular and 000000000 for both plural pronouns.

The second model was incapable of generating correct output values except for the inflexion *am*. After 50 exposures of the pattern, with learning rates smaller than 0.25 the output values for *j*, *{*, *me* and *te* were 00000, the first three inflexions *a*, *e* and *e* produced an output 00100 and the last one *at* generated the value 00001. When the learning rate was augmented, the output value for *i* become 00200, the outputs for *me* and *te* become 00001, while the output for *at* was set to its initial value 00000.

## 3.2. Linear threshold

After 7 training epoches independently on the learning rates, the weight matrices became stabile. Their elements were varying between -2 and 2. The range was greater for the rates higher than 0.25. The output values for both first person pronouns were different from their target values, while the other four were correct. It is worth noting that the output value for *jas*, 011010000 is closer to the target value for *toj*, while the output 100100100 obtained for *nie* suits better for *tie*. Slightly better recognition was obtained when stochastic sigmoid activation was combined together with the linear threshold output. In this case the output for *jas* remained the same 011010000, but the output for *nie* became correct.

The second model for pronoun recognition got stabile after 7 training epoches and all output values were exact.

## 3.3. Stochastic activation

This is the activation used for morphology acquisition and sentence analysis (Rumelhart & McClelland (1988)). The logistic function depends on a denominator T which scales the net input. In each further iteration previously calculated value x is divided by T. When the divided value exceeds 11.5129, the output value is set to 0.9999, for input value smaller than -11.5129 it is set to 0.0001, otherwise it is calculated according to the equation (3).

The default value for the denominator T is 15. This value produced matrices with big coefficients. Therefore, training was performed with T = 1.

On the other hand, the capability of efficient recognition with a linear threshold activation suggested that the results obtained during the learning with stochastic activation might differ when the linear threshold is suppressed or it is used for correction of the obtained values.

First two tables (Table 1. and 2.) contain the means and the deviations of squared sum differences between the target and the output value for each pronoun with default denominator. Local minima for three training series are marked with *. The means marked with ** correspond to the minima in all series without and with linear threshold output. First pronoun *jas*, similarly to previously used activations generated incorrect outputs. The recognition was getting better when the pattern was exposed more frequentiy. After 1000 iterations with linear threshold output in average 1.4 bits were wrong and the output was matched in less than 50% of the tests. Unexpected was the behaviour of the pronouns *toj*, *nie* and *tie*. Their best recognition was in smaller series rather than after 1000 exposures. The only pronoun with a tendency of complete generation was *vie*. Although the learning of verb inflexions is a morphology problem, it is obvious that this model does not solve it properly.

| | 100 exposures | | 200 exposures | | 1000 exposures | |
|---|---|---|---|---|---|---|
| pronouns | mean | deviation | mean | deviation | mean | deviation |
| *jas* | 2.667 | 1.192570 | 2.333 | 1.349897 | 1.400** | 0.711805 |
| *ti* | 0.933 | 0.442217 | 0.800 | 0.748332 | 0.200* | 0.400000 |
| *toj* | 1.733 | 1.181336 | 1.467* | 0.805536 | 1.667 | 0.788811 |
| *nie* | 2.133 | 0.805536 | 1.333** | 1.011051 | 1.867 | 0.718022 |
| *vie* | 1.200 | 0.748332 | 0.333 | 0.471404 | 0.000** | 0.000000 |
| *tie* | 2.333 | 0.869227 | 2.000 | 0.966092 | 1.933* | 1.062492 |

Table 1. Stochastic behaviour with linear threshold output and T = 15

| | 100 exposures | | 200 exposures | | 1000 exposures | |
|---|---|---|---|---|---|---|
| pronouns | mean | deviation | mean | deviation | mean | deviation |
| *jas* | 2.600 | 0.952191 | 2.133 | 0.884433 | 1.867* | 0.805536 |
| *ti* | 0.533 | 0.805536 | 0.600 | 0.611010 | 0.133** | 0.339935 |
| *toj* | 1.267** | 0.997775 | 2.167 | 1.067187 | 1.333 | 0.942809 |
| *nie* | 1.867 | 1.024153 | 2.600 | 0.879394 | 1.400* | 0.711805 |
| *vie* | 0.867 | 0.884433 | 0.467 | 0.618241 | 0.133* | 0.339935 |
| *tie* | 2.200 | 1.166190 | 1.067** | 0.679869 | 1.333 | 0.869227 |

Table 2. Stochastic behaviour with suppressed linear threshold and T = 15

The second pair of tables (Table 3. and 4.) correspond to the model when T was set to 1. The results were better, particularly for the second person pronouns *ti* and *vie*, and the third person pronoun *tie*, but the other three were generating wrong outputs in more than 50% of the tests. Similarly to previous case, the best recognition for *toj* and *nie* was after 200 iterations.

The recognition was perfect when the ambiguous pronoun *toj* was excluded from the pattern. In less than 100 iterations the model got stabile and all output units were equal to their targets. Then, the obtained model was trained with the excluded pronoun. The model needed 25 iterations to become stabile again. Unfortunately, the pronouns *toj* and *tie* could not be recognized even when the whole pattern was exposed.

| | 100 exposures | | 200 exposures | | 1000 exposures | |
|---|---|---|---|---|---|---|
| pronouns | mean | deviation | mean | deviation | mean | deviation |
| *jas* | 2.353 | 0.680932 | 2.059 | 0.872494 | 2.000* | 0.970143 |
| *ti* | 0.000** | 0.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| *toj* | 1.353 | 0.588235 | 1.471 | 0.775936 | 1.176* | 0.669464 |
| *nie* | 2.353 | 0.680932 | 2.294 | 0.823530 | 2.235* | 0.644379 |
| *vie* | 0.000* | 0.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| *tie* | 0.529 | 0.696010 | 0.529* | 0.605625 | 0.824 | 0.922611 |

Table 3. Stochastic behaviour with linear threshold output and T = 1

| | 100 exposures | | 200 exposures | | 1000 exposures | |
|---|---|---|---|---|---|---|
| pronouns | mean | deviation | mean | deviation | mean | deviation |
| *jas* | 1.824** | 0.616946 | 2.176 | 0.616946 | 1.942 | 0.725225 |
| *ti* | 0.118 | 0.322189 | 0.000* | 0.000000 | 0.000 | 0.000000 |
| *toj* | 1.647 | 0.836039 | 0.941** | 0.937493 | 1.942 | 0.802246 |
| *nie* | 1.824 | 0.616946 | 1.353** | 0.680932 | 2.529 | 0.775936 |
| *vie* | 0.000* | 0.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| *tie* | 0.941 | 0.539127 | 0.824 | 0.705882 | 0.059** | 2.35294 |

Table 4. Stochastic behaviour with suppressed linear threshold and T = 1

Similarly to the previous case, the pronoun *toj* was excluded from the pattern file and trained separately after the stabilization of the model. Finally, the whole pattern was exposed. The recognition remained incorrect for both third person pronouns *toj* and *tie*.

The opposite model which recognizes the pronouns by the inflexions did not need more than 200 training epoches. The first combination with T = 15 and pure stochastic activation (Table 5.) produced right outputs in more than 65% of the tests except for the inflexions *e* (37%), *j* (47%) and *at* (34%). When the denominator was set to 1, the matrices got stabile after 50 epoches and the matching was correct in more than 85% of the tests. The exclusion of the inflexion *j* from the pattern resulted with correct production of the output units after 10 iterations.

| | 50 exposures | | 100 exposures | | 200 exposures | |
|---|---|---|---|---|---|---|
| inflexions | mean | deviation | mean | deviation | mean | deviation |
| *a* | 0.235 | 0.424183 | 0.235 | 0.424183 | 0.000* | 0.000000 |
| *i* | 0.059 | 0.235294 | 0.353 | 0.588235 | 0.000* | 0.000000 |
| *e* | 0.294 | 0.570315 | 0.176* | 0.381220 | 0.636 | 0.481046 |
| *j* | 1.000 | 0.766965 | 1.059 | 0.725225 | 0.636* | 0.642824 |
| *am* | 1.118 | 0.831890 | 0.824 | 0.984306 | 0.000* | 0.000000 |
| *ſ* | 0.883 | 0.757888 | 0.883 | 0.831891 | 0.273* | 0.445362 |
| *me* | 1.059 | 0.937493 | 0.824 | 0.605883 | 0.455* | 0.655555 |
| *te* | 1.118 | 1.022244 | 1.000 | 0.907485 | 0.546* | 0.655555 |
| *at* | 1.353 | 1.233893 | 0.647* | 0.680932 | 0.727 | 0.616576 |

Table 5. Stochastic activation function in pronoun recognition problem

## 4. Conclusion

Macedonian verb inflexion acquisition cannot be well performed in th *pronoun to inflexion* direction. Fortunately, the *inflexion to pronoun* direction i very convenient for fast training, particularly when the linear threshold functio is used together with the error-correcting delta rule. This fact and th bidirectionality of the Pattern Associator imply that the second neural networ will be used for the both tasks: the recognition of the verb inflexions during th sentence parsing and for their generation during the sentence synthesis. In th comparative analysis of all Pattern Associator definitions unexpected was th incapability of the Hebbian rule in producing correct results with any activatio and output function.

The extension of the problem from Present Tense to all Macedonia tenses and clauses involves more than 25 different inflexions which generat more than 200 synthetic and analytic forms (Simov et al. (1990)). Most of ther appear in different tenses and the amount of intersections is big. The number o input units has to be augmented from 7 to more than 15 or even 20, while th number of output units has to remain the same, 5. The disproportion betwee the length of the units can be solved only by division of the network int several smaller, corresponding to particular tense or clause. Another networ will be a connection between these networks and it will point to any of them The input units of this connective network will be the verbs themselve described by their final vowels and by the corresponding inflexion in the thir person singular form. The outputs will point to the tense or the clause.

## References:

Čundeva, K. (1991) "Neural Networks Enable Computers to Translate Natura Languages", in Proceedings of the 13th International Conference on Informatio Technology Interface, Cavtat 1991, ed. by V. Ceric, V. Luzar, R. Paul, Universit Computing Centre, Zagreb, pp. 179-186.

Koneski, B. (1987) "Gramatika na makedonskiot literaturen jazik - del I i II" Kultura, Skopje

Rumelhart, D. E., McClelland J. L. (1988) "Explorations in the Microstructure o Cognition", Volume 2: Foundations, MIT Press

Silberztein, M. (1989) "The Lexical Analysis of French" in Electronic Dictionarie and Automata in Computational Linguistics, ed. by M. Gross and D. Perrin, Berlin Springer Verlag, pp. 93-100.

Simov, K., Angelova, G., Paskaleva, E. (1990) "MORPHO-ASSISTANT: The Proper Treatment of Morphological Knowledge" in Proceedings of COLING '90, Helsinki pp. 455-457.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 27 - 34

# Peirce's law and lambda calculus

Silvia Ghilezan [1]

**Abstract**

The Curry-Howard isomorphism is an explicit connection between simply typed lambda calculus and Heyting's intuitionistic propositional logic. All types inhabited in the simply typed are exactly all intuitionistically provable formulae. It is known that Peirce's law is intuitionistically not valid. By the Curry-Howard isomorphism this corresponds to the fact that Peirce's law is not inhabited in simply typed lambda calculus. We present a direct proof, within the simply typed lambda calculus that Peirce's law is not inhabited. First, we give some structural properties of lambda terms that are inhabitants of certain types. Then according to these properties we show that it is not possible to construct a lambda term that is an inhabitant of Peirce's law.

*Kew words*: inhabitation, lambda calculus, Peirce's law, provability.

## Introduction

*Lambda calculus* and *combinatory logic* introduced in the 30's by Schönfinkel, Curry and Church were originally meant to form a more rigorous basis for the foundation of logic and mathematics. Later on these investigations became rather involved, but still systems obtained in this way present a different approach to logic. Although the first systems were type free, they are in accordance with type introduction and this is nowadays one of the joint fields of investigations both in logic and theoretical computer science.

By changing the notion of the type on the one hand and by changing the notion of the lambda term on the other, it is possible to obtain a whole variety of typed lambda calculi. These systems form the *Barendregt's cube*, see Barendregt, 1992.

The *Curry-Howard isomorphism* given in Howard (1969), 1980, is an explicit connection between simply typed lambda calculus and Heyting's intuitionistic propositional logic. It can be expanded to other constructive logics versus various typed lambda calculi, e.g. second order propositional logic and polymorphic lambda calculus. Also, it can be restricted to substructural logics and restricted typed lambda calculi, e.g. relevant logic and $\lambda$I-typed calculus. It is a powerful connection since by proving facts in logic one proves the corresponding facts in lambda calculus and vice versa.

Some problems of interest in the systems of typed lambda calculi are:

[1]Faculty of Engineering, University of Novi Sad, Trg D. Obradovića 6, 21000 Novi Sad, Yugoslavia

- *Type-checking* - is it decidable whether a given term has a given type?

- *Typability* - is there a type that can be assigned to a given term?

- *Inhabitation* - is there a term of a given type?

All three problems are decidable in simply typed lambda calculus; for other typed lambda calculi some of these questions are still open, see Barendregt, 1992. In the sence of the Curry-Howard isomorphism the problem of inhabitation in simply typed lambda calculus is equivalent to the problem of provability in Heyting's propositional logic, which is known to be decidable. There are direct methods in lambda calculus to prove the decidability of inhabitations, as well.

It is known that Peirce's law is intuitionistically not valid. By the Curry-Howard isomorphism this corresponds to the fact that Peirce's law is not inhabited in the simply typed lambda calculus. We present a direct proof, within simply typed lambda calculus that Peirce's law is not inhabited. First, we give some structural properties of lambda terms that are inhabitants of certain types. Then according to these properties we show that it is not possible to construct a lambda term that is an inhabitant of Peirce's law.

## Simply typed lambda calculus and logic

First, let us recall some basic notions and notations of simply typed lambda calculus, $\lambda \rightarrow$(for more details see Barendregt, 1992).

The set of types $T$ of $\lambda \rightarrow$ is defined in the following way:

**Definition 1.**

*(i)* $\mathcal{V} = \{\alpha, \beta, \gamma, \alpha_1, \ldots\} \subset T$, $\mathcal{V}$ *is a denumerable set of type variables.*

*(ii) If* $\sigma, \tau \in T$, *then* $\sigma \rightarrow \tau \in T$.

Let $\alpha, \beta, \gamma, \alpha_1, \ldots$ be schematic letters for type variables and let $\sigma, \tau, \rho, \varphi, \psi, \sigma_1, \ldots$ be schematic letters for types.

Let $\Lambda$ be the set of *untyped (type–free) lambda terms* defined in the following way:

**Definition 2.**

*(i)* $V \subseteq \Lambda$, $V$ *is a denumerable set of variables.*

*(ii) If* $M, N \in \Lambda$, *then* $MN \in \Lambda$.

*(iii) If* $M \in \Lambda$ *and* $x \in V$, *then* $\lambda x.M \in \Lambda$.

Let $x, y, z, x_1, \ldots$ be schematic letters for term variables and $M, N, P, Q, M_1, \ldots$ schematic letters for lambda terms. The usual notion of $\beta$-reduction on $\Lambda$ is given by the following contraction rule

$$(\lambda x.M)N \to_\beta M[N/x].$$

The expression $M : \sigma$, called *statement*, where $M \in \Lambda, \sigma \in T$ links the terms of $\Lambda$ and the types of $T$. The term $M$ is the *subject* and $\sigma$ is the *predicate* of the statement $M : \sigma$. If $x \in \mathsf{V}$, then $x : \tau$ is a *basic statement.* A *basis* is a set of basic statements.

$\Gamma, \Delta, \Gamma_1, \ldots$ are used as schematic letters for bases. Intuitively, $M : \sigma$ means that the term $M$ is of type $\sigma$.

**Definition 3.** *The Curry version of the simply typed lambda calculus $\lambda \to$ is defined by the following rules*

$$(\textit{start rule})\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$(\to E)\ \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad (\to I)\ \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}.$$

Some properties of $\lambda \to$ that we need later are the Subject reduction property and the Generation lemma.

**Theorem 1.** *(Subject reduction) Let $\Gamma \vdash M : \sigma$ and $M \to_\beta N$, then $\Gamma \vdash N : \sigma$.*

**Theorem 2.** *(Generation lemma) Let $\Gamma \vdash MN : \tau$, then there is a type $\sigma$ such that $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$.*

Further we shall consider provability in classical and intuitionistic propositional logic, so let us recall their natural deduction formulation. It is known that intuitionistic connectives are independent. We shall deal with implication only. In classical logic connectives can be expressed by each other, e.g. $\to$ and $\neg$ can generate all other connectives. The natural deduction formulation of intuitionistic logic with $\to$ and $\bot$ is given by the following elimination and introduction rules

$$\frac{\varphi \to \psi \quad \varphi}{\psi}\ (\to E) \qquad \frac{\begin{array}{c}[\varphi] \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to I)$$

$$\frac{\bot}{\varphi}\ (\bot).$$

Negation is defined by $\neg\varphi = \varphi \to \bot$. Classical logic is obtained by adding the reductio ad absurdum rule

$$\dfrac{\begin{array}{c}\neg\varphi\\ \vdots\\ \bot\end{array}}{\varphi}\ (RAA).$$

Provability is decidable both in classical and intuitionistic propositional logic (for more details see Prawitz, 1965, and van Dalen, 1983).

Peirce's law is provable in classical logic. The following derivation shows that $(RAA)$ is the key point in the derivation of Peirce's law.

$$\dfrac{\dfrac{\dfrac{[(\alpha\to\beta)\to\alpha]^3 \quad \dfrac{\dfrac{[\neg\alpha]^2 \quad [\alpha]^1}{\bot}(\to E)}{\beta}(\bot)}{\alpha\to\beta}1(\to I)}{\dfrac{[\neg\alpha]^2 \qquad\qquad \alpha}{\bot}(\to E)}(\to E)}{\dfrac{\bot}{\alpha}2(RAA)}3(\to I)\ .$$

A proof is in normal form if there is no application of an introduction rule before any application of an elimination rule, i.e. elimination rules are to be applied first, before the introduction rules. If a formula is provable, then it has a proof in a normal form as well.

Peirce's law is intuitionistially not provable, since there is no proof in normal form. If we try to reconstruct a normal proof from bottom up we shall always end with some noncancelled premise, e.g.

$$\dfrac{\dfrac{[(\alpha\to\beta)\to\alpha] \quad \alpha\to\beta}{\alpha}(\to E)}{((\alpha\to\beta)\to\alpha)\to\alpha}(\to I)$$

where $\alpha \to \beta$ is not cancelled.

There is an obvious connection between the logical rules of arrow elimination and introduction and the corresponding rules in $\lambda\to$ in the sense that the lambda term construction is following (coding) the derivation in logic. This connection is called the Curry-Howard isomorphism or the interpretation formulae-as-types terms-as-proofs. It is given by the following statement.

**Theorem 3.** *(Howard (1969), 1980)*

*Let $\varphi \in T$ be given.*

*There exists a inhabitant $M$ of $\varphi$, i.e. $\vdash M : \varphi$, if and only if $\varphi$ is provable in the implicational fragment of Heyting's propositional logic.*

Since inhabitation in $\lambda \rightarrow$ and provability in intuitionistic logic are equivalent the decidability of provability gives an immediate answer to the question of inhabitation in $\lambda \rightarrow$.

## Inhabitation in $\lambda \rightarrow$ and Peirce's law

The following statement is characterizing the lambda terms that are inhabitants of certain types.

**Proposition 1.** *Let $\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \alpha \in T$ and let $\Gamma \vdash M : \sigma$. Then there is a lambda-term $M'$ of the form*

$$\lambda x_1 \ldots x_n . x N_1 \ldots N_m \quad \textit{such that}$$

$$\Gamma \vdash M' : \sigma \quad \textit{and} \quad \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x N_1, \ldots N_m : \alpha.$$

**Proof.** By induction on the derivation of $\Gamma \vdash M : \sigma$ in $\lambda \rightarrow$.

1. If the last step in the derivation is

$$\text{(start rule)} \qquad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \ ,$$

then applying $(\rightarrow E)$ $n$-times, by Subject reduction property (Theorem 1), we obtain

$$\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x x_1 \ldots x_n : \sigma.$$

Thus $M' \equiv \lambda x_1 \ldots x_n . x x_1 \ldots x_n$.

2. If the last step in the derivation is

$$(\rightarrow I) \qquad \frac{\Gamma, y : \sigma_1 \vdash N : \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \alpha}{\Gamma \vdash \lambda y . N : \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \alpha}$$

then by the induction hypothesis there is a lambda-term $N'$ of the form $\lambda x_2 \ldots x_n . x M_1 \ldots M_m$ such that

$$\Gamma, y : \sigma_1 \vdash \lambda x_2 \ldots x_n . x M_1 \ldots M_m : \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \alpha$$

and

$$\Gamma, y : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n \vdash x M_1 \ldots M_m : \alpha.$$

Hence,

$$\Gamma \vdash \lambda y x_2 \ldots x_n . x M_1 \ldots M_m : \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \alpha,$$

and $(\lambda y . N)'$ is $\lambda y x_2 \ldots x_n . x M_1 \ldots M_m$.

3. If the last step in the derivation is

$$(\to E) \qquad \frac{\Gamma \vdash N : \varphi \to (\sigma_1 \to \ldots \to \sigma_n \to \alpha) \quad \Gamma \vdash P : \varphi}{\Gamma \vdash NP : \sigma_1 \to \ldots \to \sigma_n \to \alpha},$$

then by the induction hypothesis there is a lambda-term $N'$ of the form $\lambda y x_1 \ldots x_n . x M_1 \ldots M_m$ such that

$$\Gamma \vdash \lambda y x_1 \ldots x_n . x M_1 \ldots M_m : \varphi \to \sigma_1 \to \ldots \to \sigma_n \to \alpha$$

and

$$\Gamma, y : \varphi, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x M_1 \ldots M_m : \alpha.$$

Now we have to distinguish two cases:

(a) $x \neq y$

(b) $x = y$.

(a) If $x \neq y$, then by $(\to I)$ and of Subject reduction we have the following derivation

$$\frac{\dfrac{\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash \lambda y . x M_1 \ldots M_m : \varphi \to \alpha \quad \Gamma \vdash N : \varphi}{\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x M_1[N/y] \ldots M_m[N/y] : \alpha}\ (\to I)}{\Gamma \vdash \lambda x_1 \ldots x_n . x M_1[N/y] \ldots M_m[N/y] : \sigma_1 \to \ldots \to \sigma_n \to \alpha}\ (\to I$$

hence, $(NP)' = \lambda x_1 \ldots x_n . x M_1[N/y] \ldots M_m[N/y]$.

(b) If $x = y$, then from

$$\Gamma, x : \varphi, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x M_1 \ldots M_m : \alpha$$

by Generation lemma (Theorem 2) it follows that $\varphi = \varphi_1 \to \ldots \to \varphi_m \to \alpha$, nence by the induction hypothesis, since $\Gamma \vdash P : \varphi$ there is a lambda-term $P'$ of the form $\lambda y_1 \ldots y_m . z P_1 \ldots P_k$ such that

$$\Gamma \vdash P' : \varphi$$

and

$$\Gamma, y_1 : \varphi_1, \ldots, y_m : \varphi_m \vdash z P_1 \ldots P_k : \alpha.$$

We can suppose that $x_i \neq y_j$ for all $1 \leq i \leq n, 1 \leq j \leq m$ without lost of generality. Let $\Gamma' = \Gamma, y_1 : \varphi_1, \ldots, y_m : \varphi_m$, then

$$\frac{\Gamma', x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash z P_1 \ldots P_k : \alpha}{\Gamma' \vdash \lambda x_1 \ldots x_n . z P_1 \ldots P_k : \sigma_1 \to \ldots \to \sigma_n : \alpha}\ (\to I);$$

hence, if we take $\Gamma'$ instead of $\Gamma$, and we can do that easily,

$$(NP)' = \lambda x_1 \ldots x_n . z P_1 \ldots P_k. \quad q.e.d.$$

By using this property it is easy to check whether a type with a type variable at rightmost place is inhabited.

The fact that Peirce's law is intuitionistically not provable in terms of lambda calculus means that *Peirce's law is not inhabited.*

This is a consequence of Theorem 3, but it can be shown by application of Proposition 1 as well, for Peirce's law is of the requiered type shape.

**Corollary 2.** *Peirce's law is not inhabited in simply typed lambda calculus.*

**Proof.** Suppose that there is an inhabitant $M$ of Peirce's law, i.e.

$$\vdash M : ((\alpha \to \beta) \to \alpha) \to \alpha.$$

Then by Proposition 1 there is a term $M'$ of the form $M' \equiv \lambda x_1.xM_1 \dots M_m$ such that

$$\vdash M' : ((\alpha \to \beta) \to \alpha) \to \alpha \text{ and } x_1 : (\alpha \to \beta) \to \alpha \vdash xM_1 \dots M_m : \alpha.$$

Thus $x_1 \equiv x$ since $\Gamma = \emptyset$. Then

$$x : (\alpha \to \beta) \to \alpha$$

and $m = 1$. Thus from

$$x : (\alpha \to \beta) \to \alpha \vdash xM_1 : \alpha$$

by Generation lemma,

$$x : (\alpha \to \beta) \to \alpha \vdash M_1 : \alpha \to \beta$$

and $Fv(M_1) \subseteq \{x\}$.

Then again by Proposition 1 there is a term $N'$ of the form $\lambda y_1.yN_1 \dots N_k$ such that $x : (\alpha \to \beta) \to \alpha \vdash N' : \alpha \to \beta$ and

$$x : (\alpha \to \beta) \to \alpha, \quad y_1 : \alpha \vdash yN_1 \dots N_k : \beta.$$

$Fv(N') \subseteq \{x\}$, so there are two possibilities $y \equiv y_1$ or $y \equiv x$.

If $y \equiv y_1$, then $y : \alpha$, so $yN_1 \dots N_k$ cannot be of type $\beta$. It can be typed only in the case of $k = 0$, but still it cannot by typed by $\beta$.

If $y \equiv x$, then $y : (\alpha \to \beta) \to \alpha$. Thus $k = 1$ and $yN_1$ can be only of type $\alpha$, but certainly not of type $\beta$. q.e.d.

## References

[1] Barendregt, H.P., 1992, Lambda calculi with types, in Handbook of Logic in Computer Science, S. Abramsky, D.M. Gabbay and T.S.E. Maibaum eds, Oxford University Press, Oxford.

[2] Dalen, D. van, 1983, Logic and Structure, Springer-Verlag, Berlin.

[3] Howard, W.A., (1969) 1980, The formulae-as-types notion, in To H.B. Curry: Essay on Combinatory Logic, Typed Lambda Calculus and Formalism, eds. J.R. Hindley and J.P. Seldin, Academic Press, New York, pp 479-490.

[4] Prawitz, D., 1965, Natural Deduction, Almgvist and Wiksell, Stockholm.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA 92
NOVI SAD, OCTOBER 29-31, 1992., PP. 35 - 42

# ДЕДУКТИВНЫЙ ПОДХОД К АВТОМАТИЧЕСКОМУ ПОРОЖДЕНИЮ КОМБИНАТОРНЫХ РАСПОЛОЖЕНИЙ

Петар Хотомски
Технички Факултет "М. Пупин"
Дж. Джаковича б.б. 23000 Зренянин

## Р Е З Ю М Е

В статье рассмаривается дедуктивное порождение комбинаторных расположений, обоснованое на методе резолюции. Для задачи распоряжения уроков приводится часть аксиоматической базы и описывается процедура порождения одного из вариантов искомого распоряжения. Дается один простой пример, илюстрирующий предложенную процедуру.

КЛЮЧЕВЫЕ СЛОВА: Комбинаторная структура / метод резолюции / распоряжение уроков

## В В Е Д Е Н И Е

Исследования по автоматическому доказательству теорем осуществляются в двух направлениях:

- разработка и совершенствование методов и подходов к автоматическому выводу,
- отыскание приложений существующих результатов.

К первому направлению относятся исследования и результаты полученные в рамках развития экспертной системы "Graph" /1/. Результаты которые обоснованны на методе резолюции /2/ и на естественном выводе подробно описнны в монографие /3/.

Настоящая статья принадлежит к второму из приведенных направлений, т. е. здесь рассматривается приложение метода резолюции к автоматическому порождению комбинаторных структур.

Понятие "комбинаторной структуры" используем в духе следующего определения.

Пусть X конечное дискретное множество и $A=[a_{ij}]$ матрица nxm. Расположение элементов множества X по ячейкам матрицы A будем рассматривать как отображение множества $S=\{(i,j) \mid i=\overline{1,n}\ ; j=\overline{1,m}\}$ в множество X. При этом, получение пустой ячейки, либо стирание содержания $a_{ij}$ в ячейке (i,j), осуществляется включением особого элемента e в множество X.

Пусть $\mathcal{F} = \{ f \mid f: S \rightarrow X \}$ и $\mathcal{U}$ множество условий, требуемых на множестве $\mathcal{F}$. Пусть $K \subseteq \mathcal{F}$ множество тех и только тех элементов $\mathcal{F}$ для которых выполнено каждое условие из $\mathcal{U}$. Множество K вполне определенно множествами $S, X, \mathcal{F}, \mathcal{U}$. Поэтому скажем что $( S, X, \mathcal{F}, \mathcal{U} )$ определяет одну комбинаторную структуру на множестве S по отношению к множествам X и $\mathcal{U}$. Элементы множества $\mathcal{U}$ будем понимать как аксиомы комбинаторной структуры $\mathcal{K}$. Обозначим $\mathcal{A}(\mathcal{F})$ множество аксиом. Теперь, структуру $\mathcal{K}$ можно определить как аксиоматический объект.

Определение 1. Комбинаторная структура $\mathcal{K}$ на множестве S по отношению к множеству X, это $( S, X, \mathcal{F}, \mathcal{A}(\mathcal{F}) )$, где $\mathcal{F}$ множество отображений $f: S \rightarrow X$, $\mathcal{A}(\mathcal{F})$ множество аксиом.

Так как $\mathcal{A}(\mathcal{F})$ содержит множество условий $\mathcal{U}$, которые могут быть противоречивыми, то при оформлении структуры $\mathcal{K}$ неопходимо предварительно обеспечить выполнимость множества $\mathcal{A}(\mathcal{F})$. Невыполнимость множества $\mathcal{A}(\mathcal{F})$, когда аксиомы выраженны на языке исчисления предикатов, можно установить при помощи метода резолюции. На практике, таким способом можно предусмотреть и устранить условия которые приводят к невыполнимости множества $\mathcal{A}(\mathcal{F})$. Это особенно удобно когда множество $\mathcal{U}$, кроме неопходимых условий, содержит и некоторые желаемые условия. Когда $\mathcal{U}$ включает только неопходимые условия, можно сформулировать теорему об существовании отображения $f \in \mathcal{F}$ по отношению к аксиомам $\mathcal{A}(\mathcal{F})$. Доказательство существования f возможно, хотя в принципе, осуществить также при помощи метода резолюции исходя от $\mathcal{A}(\mathcal{F})$ и отрицания теоремы существования.

Иной подход состоит в использовании метода резолюции в процессе порождения самой структуры. Конкретизируем эту общую концепцию на задаче комбинаторного расположения, которая мотивированна распоряжением уроков.

## 1. ДЕДУКТИВНОЕ ПОРОЖДЕНИЕ КОМБИНАТОРНОГО РАСПОЛОЖЕНИЯ

Рассмотрим задачу распоряжения уроков.

Элементы множества X определенны следующими данными:

имя преподавателя, предмет, класс, индекс урока

Индекс урока дает возможность различать элементы множества X когда преподаватель в томже классе преподает тотже предмет на нескольких уроках. Индексом O определяется элемент e предявляющий

пустую ячейку. В матрице S каждая строка принадлежит одному преподавателю, а каждый столбец обозначает термин урока. Термин урока определяется двумя параметрами: день, очередной номер урока На примерь, ячейка (a,b) отвечает а-тому преподавателю в термине b, при чем b описывается парой (d,t).

Отображение f: S⟶X определяется основным предикатом

D(n,p,q,i,d,t) - преподаватель n держит урок с индексом i по предмету p в классе q, дня d с очередным номером t

Таким образом, заполнение ячейки (a,b) состоит в определении предмета p и класса q в котором преподаватель a держит урок в термине b.

Множество $\mathcal{A}(\mathcal{F})$ содержит:

- общие аксиомы (присущие в каждом распоряжении).
- неопходимые условия (требуемые в конкретном случае),
- желательные условия (не обязательны для выполнения),
- данные.

Общие аксиомы

Приводим концептуальную модель которая не включает разделение класса на группы, обединения классов и блоки уроков. Для илюстрации метода приводим только некоторые из общих аксиом:

1. Аксиомы одновременности

1.1. $p_1 \neq p_2 \Rightarrow ( D(n,p_1,q_1,i_1,d,t) \Rightarrow \neg D(n,p_2,q_2,i_2,d,t) )$

Преподаватель n не держит два разных предмета одновременно

1.2. $q_1 \neq q_2 \Rightarrow ( D(n,p_1,q_1,i_1,d,t) \Rightarrow \neg D(n,p_2,q_2,i_2,d,t) )$

Преподаватель n не держит урок в разных классах одновременно

1.3. $n_1 \neq n_2 \Rightarrow ( D(n_1,p_1,q,i_1,d,t) \Rightarrow \neg D(n_2,p_2,q,i_2,d,t) )$

Различные преподаватели не держат урок в томже классе одноврем.

2. Аксиомы дистанции

2.1. $t_1 \neq t_2 \Rightarrow ( D(n,p,q,i_1,d,t_1) \Rightarrow \neg D(n,p,q,i_2,d,t_2) )$

Преподаватель n не держит тотже предмет в томже классе два раза в течении дня.

Аксиомы выражающие неопходимые условия, а также и аксиомы выражающие желательные условия, выражаются также формулами исчисления предикатов, зависящими от конкретной ситуации.

Данные

Данные разделяются на общие и конкретные.

Общие данные определяют число дней и уроков в течении дня, т.е. определяют число столбцов в матрице. Число строк является

тоже общим данным, но предполагается не меньше числа преподавателей.

Конкретные данные записываются в виде формул P(n.p,q,r) обозначающих что преподаватель n по предмету p в классе q имеет всего r уроков.

Методом резолюции можно пользоваться на различных этапах оформления распоряжения:

1) оформление множества условий,

2) предъявление существования требуемого распоряжения,

3) непосредственное порождение требуемого распоряжения.

Подробнее задержимся на 3). Дедуктивный подход обоснуем на следующих положениях.

Различаются и с х о д н о е множество и множество д а н н ы х. Исходное множество содержит общие аксиомы и условия, а множество данных состоит из элементов вида D(n,p,q,i,d,t) которые оформляются исходя из данного P(n,p,q,r).

О с н о в н о е множество состоит из элементов исходного множества ; и из тех элементов множества данных, которые при фиксированных значениях d,t не противоречат элементам существующего основного множества. В начале, основное множество совпадает с исходным множеством.

На каждом шаге алгоритма, для предиката D(n,p,q,i,d,t) котором все аргумменты фиксированы, методом резолюции исследуется противоречит ли он элементам основного множества. На практике оказывается что число возможных резольвент можно сделать конечным и поэтому процедура разрешима.

Если противоречие найдено, то выбирается другой элемент множества данных, в противном данное D(n,p,q,i,d,t) включается в основное множество, а значения p,q записываются в ячейку строки n и столбца (d,t) матрицы S. Если непротиворечивых данных нет, то ячейка заполняется элементом e, т.е. остается пустой.

Пример.

Приведенные аксиомы определяют следующее множество дизъюнктов:

$$1.1.\ p_1=p_2 \vee \neg D(n,p_1,q_1,i_1,d,t) \vee \neg D(n,p_2,q_2,i_2,d,t)$$

$$1.2.\ q_1=q_2 \vee \neg D(n,p_1,q_1,i_1,d,t) \vee \neg D(n,p_2,q_2,i_2,d,t)$$

$$1.3.\ n_1=n_2 \vee \neg D(n_1,p_1,q,i_1,d,t) \vee \neg D(n_2,p_2,q,i_2,d,t)$$

$$2.1.\ t_1=t_2 \vee \neg D(n,p,q,d,i_1,t_1) \vee \neg D(n,p,q,i_2,d,t_2)$$

Данное $D(N_1,M,I_1,1,1,1)$ порождает следующие резольвенты:

$P_1=M \vee \neg D(N_1,p_1,q_1,i_1,1,1)$ с 1.1. по третьей литере

$M=p_2 \vee \neg D(N_1,p_2,q_2,i_2,1,1)$ с 1.1. по второй литере,

которые дальше с $D(N_1,M,I_1,1,1,1)$ порождают резольвенту M=M.

Других резольвент с 1.1. нет, поэтому данное не противоречит аксиоме 1.1. Аналогично, данное $D(N_1,M,I_1,1,1,1)$ не противоречит остальным аксиомам, так как получаются резольвенты: $I_1=I_1$; $N_1=N_1$; 1=1, респективно. Поэтому данное $D(N_1,M,I_1,1,1,1)$ включается в основное множество.

Дальше, данное $D(N_2,S,I_1,1,1,1)$ с 1.1. порождает S=S, с 1.2. порождает $I_1=I_1$, но с 1.3. сначала порождает резольвенту $n_1=N_2 \vee \neg D(n_1,p_1,I_1,i_1,1,1)$ которая дальше с $D(N_1,M,I_1,1,1,1)$ порождает противоречие $N_1=N_2$. Поэтому, $D(N_2,S,I_1,1,1,1)$ нельзя включить в основное множество.

Оформление распоряжения закончено успехом, если для каждого данного определенно (d,t) так что нет противоречия с основным множеством.

Заметим что такая процедура не обеспечивает обязательно оптимальный вариант распоряжения. Более того, от очередного выбора данных зависит компактность распоряжения, даже и его существование.

Очередь данных возможно определить по разному. Опишем одну из возможных процедур.

## 2. ПРОЦЕДУРА ЗАПОЛНЕНИЯ ПО СТОЛБЦАМ

Процедура использует следующие модули:

ВХОДНОЙ МОДУЛЬ (В): последовательность аксиом $\mathcal{A}$;
последовательность данных $\mathcal{P}$ упорядоченных по преподавателям, так что очередно следуют все уроки одного преподавателя за другим;
параметры: $t_{max}$ и $d_{max}$; матрица S (пустая).

МОДУЛЬ ФИКСИРОВАНИЯ (Ф): осуществляет фиксирование параметров d,t в последовательности данных.

МОДУЛЬ ПРЕПОДАВАТЕЛЯ (П): находит следующее данное в последовательности одного преподавателя.

МОДУЛЬ РЕЗОЛЮЦИИ (R): порождает множество резольвент выбраного данного с основным множеством.
($R_0$): очищает ненужные резольвенты.

МОДУЛЬ МАРКИРОВАНИЯ (М): состоит из следующих подмодулей
$M_1$ - маркирует данное в последовательности данных и

включает маркированое данное в основное множество.

$M_2$ – снимает маркирование с элементов основного множества и удаляет маркированные данные из последовательности данных.

$M_3$ – удаляет маркированные элементы из основного множества и снимает маркирование с элементов последовательности данных.

МОДУЛЬ ПЕРЕХОДА (Н): переход на данные следующего преподавателя.

КОНТРОЛЬНЫЙ МОДУЛЬ (К): проверяет существует ли у первого преподавателя $N_1$ данное которое не противоречит основному множеству.

МОДУЛЬ ЗАПИСИ (З): заполняет столбец ячеек матрицы.

ВЫХОДНОЙ МОДУЛЬ (S): печатает вариант распоряжения.

Схема алгоритма приведенна на Рис. 1.

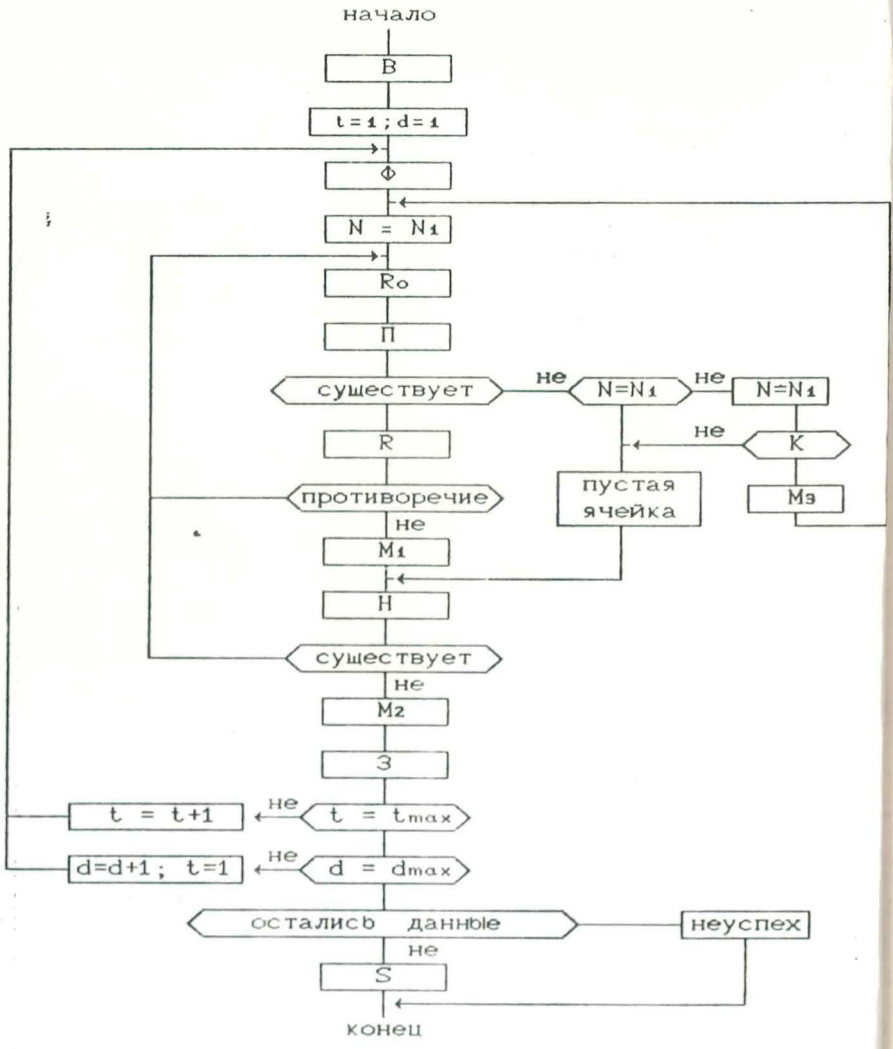



Рис. 1.

Маркирование данных обусловлено тем что возможно в строке преподавателя нет данных которые не противоречат основному множеству. В таком случае соответственная ячейка осталась бы пустой, хотя возможно могла бы быть заполнена, если бы были выбраны другие данные для этого столбца. Поэтому анулируется заполнение ячеек столбца, но предварительно проверяется существует ли у первого по очереди преподавателя данное которое не противоречит основному множеству. Если такое данное не существует, то ячейка остается пустой и продолжается заполнение столбца с сохранением уже порожденных элементов. Если данное, не противоречащее основному множеству, существует у первого по очереди преподавателя, то столбец заполняется заново.

Только когда заполнение столбца вполне закончено, снимаются все маркирования в основном множестве и все маркированные данные удаляются из последовательности данных. Фактически, осуществляется перенос "удачных" данных из последовательности данных в основное множество. Затем переходится к заполнению следующего столбца.

## 3. ПРИМЕР ДЕДУКТИВНОГО ПОРОЖДЕНИЯ РАСПОРЯЖЕНИЯ

Проилюстрируем описанную процедуру на одном совсем простом примере. Пусть $\mathcal{A}(\mathcal{F})$ содержит только аксиомы 1.1, 1.2, 1.3, 2.1 в форме дизъюнктов.

Общие данные: $d_{max} = 2$ ; $t_{max} = 4$ ; $n_{max} = 3$.

Конкретные данные: $P(N_1,M,I_1,2)$, $P(N_1,M,I_2,2)$, $P(N_1,R,II_1,2)$,
$P(N_2,S,I_1,1)$, $P(N_2,S,I_2,1)$, $P(N_2,S,II_1,2)$,
$P(N_3,F,I_1,2)$, $P(N_3,F,I_2,2)$, $P(N_3,f,II_1,2)$.

Приведенная процедура порождает распоряжение представленое на Рис. 2.

| преподаватель | 1. день | | | | 2. день | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| $N_1$ | $I_1^m$ | $II_1^R$ | $I_2^M$ | | $I_1^m$ | $II_1^R$ | $I_2^M$ | |
| $N_2$ | $I_2^S$ | $I_1^S$ | $II_1^S$ | | $II_1^S$ | | | |
| $N_3$ | $II_1^F$ | $I_2^F$ | $I_1^F$ | | $I_2^F$ | $I_1^F$ | $II_1^F$ | |

Рис. 2.

Заметим что класс $I_2$ второго дня не имеет второго урока. Это можно поправить простой перестановкой первого и второго столбца, причем "пустой урок" выдвигается на маргину.

## 4. ЗАКЛЮЧЕНИЕ

Мы указали три исходные возможности для использования метода резолюции в процессе креирования комбинаторных расположений. Каждая из их подлежит дальнейшему исследованию и совершенствованию. В итоге можно ожидать создание системы для оформления комбинаторных расположений на положениях искусственного интеллекта.

## ЛИТЕРАТУРА

1. Цветкович Д, Хотомски П. и др., Десять лет развития и использования экспертной системы "Graph", (сербский), Сборник симпозия "Осуществления и приложения искусственного интеллекта" Дубровник 1989, стр. 25-46.
2. Robinson J.A, A machine oriented logic based on resolution principle, J. ACM. Vol. 12, 1965, pp. 23-41.
3. Хотомски П, Певац И, Математические и программные проблемы искусственного интеллекта в области автоматического доказательства теорем, (сербский), Научна книга, Београд 1991.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 43 - 50

# TOWARDS SYSTEMATIZATION OF INFORMATION (RE)PRESENTATION SOFTWARE

Mirjana Ivanović, Đura Paunić
University of Novi Sad, Faculty of Natural Sciences and Mathematics,
Institute of Mathematics, Trg D. Obradovića 4
21000 Novi Sad

e-mail: {ivanovic,paunicdj}%unsim@yubgef51.bitnet

ABSTRACT: The paper describes the result of an analysis of different systems for representation and presentation of information of the types HyperText, Authoring Systems, Intelligent Tutoring Systems and Intelligent User Interfaces. Some of their important advantages and disadvantages as well as their characteristics are shown. Based on this description, a main characteristics of the whole software class is drawn out and can be used for further systematization.

KEY-WORDS: HyperText, Interface, Multimedia, Presentation

## 1. INTRODUCTION

The development and wide application of different systems for representation and presentation of information and knowledge offers an opportunity to recognize their advantages and disadvantages, thus influencing their further enlargements and improvements together with the development of new systems.

Different profiles of the users of these systems require the presentation of information to be clear and to include different media realizations.

The development of methods and techniques in different fields of computer science influences the emergence of integrated systems which include the representation and presentation of information and knowledge. However, one of major problems which needs to be solved is whether to choose highly specialized data structures for these systems and then adopt them as much as possible to a particular field, or to aim for a general and widely applicable structure.

Most research today is undertaken with a view to developing systems which would possess the following components:

- **Universal formalism** for the representation of different existing forms of information and knowledge (text, picture, sound, ...).

- **Universal method of presentation** of accessible information and knowledge. Such methods should single out, from the existing formalized representation, a meaningful unit of information and present it in an adequate way, depending on the user profile.

- **General applicability** - the possibility of application in different fields.

Research in this field resulted in ready-made software systems with different concepts of representation and presentation, but they can all be generally described as systems characterized by the following elements - units (Figure 1.):

1. **Basic information and knowledge** - a collection of information and knowledge to be represented. It needs to be classified and divided into smaller semantic units.

2. **Additional information** - a collection of information and previous knowledge which can help the user of the system to understand basic concepts more easily. Generally speaking, this kind

of information is not directly related to basic information and since it is not necessary for all users of the system, it is stored separately.

3. **Formalism of representation** is a formalism by means of which basic and additional information are represented in an appropriate way in a computer, forming a data base.

4. **Mechanisms of presentation** are mechanisms which, on the basis of represented knowledge and information, perform an adequate presentation.

5. **Feedback mechanism** is a mechanism which should receive feedback information from the user, turn them into adequate actions and then forward to the system. The system of forwarded information processes and determines the behavior and further course of presentation.

The existing systems and environments are usually applied in the following fields:

- The processes of learning and teaching, i.e. educational processes at all levels.

- Presentations in the strict sense of the term, i.e. presentations of software products, projects, ideas, etc.

- Intelligent user interfaces should enable a sophisticated and natural way of man-machine communication.

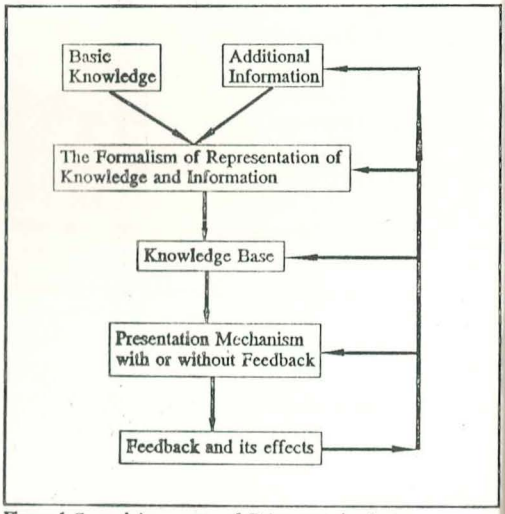



Figure 1 General Appearance of (Re)presentation System

The complexity of information and knowledge which should be represented and presented requires different methods of their media presentations: through text, picture, simulation, sound etc. The inclusion of different aspects of representation and presentation of information and knowledge characterizes these systems as multimedia systems. Methods of artificial intelligence should play an important part in these systems. Since artificial intelligence cannot satisfy high criteria which are set in most systems for representation and presentation of information and knowledge, it is undeveloped. However, most research strives to make use of different artificial intelligence methods in order to obtain systems for representation and presentation of information which are of high quality and easy to use.

## 2. MULTIMEDIA SYSTEMS

Multimedia systems integrate textual information with sound, video, animation and graphics, in order to obtain high-quality computer-aided presentation and representation which will be all inclusive and clear.

When we speak of multimedia features of a software product, we mean the software product which integrates designing simple animation, moving of graphics on the screen, recording and reproducing digitalized speech and music, memorizing of the whole section of work in order to be used subsequently for different purposes. If a software product is to support these different possibilities of expressing information, it has to support different peripherals: monitors, videos, CD players etc.

From the point of view of using peripherals, there are two ways of presenting information:

- **Multiperipheral (dispersed) presentation** - the process of presentation includes and uses several different peripherals.

- **Monoperipheral presentation** - the process of presentation includes only a computer monitor on which various types of information can be shown simultaneously.

From the point of view of coordination of information, there are two types of presentations:

- **Coordination of media** - integral information are obtained as a union of non-redundant individual forms of information. This type of presentation is difficult to carry out, because it requires the system to make intelligent decisions concerning the division of information and mode of presenting particular elements.

- **Non-coordination of media** - some information can be multiplied, i.e. expressed simultaneously through different already existing forms. This type of presentation is easier to carry out because it doesn't require a deep and intelligent analysis and a precise classification of information according to a particular media.

## 3. CLASSIFICATION OF SYSTEMS FOR INFORMATION (RE)PRESENTATION

Systems for representing and presenting information and knowledge can be classified according to several criteria.

1. According to the criterion of **applicability**, there are:

   a) specialized systems - intended for highly specialized fields,
   b) general purpose systems applicable in various fields.

2. According to the criterion of **intelligence**, there are:

   a) intelligent systems - which make use of different methods of artificial intelligence,
   b) non-intelligent systems which do not use methods of artificial intelligence.

3. According to the criterion of **interaction**, there are:

   a) systems with feedback which expect feedback response from the user,
   b) systems without feedback which expect no feedback from the user.

As a rule, the more specialized the system, the greater the possibility of employing methods of artificial intelligence, and vice versa, if a system is of general nature, it is more difficult to make it intelligent. The ultimate aim of most research are highly intelligent systems of general nature.

The existing systems for representation and presentation information can be classified into several basic groups. The most interesting systems for wide use are HyperText systems, Authoring systems, Intelligent tutoring systems and Intelligent user interfaces.

### 3.1. HYPERTEXT

In a strict sense, HyperText can be considered a data structure, i.e. a formalism for representing information which includes a collection of tools for handling structures and presenting the represented information.

In a broader sense, HyperText is a software tool for collecting, storing, searching and presenting information with references. HyperText simulates the ability of the brain to store and search for information with references quickly and intuitively.

Basically, it is a data base management system which makes possible the linking of information on the screen and in the base by the use of references.

HyperText systems find their application in learning processes, presentations in the strict sense of the term, and user interfaces.

According to the classification from chapter 3, they can be classified as non-intelligent general purpose systems without feedback.

### 3.1.1. Characteristics of HyperText

The following elements form the essence of HyperText:

a) **Database** and a method of recording and accessing information in it.

b) **Scheme of the links among elements** (nodes) in the database, i.e. a semantic network. At least one link towards other nodes in the base has its source in each node of the base (Figure 2).

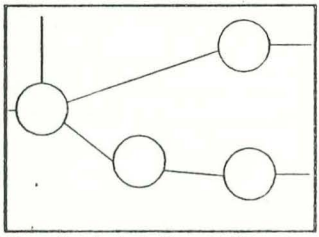


Figure 2 A Part of a Semantic Network

c) **Interface** which enables a visual presentation of information from the elements of the base as well as a simple method of moving from one element to other elements linked with it.

There are several ways of linking elements - nodes of the database. The most frequently used ones are: linear (Figure 3) and hierarchic (Figure 4).

HyperText database usually consists of nodes, the working space of which is the size of a screen. Nodes can be loaded with textual, graphic, audio and video information. In a considerable degree, HyperText functions serve the purpose of multimedia presentation.

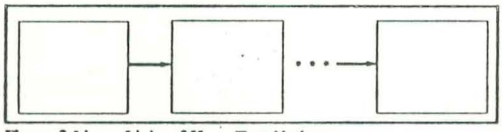


Figure 3 Linear Links of HyperText Nodes

A typical HyperText includes the following auxiliary tools: text editor, graphic editor, tool for three-dimensional representation of pictures, mouses,windows, icons, and pull-down menus. In addition it has a number of index possibilities: inverted word files, hierarchic indexes. In HyperText it is possible to connect the system with an external executing program.

Recently there is a noticeable tendency to combine the techniques of HyperText with artificial intelligence techniques in the implementation of intelligent systems. Apart from basic elements of HyperText, such systems have some new elements:

d) **Scheme of gathering knowledge** which enables automatic addition of information and relations among them into the existing structure of HyperText.

e) Observing the activity in the system and making suggestions about the changing of information in the nodes and links among them.

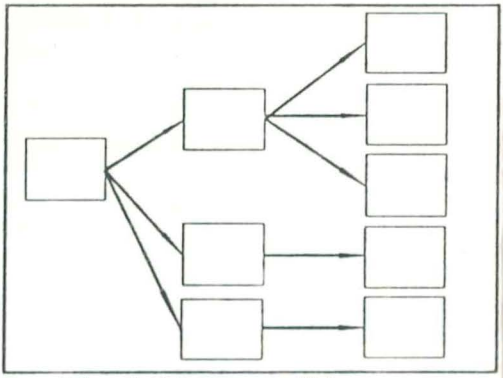


Figure 4 Hierarchic Links of HyperText Nodes

### 3.1.2. Utilization of HyperText

When HyperText is used to create some (usually) multimedia product it is necessary to divide available information, which is to be represented, into simple semantic units with a simple concept or idea which can be represented on a screen.

Individual nodes are connected into compositions - a network which is supposed to make possible quick movement from one to another. Nodes can be:

- **Qualified (named)** - with an associative name which reflects the content of the node.

- **Unqualified (unnamed)** - nodes are unnamed parts of the network. Usually in HyperText it is possible to assign names both to nodes and links.

When using HyperText system, a user can forget, if the database is large, how and why he has reached the point in which he is located at the moment. If this is the case, in many systems it is possible to give a graphic representation - a structural diagram of a node network. Such a diagram can enable the shift into any of the network nodes.

Some HyperTexts allow the assigning of a default course through the database which guides and directs the user through an ordered list of nodes.

Depending on users and fields of application there are considerable variations among HyperText systems. Basically, there are four types:

- systems for problem solving (practicing),
- systems for on-line searching,
- library systems,
- multipurpose (general) systems.

HyperText systems are not basically intelligent, and therefore in the process of their application, a man is the main supervisor of the whole process. He decides which information are included in the database, how to organize and form the nodes and which links are formed among them.

### 3.1.3. Problems with HyperText

HyperText represents a new technology with a number of problems still unsolved, and without established standards.

1. The main problem is how create a convenient models which is relatively easy to manage. Large amounts of information which are to be stored can result in a system which is too large and bulky and the movement through which is slow and difficult.

2. HyperText user is self-guided and he can move through nodes of the network according to his own needs. However, there is a significant danger of choosing a wrong course and getting lost in the network. Such a system with no real determinism in choosing a course through nodes and without actual feedback is too open and often non-effective. Sometimes it is difficult to split up the existing collection of information into nodes and later the classification may turn out to be wrong and in need of correcting. In such cases, most HyperText systems have problems.

## 3.2. AUTHORING SYSTEMS

Authoring systems are software tools which link into a functional unity a computer and

different peripherals, such as CD ROM-s, laser videodiscs etc. These systems are usually called CAI and they find their application in the field of teaching and presentations in the strict sense of the term. They can be divided according to their purpose, and in most cases they are specialized in particular fields. According to the classification from Chapter 3, they can be grouped as non-intelligent systems with feedback.

A numerous group of people, such as teachers, managers, authors of different teaching processes uses authoring systems to create their own interactive multimedia programs (systems).

The first authoring systems were independently developed software products with no ability to control external devices. However, the current trend is to develop such authoring systems which would be able to control and synchronize a large number of external devices, they are becoming more powerful and more extensively used in the field of interactive multimedia systems.

Knowledge and information which are to be represented in a system are divided into smaller sections - topics, which are linked to make a semantic unit. Topics and links among them form a graph. In the process of presenting information stored in a graph, there is a movement along different parts and nodes in the graph of the system.

### 3.2.1. Features of authoring system

By analyzing different authoring systems available in the market certain features can be determined and criteria set which a system should meet in the process of its creation as well as in its operation.

- Authoring systems, using different convenient tools enable the user to design and create his own multimedia application without using and being familiar with a procedural language. However, the existence of a programming language as an option can be very useful, so that an authoring system can have its own procedural language or offer a direct access to some such language.
- An authoring system should enable simple integration of outputs from some other programs i.e. it should make possible the use of textual data files already existing, as well as including graphics created by some independent graphical applications outside the system.
- An authoring system must offer an appropriate method to create text and graphics by
  - having its own text (graph) editor and/or
  - supporting some already existing text (graph) editors.
- It should make possible the control of the process of presentation and moving from one part of the system into another.
- It should enable simple "switching" of multimedia aspects.
- It should have simple singular interface, which will enable unified use of all tools available on a computer. Each time a new multimedia element is created (or a new device added) the existing interface should support it.
- It should allow full integration of text, graphics, animation, sound, video. The access to these elements should be built into a standard interface. The system should comprise two basic functions:
  - branching moving to a designated topic, from where it is possible to return to previous state or to move along some other path in the graph system,
  - activation of some other programs and automatic return from the activated program into the authoring system.
- An authoring system should offer context sensitive help system.

### 3.2.2. Application of authoring systems in learning processes

Authoring systems can be used in teaching processes preparing lessons and courses etc. In such cases a system should have certain mechanisms of testing the acquired knowledge. Authoring system in educational processes should advance the possibility of handling questions/ answers of the following type:

1. true-false,
2. multiple-choice questions,
3. open questions.

In order to obtain information about the quality of teaching and the results of the evaluation of acquired knowledge within an authoring system, it is suitable to create separate units for each participant in the process. Such units would store the following types of information:

- the length of the teaching process,
- the number of correct answers,
- the number of individual topics (i.e. nodes in the graph) covered.

It would be preferable for this data to be recorded in the format which could be forwarded to specialized programs for obtaining graphic presentations and proving statistical data about the effectiveness of teaching.

Authoring systems support considerable number of tools necessary to create sophisticated and powerful interactive multimedia applications, but their quality depends on the creativeness and imagination of their author.

Mechanisms of an authoring system for testing the acquired knowledge by means of feedback allow clearer and easier movement through system graph. Paths through the system are determined by students' answers to questions and tasks and there is no possibility of students' getting lost in the graph and knowing how to continue.

The existence of a feedback which determines the progress of movement through the graph is a considerable improvement in these systems in comparison with HyperText systems.

### 3.3. INTELLIGENT TUTORING SYSTEMS

Intelligent tutoring systems are exclusively intended for learning processes. They represent a combination of different fields: education, psychology, artificial intelligence, cognitive sciences etc. These are usually highly specialized systems which is the reason why they successfully employ the methods of artificial intelligence. Since they are basically intended for learning processes, ITS systems are systems with feedback. Their architecture and structure varies, but usually four types can be recognized:

1. **Expert (teaching) module** is a data base which stores information from the field in which ITS is used

2. **Students' module** - is intended for modelling individual student's knowledge in the field where ITS is used. The content of the module constantly changes during the process of learning.

3. **Tutoring module** specifies the method of presenting the material in the field in which ITS is used, the manner and rhythm of presentation.

4. **Diagnostic module** - which constantly modifies the students module in accordance with answers that students give to the questions asked. This module is also linked to the expert module.

Efforts that are made concerning further development of these systems have as their aim the communication between students and computer in natural language. However, taking into account current state of affairs in research in natural language interfaces, nowadays the communication is performed either in meta-language which is close to a natural language or in some subset of a natural language.

The communication is established by translating a natural language into a computer code at the input of the system, while at the output machine cone is translated into a 'natural' language. The communication is mainly textual or has some rudimentary graphic capabilities.

## 3.4. INTELLIGENT MULTIMEDIA INTERFACES

An intelligent interface should accept formally represented information from an application program and present them independently in the form of graphic, animation, sound or a natural language, and if possible, through coordinated media. In addition it should allow accepting information from the user, in natural language, by demonstration, by choosing from the menu etc., and transform them into a formalism and a code which is understandable for the application.

If an intelligent interface is to realize such communication, it should have additional information about:

1. application,
2. the user and his previous knowledge of application,
3. the aim which is supposed to be achieved by the presentation.

These requirements are not simple. In favor of this, there the fact that today there are only a few projects which come close to these goals but have not reached them yet. Unfortunately, these projects are highly specialized and applicable to a very small group of problems. Formally, general structure of an intelligent interface can be determined as follows (Figure 5):

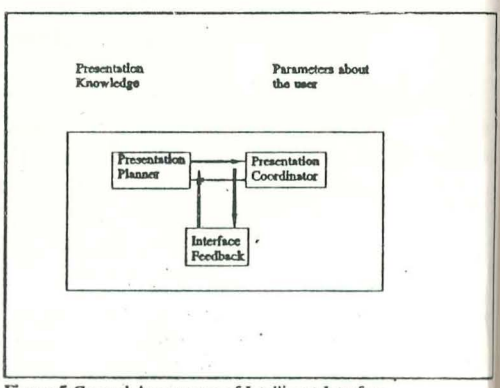



Figure 5 General Appearance of Intelligent Interface

- **Presentation planner** - takes formalized knowledge and (if possible, coordinativelly) generates text, picture, animation, sound etc. The presentation formed in this way later sent to presentation coordinator for processing.

- **Presentation coordinator** - it integrates in the meaningful way in the memory elements obtained by presentation in the media which are available. Failing that, it gives back these elements to the planner requesting a correction. When presentation planner and coordinator are coordinated the presentation is sent for further processing to the feedback.

- **Presentation feedback** simulates the man. Its task is to see whether a set goal is achieved by given presentation. Unless the comprehensibility and mode of presentation is satisfying the feedback returns all given elements to presentation coordinator asking for corrections. The moment feedback is satisfied with the presentation it forwards it to the ultimate man. Since the whole process should occur in real time, the communication between individual elements of the interface should be reduced to reasonable duration.

### References

1. **Carando, P.**, *SHADOW - Fusing HyperText with AI*, IEEE Expert, Winter 1989, pp.65-78.
2. **Frisse, M.**, *From Text to HyperText*, Byte, October 1988., pp.237-244.
3. **Ivanović, M., Paunić, Đ., Budimac, Z.**, *A Tool for general CAI Lesson Creation*, Proc. of XV International Summer School "Programming 91" (Sofia, Bulgaria), 1991, 107-110.
4. **Ivanović, M.**, *A Contribution to Development of Programming Languages using Object-Oriented Methodology*, Ph.D. Thesis, University of Novi Sad, 1992 (in Serbian).

CEEDINGS OF THE VI CONFERENCE ON
IC AND COMPUTER SCIENCE LIRA 92
SAD, OCTOBER 29-31, 1992., PP. 51 - 59

# DYNAMIC DETERMINATION OF WEIGHT COEFFICIENTS IN THE IMPLEMENTATION OF FFT ON FINITE ABELIAN GROUPS

Janković Dragan

Elektronski Fakultet
Beogradska 14, Niš

**Abstract**

In this paper we consider the implementation of fast Fourier transform (FFT) on finite Abelian groups. The rules for generation of FFT flow-graphs are given. A method for the dynamic determination of weight coefficients in the flow-graph of FFT is presented.

Key words: Fourier transform, fast Fourier transform, finite Abelian groups, weighting coefficients.

## 1 INTRODUCTION

The main problem in the application of discrete transforms is the efficiency of their calculation. The use of defining expressions is, in general case, not efficient, which caused the fast algorithms to be developed. The obvious example is given in table 1 [1]. Calculation times $T_d$ for discrete Fourier transform (DFT) obtained by the application of defining expressions are compared to calculation times for FFT ($T_f$) for the same digital image processing task. $M$ is the number of points in the image.

| M | $T_d$ | $T_f$ |
|---|---|---|
| 64×64 | 8 minutes | 3 seconds |
| 256×256 | 30 hours | 1 minute |
| 512×512 | 20 days | 5 minutes |
| 1024×1024 | 1 year | 20 minutes |

Table 1: Calculation times for DFT

In this paper we consider the problem of an efficient implementation of Fourier transform on finite Abelian groups. In practical realizations of FFT one of basical problems is the determination of weight coefficients. An approach usualy applied in practical realizations of FFT consists of an a priori determination of all weighting coefficients in flow-graph of FFT on a given group, which are stored for later use. This approach implies a strong limitation on the choice of the group on which FFT could be implemented. Actually, it is efficient in situations where repeated calculations of FFT on a given group are required. Another disadvantage of this approach is relatively high memory requirements, or specific hardware structure (for real-time).

The method proposed in this paper tends to overcome these disadvantages. The method is based upon the caracteristic structure of Fourier transformation matrix on a given group and consists of the determination of the weight coefficients needed for calculation of FFT on the corresponding subgroups. In this way the problem of determination of the set of weighting coefficients on a group is reduced to an a priori determination of some subsets of representative weighting coefficients. The other weighting coefficients are determinated during the implementation of FFT algorithm by application of rules derived in this paper.

In such a manner memory requirements are greatly reduced and calculation of FFT on groups of arbitrary orders is provided.

## 2 NOTATION AND DEFINITIONS

Let $G$ be a finite Abelian group of order $N$. Let us suppose that $G$ is representable a a direct product of cyclic subgroups $G_i$ of orders $g_i$, $i = 1, ..., k$, respectively, i.e.

$$G = \mathbf{X}_{i=1}^{k} G_i\,, \quad N = \prod_{i=1}^{k} g_i\,, \quad g_1 \leq g_2 \leq \ldots \leq g_k.$$

Denote by $C(G)$ the space of all complex functions on $G$. Recall that the characters o $G$ are defined as the homomorphisms of $G$ into the unit cicle, i.e. they are given by:

$$\begin{aligned} \chi(w,x) &= \chi((w_1, \ldots w_k), (x_1, \ldots x_k)) \\ &= \exp 2\pi i \sum_{i=1}^{k} \frac{w_i x_i}{g_i}, \end{aligned}$$

where :

$$\begin{aligned} x &= \bigcirc_{i=0}^{N-1} x_i e_i, \\ w &= \bigcirc_{i=0}^{N-1} w_i e_i, \quad e_i \text{ - identity of } G_i. \end{aligned}$$

Using the characters of group $G$, the direct and inverse Fourier transforms on C(G are defined respectively by:

$$S_f(w) = N^{-1} \sum_{x=0}^{N-1} f(x)\chi^*(w,x)\,,$$

$$f(x) = \sum_{w=0}^{N-1} S_f(w)\chi(w,x) \ ,$$

or in matrix form:

$$[\hat{f}] = N^{-1}[\chi^*][f] \ ,$$
$$[f] = [\chi][\hat{f}],$$

where $\chi^*$ is the complex conjugate of $\chi$ [2].

Fourier transformation matrix can be represented as the Kronecker product of Fourier transformation matrices on the coresponding subgroups $(G_i)$, i.e.

$$[\chi^*] = \bigotimes_{i=1}^{k} [\chi_i^*] \ ,$$

where is $[\chi_i^*]$- character matrix of subgroup $G_i$ [3].

The flow-graph of FFT on a given group $G$ is derived from the factorization of Fourier transformation matrix into the product of sparse matrices. The factorization of $[\chi^*]$ is not uniquely defined, and therefore, different FFT algorithms exist in the literature, see [4,5].

In this paper we will use Good-Thomas factorization for the derivation of FFT on finite Abelian groups. In this approach the factorization method is given as:

$$[\chi^*] = \left[C^{(1)}\right] \cdot \left[C^{(2)}\right] \cdot \ldots \cdot \left[C^{(k)}\right] \ ,$$

where

$$\begin{aligned}
\left[C^{(1)}\right] &= [\chi_1^*] \otimes I_{g_2} \otimes \cdots \otimes I_{g_k}, \\
\left[C^{(2)}\right] &= I_{g_1} \otimes [\chi_2^*] \otimes \cdots \otimes I_{g_k}, \\
&\vdots \quad\quad \vdots \quad\quad\quad\quad \vdots \\
\left[C^{(k)}\right] &= I_{g_1} \otimes I_{g_2} \otimes \cdots \otimes [\chi_{g_k}^*],
\end{aligned}$$

with $I_{g_i}$ is the identity matrix of order $g_i$.

EXAMPLE:

A given finite Abelian group $G$ of order 6 can be conveniently identified with the direct product of two cyclic groups $G_1 = (\{0,1\},\oplus)$ and $G_2 = (\{0,1,2\},\odot)$ where $\oplus$ and $\odot$ are the addition modulo 2 and 3, respectively. In this case the Fourier transformation matrix on $G$ is given by:

$$[\chi_G^*] = [\chi_2^*] \otimes [\chi_3^*] = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 \\ 1 & e_2 & e_1 \\ 1 & e_1 & e_2 \end{bmatrix} \ ,$$

where is $e_1 = \exp \frac{i2\pi}{3}$, $e_2 = \exp 2\frac{i2\pi}{3}$.

Using the Good-Thomas factorization this matrix can be represented as

$$[\chi_G^*] = \left[C^{(1)}\right]\left[C^{(2)}\right] ,$$

where

$$\left[C^{(1)}\right] = [\chi_2^*] \otimes I_3 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix} ,$$

$$\left[C^{(2)}\right] = I_2 \otimes [\chi_3^*] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 \\ 1 & e_1 & e_2 \\ 1 & e_2 & e_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & e_1 & e_2 & 0 & 0 & 0 \\ 1 & e_2 & e_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & e_1 & e_2 \\ 0 & 0 & 0 & 1 & e_2 & e_1 \end{bmatrix} .$$

# 3 IMPLEMENTATION

The flow-graph of FFT on a given group $G$ can be derived from the factorization fr section 2. The procedure is explained on an example. The flow-graph of FFT on gro $G$ from example from section 2, is given on Figure 1.

Note that, each matrix $[C^{(i)}]$ describes uniquely one step of the fast Fourier transfor The weight coefficients in flow-graph are determined by non-zero elements of $C^{(i)}$.

From the analyses of factorizations of Fourier transform and flow-graphs on differe groups the rules for the generation of FFT flow-graphs on an arbitrary group can derived. In this way the generation of FFT on finite Abelians is almost complet formalized and can be given directly without factorization of the transformation matr

In the case that the number of group equals 1 ($k = 1$, i.e., transform is DFT) t rules are:

- The number of steps of the FFT flow-graph $n$, is equal to the number of factors group order ($N = f_1 f_2 \dots f_n$) where $f_i$ is $i$-th factor of $N$.
- The number of inputs as well as outputs of $i$-th step is $N$.
- The output of $m$-th step is input for $(m+1)$-th step.
- In $j$-th step $\frac{N}{\prod_{i=1}^{j} f_i}$ identical nodesets can be distinguished.

Let us divide each set of nodes, in step $j$, into subsets consisting of $\prod_{i=0}^{j-1} f_i$ nodes, $f_0 =$

- Nodes with identical relative positions in the nodesubsets are connected.
- Each node, at $j$-th step, is connected to $f_j$ different nodes.
- Weights coefficients of branches sinking in the first node in nodeset are equal to 1.
- Each branch that starts from a node in the first subsets has weight coefficient equal to 1.
- The first branch sinking in the node has weight 1.
- Weight coefficient of $i$-th branch $(2 \leq i \leq f_j)$ sinking in the node with relative position in nodeset $r_p$, at $j$-th step, is $W_l$
  where $l = (Brgr(j-1)(rp-1)) \ mod \ N$,
  $Brgr$ - the number of identical nodesets at $j$-th step.

Note that input data are in bit-reverse ordering.

In the case that the number of groups is greater than 1 (i.e. Chrestenson transform) the rules are similar to the rules for DFT.

- The number of steps $k$, equals to the number of subgroups $(G = G_1 \circ G_2 \circ \dots G_k)$.
- In $j$-th flow-graph step $\prod_{i=0}^{j-1} g_i$, $g_0 = 1$, identical nodesets can be distinguished, where $g_i$ – order of subgroup $G_i$,
  $N$ – order of group $G$, $N = g_1 g_2 \dots g_k$.
- Each node, at $j$-th step, is connected to $g_j$ different nodes.

Let us divide each set of nodes, in step $j$, into nodesubsets consisting of
$\prod_{i=j+1}^{k+1} g_i$, $g_{k+1} = 1$.

- Nodes with identical relative position in nodesubset are connected.
- The first branch sinking in the node has weight coefficient 1.
- The nodes of the same nodesubset are sinks for branches with identical weight coefficients.
- Weight coefficients of branches, in step $j$, in the nodes of $j$-th nodesubset are equal to the elements of $j$-th column of character matrix of subgroup $G_j$.

The rules derived for DFT, i.e. for the derivation of FFT on the cyclic groups, enable the generation of the same procedure, for any finite Abelian group. Note that besides cyclic groups the derivation of FFT in literature, is mainly restricted to some particular finite Abelian groups most frequently used in some practical applications, for example

the finite dyadic or $p$-adic groups. See, for example, [6,7].

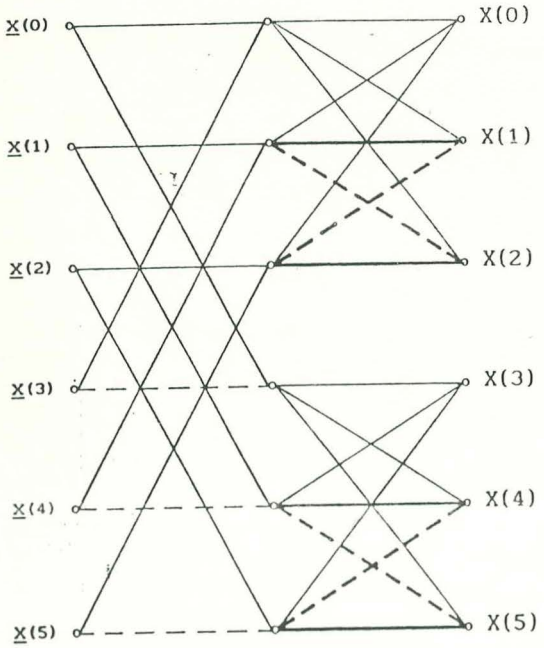



Figure 1. FFT flow-graph for N=6.

By means of derived rules, the implementation of FFT on finite Abelian group of arbitrary order $N$ is straightforward, as well as the generation of the flow-graph. T factorization of Fourier transformation matrix is not needed any more. Derived ru are sufficient for generation of the flow-graph.

It is necessary to calculate and store representative weight coefficients on subgrou $G_i$ (i.e. characters of subgroups $G_i$), and to apply the derived rules. This could noted as an advantage in respect to required memory, compared to majority of existi

practical realizations of FFT on some particular finite groups where the storage of all weighting coefficients is common requirement.

The procedure Fast for calculation of DFT on groups of on arbitrary order N is presented as an illustrative example in the appendix.

The proposed method is implemented in the programming package "Fourier" at the Faculty of Electronics, University of Niš, capable of calculating DFT, Walsh transform for different orderings and Chrestenson transform as some particular examples of Fourier transform on finite Abelian groups [8]. The FFT on groups obtained by the proposed method is further used for calculation of the convolution, correlation, autocorrelation, power and frequency spectrum of functions on finite Abelian groups.

# 4 CONCLUSION

The implementation of Fourier transform on finite Abelian groups is considered. We propose the method for dynamic determination of weight coefficients of FFT flow-graph on arbitrary finite Abelian groups. We believe that the implementation of Fourier transform by means of derived rules, is very convenient for hardware-inferior systems. The method enables the considerable savings of the memory storage usually required for the implementation of FFT on a group.

# References

[1] Bojković, S.Z., *Digital Image Processing*, Naučna knjiga, Beograd, 1989, (in Serbian).

[2] Stanković, R.S., Stojić, M.R., Bogdanović, S.M., *Fourier Signal Prezentation*, Naučna knjiga, Beograd, 1988, (in Serbian).

[3] Stojić, M.R., Stanković, M.S., Stanković, R.S., *Discrete Transformations in Application*, Naučna knjiga , Beograd, 1985,(in Serbian).

[4] Good, I.J., *"The interaction algorithm and practical Fourier analysis"*, J.Royal Statist. Soc.,B20, pp.361-375, 1958.

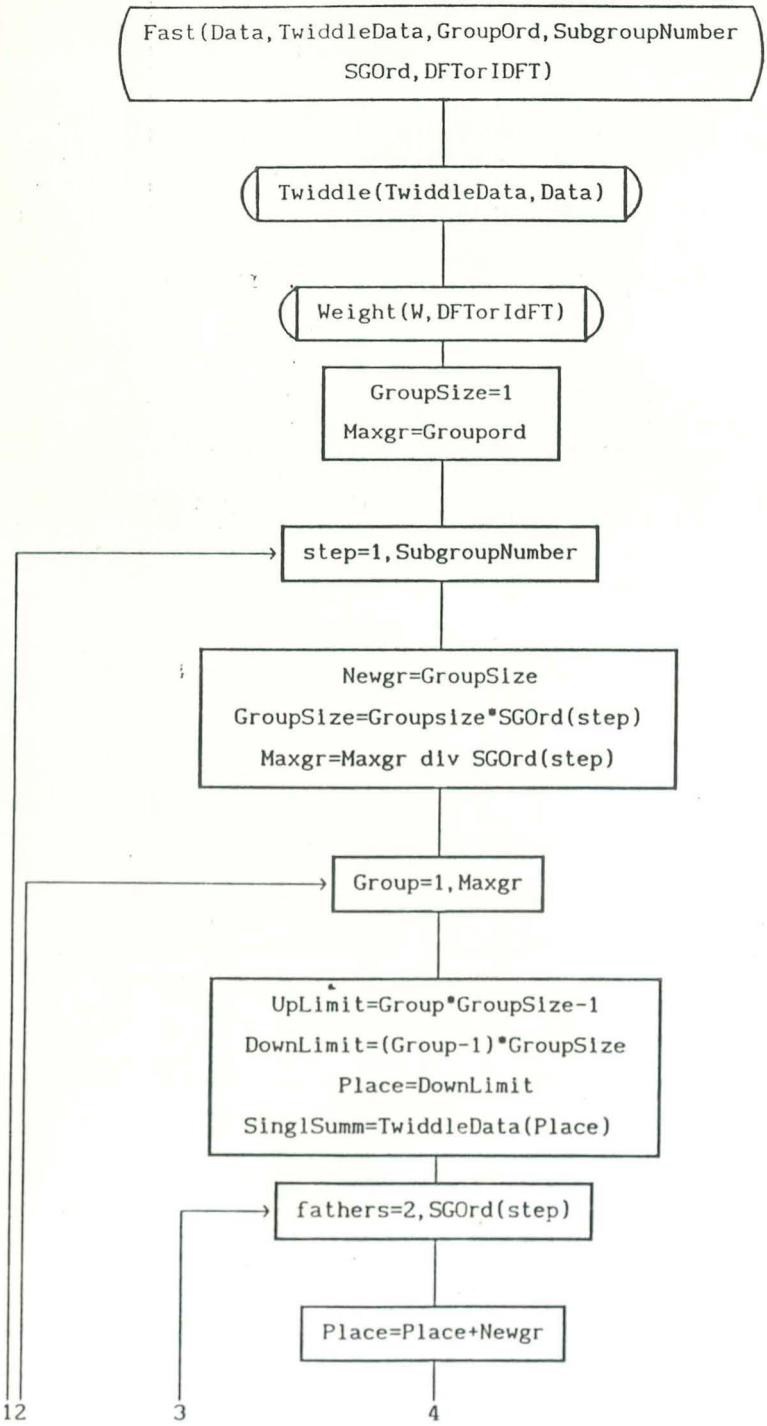
[5] Ahmed N., Rao, K.R., *Orthogonal Transform for Digital Signal Processing*, Springer-Verlag, Berlin, 1975.

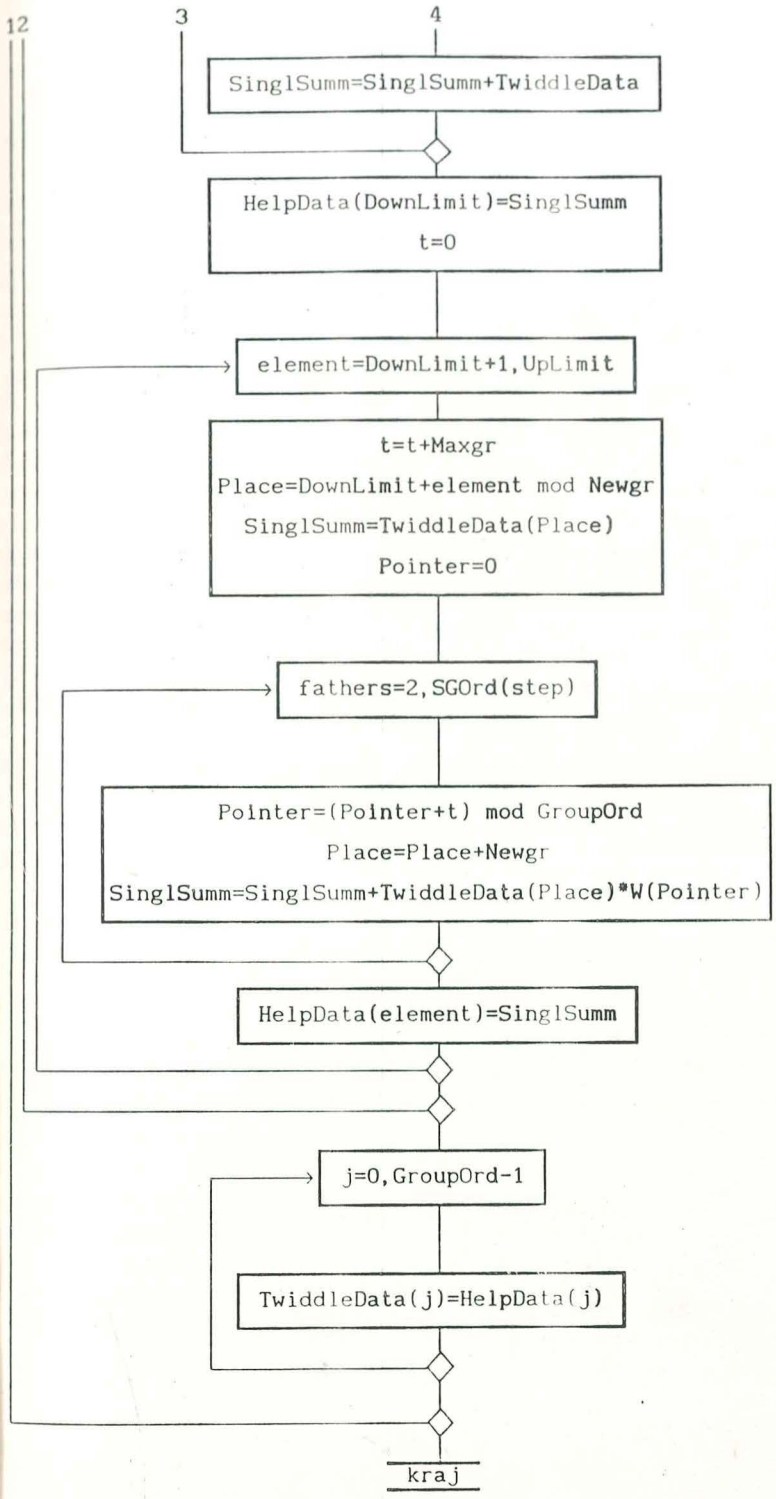
[6] Burus, C.S., Parks, T.W., *DFT-FFT and Convolution Algorithms, Theory and Implementation*, John Wiley, New York, 1985.

[7] Beauchamp, K.G., *Theory and Applications of Walsh and Related Functions*, Academic Press, New York, 1985.

[8] Janković, S.D., *Elements of Fourier Analysis on Finite Abelian Groups*, B.Sc. thesis, Elektronski fakultet, Niš, 1991, (in Serbian).

APPENDIX: The algorithm for procedure 'Fast'.




12
3
4
SinglSumm=SinglSumm+TwiddleData
HelpData(DownLimit)=SinglSumm
t=0
element=DownLimit+1,UpLimit
t=t+Maxgr
Place=DownLimit+element mod Newgr
SinglSumm=TwiddleData(Place)
Pointer=0
fathers=2,SGOrd(step)
Pointer=(Pointer+t) mod GroupOrd
Place=Place+Newgr
SinglSumm=SinglSumm+TwiddleData(Place)*W(Pointer)
HelpData(element)=SinglSumm
j=0,GroupOrd-1
TwiddleData(j)=HelpData(j)
kraj

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 61 - 68

# A MODIFIED DENOTATIONAL APPROACH FOR A SEMANTICS OF PROGRAMMING LANGUAGES

Jerinić Ljubomir[1]
Institute of Mathematics
Trg D. Obradovića 4, 21000 Novi Sad

**KEYWORDS:** Functional Programming, Programming Languages, Operational Semantics, Denotational Semantics

**ABSTRACT**

A formal system of describing a semantics of programming languages, based on the denotational approach is presented. That approach is modified with some operational view for formal description, combined with the object oriented methods of programming.

The method is used for characterizing a semantics of a machine language of an abstract machine, used in the implementation of a purely functional programming language A_LispKit Lisp [Je92a, Je92b]. This machine (called A_SECD machine), is a natural extension of the SECD machine [La64, St83a, St83b], which has been used in the implementation of various versions of functional programming languages [Sto84, St84, He80, Bulv89, Bulv90a, Bulv90b].

## 1. INTRODUCTION

The formal description of the meaning of some valid syntax constructions of any formal language is very interesting field of work for mathematicians. There are a lot of methods for describing a semantics [Me90]: Attribute grammars, Translational semantics, Operational semantics, W-grammars, Axiomatic semantics, Denotational semantics, and so on.

In the formal definition of a language and realization of A_SECD machine [Je92a, Je92b], we used the denotational semantics technique, combined with the operational method and with some kind of object-oriented definitions.

## 2. OPERATIONAL AND DENOTATIONAL SEMANTICS

As we use a function as a main object for describing the semantics (meaning) of some language constructions, it can be considered either as an algorithm which will produce a value given an argument, or as a set of ordered argument-value pairs. The first view is **dynamic**, or **operational**. A semantic function is defined as a sequence of operations in time. The second idea is **static** or **denotational**, in which the function is regarded as a fixed set of associations between the arguments and corresponding values.

As we said, the **operational semantics** approach, is a kind of implementation of an algorithm of meaning of a construction, i.e. that approach is similar to an interpreter. The idea is to express the semantics of a language by giving a mechanism that makes possible to determine the effect of any valid language construction. Such a mechanism is an **interpreting automation**, a formal device

[1]This research was supported by Science Fund of Serbia

capable of formally executing a program in that language, by giving the machine transitions from state to state.

The operational approach presents some definite advantages. It gives a concrete, intuitive description of a programming language, it appeals to the programmers because the descriptions given are so close to real programs. Also, it is fairly easy to devise an interpreter to execute such description on example programs, which makes operational semantics approach attractive as a tool for testing new languages or languages features, long before any compiler has been written. The very qualities of the operational method, however, speak also of its limitations. Striving to be executable, operational descriptions lose one essential quality of specification-independence from implementation.

On the other hand, the denotational semantics approach, may be viewed as a variant of translational semantics. Using this method, we will express the semantics of a programming language by a translation schema that associates a meaning (denotation) with each valid language construction. The difference is in result of translation. In translation semantics, the meaning of a construction is a program, while in denotational semantics it is a mathematical object.

The denotational description of a programming language is given by a set of meaning functions M associated with the construction of its grammar. Each of these functions is of the form: $M_T : T \to D_T$, where T is a language construction. Such functions will consistently have names of the form M (for meaning), subscripted by the name of a construction. The set $D_T$ of denotations may be different for various constructions T, and they are called semantic domains. In contrast, constructions are called syntactic domains.

The denotational method is exclusively focused on the programs. It excludes the state and other data elements, enables to reach a level of abstraction which cannot be obtained in the operational approach, whatever abstract interpreting automata were chosen. More generally, denotational specifications provide an elegant mechanism to define the semantics of the programs in terms of classical mathematical notations such as functions.

## 3. SEMANTIC DEFINITION OF A_SECD MACHINE

A_SECD machine can be defined as a general function **Exec** [He80] which takes a compiled version of a function **Fun**, denoted with **Fun**$^*$, and the S-expression representation of the arguments **Args**. Thus, it produces an S-expression representation of the result of applying **Fun** to **Args**. The formal definition of **A_SECD** machine, the function **Exec**, given in terms of denotational semantics approach, is:

$$\mathbf{Exec} : \mathcal{F}_{\mathbf{A\_SECD}} \times \mathcal{Z} \to \mathcal{Z}, \text{ and}$$
$$\mathbf{Eval}\ [\![\ \mathbf{Exec(Fun^*, Args)}\ ]\!]\ \rho = \mathbf{Eval}_{\mathbf{A\_SECD}}\ [\![\ \mathbf{Fun^*(Args)}\ ]\!]\ \rho = \mathbf{Res},$$

where $\mathbf{Fun}^* \in \mathcal{F}_{\mathbf{A\_SECD}}$, $\mathbf{Args} \in \mathcal{Z}$ and $\mathbf{Res} \in \mathcal{Z}$. The set $\mathcal{F}$ represents a set of all programs-functions of A_LispKit Lisp language, $\mathcal{Z}$ is a set of all S-expressions, and the set $\mathcal{F}_{\mathbf{A\_SECD}}$ is a set of all possible, executable, programs in the machine language of **A_SECD** machine. The general denotational function **Eval** is defined by $\mathbf{Eval} : \mathcal{E} \to \mathcal{C}$, where $\mathcal{E}$ is a set of all expressions, and $\mathcal{C}$ is a set of all values of a language. The denotational function $\mathbf{Eval}_{\mathbf{A\_SECD}}$ describes a semantics of the **A_SECD** machine, and it is given with $\mathbf{Eval}_{\mathbf{A\_SECD}} : \mathcal{F}_{\mathbf{A\_SECD}} \to \mathcal{Z}$, where $\mathcal{F}_{\mathbf{A\_SECD}}$ is a set of all valid functions written in the machine languages of A_SECD machine.

From the point of view of operation semantics, formal definition of A_SECD machine, given by the function Exec is:

$$\mathbf{Exec(Fun^{*}, Args) = Apply(Fun, Args),}$$
$$\mathbf{Compile(Fun) = Fun^{*},}$$

where the function Compile translates a source code of a program function Fun into the machine language of A_SECD machine. That is, in some way the A_SECD machine, given the S-expression representations of the compiled function (a machine language program) and its arguments, executes the machine language program to compute the result of applying that function to these arguments.

The function **Exec(Fun*,Args)** is implemented in such a way that it operates a stack for the evaluation of function calls, much as the process described. Since the program **Fun*** is an S-expression and since the data with which it operates are S-expressions, the natural notation for expressing the state of this stack machine is the S-expression notation. Thus, if we wish to denote the stack in such a way that its top item is an S-expression of **X** we will write **(X.s)**, where **s** represents the remaining items.

Strictly speaking, a pure stack is a data structure which has only two operations, the pushing of a new element onto the stack, and conversely, the operation of popping an element from the stack. It is said to be used in last-in-first-out discipline. The denotational semantics of a stack is:

$$\mathbf{Stack} : \mathfrak{C} \to \mathfrak{C},$$
$$\mathbf{Stack_{Pop}} : \mathfrak{C} \to \mathfrak{C},$$
$$\mathbf{Stack_{Push}} : \mathfrak{C} \times \mathfrak{C} \to \mathfrak{C},$$
$$\mathbf{Eval}\ [\![\ \mathbf{Stack_{Pop}}\ ((X.s))\ ]\!]\ \rho = \mathbf{Eval}\ [\![\ X\ ]\!]\ \rho,$$
$$\mathbf{Eval}\ [\![\ \mathbf{Stack_{Push}}\ (X, (Y.s))\ ]\!]\ \rho = \mathbf{Eval}\ [\![\ (X.(Y.s))\ ]\!]\ \rho,$$

for $(\forall X \in \mathfrak{C})$, $(\forall Y \in \mathfrak{C})$ and $(\forall s \in \mathfrak{C})$.

## 4. THE OPERATIONAL SEMANTICS OF A_SECD MACHINE

The **A_SECD** machine consists of five registers and each of them holds an S-expression. These registers derives their names from the purpose they have in dealing with S-expressions:

**S** the stack, used to hold the intermediate results during computation. At the end of the program execution, the top of the stack S contains the final result,

**E** the environment, holds the values which are bound to variables during evaluation,

**C** the control list, used to hold the machine-language program which is currently executed. In each moment of the evaluation process, the first element of the control list is the command which will be processed next,

**D** the dump, which saves the values of all other registers S, E, and C during a new function call.

the resident library manager, the stack which contains the resident libraries, i.e. the programs in an executable code written in the machine language of A_SECD machine, which are consulted during the evaluation of a program.

The machine language of the A_SECD machine consists of a certain number of commands. The execution of a command forces the machine to change its state, i.e. the contents of its registers. We call this a **machine transition**, and it can be denoted, from the point of operational semantics, in the following way:

S E C D L ⇒ S' E' C' D' L',

where S, E, C, D and L are the contexts of the registers before the next command execution, and S', E', C', D', and L' denote the new contexts of the all registers after that. For example, a machine transition of arithmetic operations of A_SECD machine is:

(a b.S) E (OpA.C) D L ⇒ (b SiA a.S) E C D L,

where OpA ∈ { **ADD, SUB, MUL, DIV** ... } and SiA ∈ { ⊕, ⊖, ⊗, ⊘, ... }. For another example, a machine transition of relation between the data of A_SECD machine is:

(a b.S) E (ReA.C) D L ⇒ (b RiA a.S) E C D L

where ReA ∈ { **GT, GE, LE, NE** ... } and SiA ∈ { >, ≥, ≤, ≠, ... }.

As all registers of A_SECD machine, according to the rules of machine transition, perform some operations on S-expressions, the simulator of A_SECD machine is naturally implemented by mapping these rules in some procedures of the implementation language.

On the other hand, the meaning of the instructions of A_SECD machine, in a mathematical sense is not clear. A much better way to describing the meaning of A_SECD machine, i.e. of its semantics, is to used denotational semantics approach.

## 5. THE MEANING FUNCTIONS OF A_SECD MACHINE SEMANTICS

A_SECD machine, from operational semantics standpoint, has been defined by an **Exec** function. Now, the modified denotational semantics of A_SECD machine will be presented. The meaning function of A_SECD machine in the terms of denotation is $Eval_{A\_SECD}$: $\mathfrak{C} \to \mathfrak{C}$, where $\mathfrak{C}$ is a set of all S-expressions.

Firstly, we must define the semantics of the abstract data type, which is used in the implementation of S-expression, and A_SECD machine operations denoted with **LispCell**. The denotational semantics of some operations which are defined under that data type, is given by a semantic function: $Sem_{LC} : \mathcal{F} \to \mathcal{E}$, where $\mathcal{F}$ is a set of all operations and functions of A_SECD machine and/or operations defined under data type **LispCell**. The set $\mathcal{E}$ is a set of standard types of implementation language like **Real, Boolean, Integer** and so on, together with data type **LispCell**, $\mathcal{E}$ = {**Real, Boolean, Integer, String, LispCell**}.

With $\mathcal{E}_{LC} \subset \mathcal{E}$, we denoted a set of all possible values of the type **LispCell**. Let us define all of the subsets of the set $\mathcal{E}_{LC}$: $\mathcal{E}_{LCreal}$ (the set of real values), $\mathcal{E}_{LCinteger}$ (the set of integer values), $\mathcal{E}_{LCpair}$ (the set of all pairs), $\mathcal{E}_{LClist}$ (the set of all lists), $\mathcal{E}_{LCsymbol}$ (the set of all symbolic atoms), $\mathcal{E}_{LCBoolean}$ ($\mathcal{E}_{LCBoolean}$ = {T, F}), and $\mathcal{E}_{LCnil}$ ($\mathcal{E}_{LCnil}$ = {**NIL**}).

Also, with $\mathcal{E}_{IL} \subset \mathcal{E}$, we denoted the set of all values of types of the implementation language. This set has the following subsets: $\mathcal{E}_{ILreal}$ (the set of all

real values), $\mathcal{E}_{lInteger}$ (the set of all integer values), $\mathcal{E}_{lboolean}$ (the set of all logical values), and $\mathcal{E}_{lstring}$ (the set of all string values).

For $\forall f \in \mathcal{F}$, we will define: $f : \mathcal{L} \times \ldots \times \mathcal{L} \to \mathcal{E}$, where the set $\mathcal{L}$ is defined by $\mathcal{L} = \{\bullet, \text{LispCell, Integer, Real, String}\}$, where the $\bullet$ represents that some operations or functions of A_SECD machine have no arguments.

## 6. SEMANTICS OF A FUNCTION DEFINED UNDER LispCell DATA TYPE

The method of denotations is modified in the following sense. Firstly, we want to preserve all the advantages of a very reach level of abstraction, and the elegant mechanism to define the semantics of programs in terms of classical mathematical notations such as functions, provided by the denotational semantics method. But, we also want to put into them some kind of 'operational' view, which is more suitable for implementations.

For these reasons we introduce some modifications of denotational semantics, putting into this approach some terms from object oriented style of programming. The semantics of some valid constructions of a language, can be defined using the definition of **semantic class**.

In this section we will give some examples of the denotational semantics definitions using the object-oriented modifications for some operations defined under the S-expressions.

**Converting Functions**. These functions convert data between some subtypes of LispCell data type, which are used in the implementation of S-expressions. The converting functions are: **ValueRealLC, ValueIntegerLC, CostIntegerLC, ConstRealLC, ConstStringLC, ValueStringLC,** etc. Their semantic definitions are:

Class *ConvFun*

Mapping
$\text{ConstRealLC} : \mathcal{E}_{lReal} \to \mathcal{E}_{LCreal}$
$\text{ValueRealLC} : \mathcal{E}_{LCreal} \to \mathcal{E}_{lReal}$
$\text{ConstIntegerLC} : \mathcal{E}_{lInteger} \to \mathcal{E}_{LCinteger}$
$\text{ValueIntegerLC} : \mathcal{E}_{LCInteger} \to \mathcal{E}_{lInteger}$
.....

Rules
$\text{ConstRealLC}(r : \mathcal{E}_{lReal}) : \mathcal{E}_{LCreal} \equiv$
$\text{Eval}_{A\_SECD} [\![ \text{ConstRealLC}(r) ]\!] \rho = r_{LC};$
$\text{ValueRealLC}(r_{LC} : \mathcal{E}_{LCreal}) : \mathcal{E}_{lReal} \equiv$
$\text{Eval}_{A\_SECD} [\![ \text{ValueRealLC}(r_{LC}) ]\!] \rho = r;$
$\text{ConstIntegerLC}(i : \mathcal{E}_{lInteger}) : \mathcal{E}_{LCinteger} \equiv$
$\text{Eval}_{A\_SECD} [\![ \text{ConstIntegerLC}(i) ]\!] \rho = i_{LC};$
$\text{ValueIntegerLC}(i_{LC} : \mathcal{E}_{LCInteger}) : \mathcal{E}_{lInteger} \equiv$
$\text{Eval}_{A\_SECD} [\![ \text{ValueIntegerLC}(i_{LC}) ]\!] \rho = i;$
.....

End; (* *ConvFun* *),

where $\rho$ is an arbitrary context in which all bindings of variable to their values are performed.

Now, we could easily proved the following statements:

**Theorem 6.1.** $(\forall x \in \mathcal{E}_{lReal})(\text{ValueRealLC}(\text{ConstRealLC}(x)) = x)$,
**Theorem 6.2.** $(\forall y \in \mathcal{E}_{LCreal})(\text{ConstRealLC}(\text{ValueRealLC}(y)) = y)$,
**Theorem 6.3.** $(\forall i \in \mathcal{E}_{lInteger})(\text{ValueIntegerLC}(\text{ConstIntegerLC}(i)) = i)$,
**Theorem 6.4.** $(\forall j \in \mathcal{E}_{LCinteger})(\text{ConstIntegerLC}(\text{ValueIntegerLC}(j)) = j)$, etc.

Logical Functions. The semantic class for some logical operations are:

**Class** *LogFun*
**Mapping** AndLC : $\mathcal{E}_{LCBoolean} \times \mathcal{E}_{LCBoolean} \rightarrow \mathcal{E}_{LCBoolean}$
.....
**Rules** AndLC($l_1$: $\mathcal{E}_{LCBoolean}$, $l_2$ : $\mathcal{E}_{LCBoolean}$) : $\mathcal{E}_{LCBoolean}$ ≡
$Eval_{A\_SECD}$ [[ AndLC($l_1$,$l_2$) ]] $\rho$ =
{$Eval_{A\_SECD}$ [[ $l_1$ ]] $\rho$ ∧ $Eval_{A\_SECD}$ [[ $l_2$ ]] $\rho$} ∈ $\mathcal{E}_{LBoolean}$,
$Eval_{A\_SECD}$ [[ $l_1$ ]] ∈ $\mathcal{E}_{LBoolean}$,
$Eval_{A\_SECD}$ [[ $l_2$ ]] ∈ $\mathcal{E}_{LBoolean}$,
$Sem_{LC}$ [[ $l_1$ ]] ∈ $\mathcal{E}_{LCboolean}$,
$Sem_{LC}$ [[ $l_2$ ]] ∈ $\mathcal{E}_{LCboolean}$,
StateTrans(($l_1$ $l_2$.S), E, (AND.C), D, L);
.....
**End**; (* *LogFun* *).

Arithmetic Functions. Before we define the class of arithmetic functions, let us introduced some new notations:

$$\mathcal{E}_{LNumpod} \equiv \mathcal{E}_{LInteger} \cup \mathcal{E}_{LReal},$$
$$\mathcal{E}_{LCnumpod} \equiv \mathcal{E}_{LCinteger} \cup \mathcal{E}_{LCreal}, \text{ i}$$
$$\mathcal{E}_{LCstruct} \equiv \mathcal{E}_{LCpar} \cup \mathcal{E}_{LClist},$$

for the subsets of numeric data for the implementation language and the data type LispCell. Then, the semantic definitions of the arithmetic functions are:

**Class** *ArithFun*
**Mapping** AddLC : $\mathcal{E}_{LCnumpod} \times \mathcal{E}_{LCnumpod} \rightarrow \mathcal{E}_{LCnumpod}$
.....
**Rules** AddLC($l_1$ : $\mathcal{E}_{LCnumpod}$, $l_2$ : $\mathcal{E}_{LCnumpod}$) : $\mathcal{E}_{LCnumpod}$ ≡
$Eval_{A\_SECD}$ [[ AddLC($l_1$,$l_2$) ]] $\rho$ =
{$Eval_{A\_SECD}$ [[ $l_1$ ]] $\rho$ ⊕ $Eval_{A\_SECD}$ [[ $l_2$ ]] $\rho$} ∈ $\mathcal{E}_{LNumpod}$,
$Eval_{A\_SECD}$ [[ $l_1$ ]] ∈ $\mathcal{E}_{LNumpod}$,
$Eval_{A\_SECD}$ [[ $l_2$ ]] ∈ $\mathcal{E}_{LNumpod}$,
$Sem_{LC}$ [[ $l_1$ ]] ∈ $\mathcal{E}_{LCnumpod}$,
$Sem_{LC}$ [[ $l_2$ ]] ∈ $\mathcal{E}_{LCnumpod}$,

(( AddLC($l_1$,$l_2$) = { ConstRealLC(ValueRealLC($l_1$) ⊕ ValueRealLC($l_2$)) ∈ $\mathcal{E}_{LCreal}$, if $l_1$, $l_2$ ∈ $\mathcal{E}_{LCreal}$ } ∨
( AddLC($l_1$,$l_2$) = { ConstIntegerLC(ValueIntegerLC($l_1$) ⊕ ValueIntegerLC($l_2$)) ∈ $\mathcal{E}_{LCinteger}$, if x, y ∈ $\mathcal{E}_{LCinteger}$
ConstRealLC(ValueRealLC($l_1$) ⊕ ValueIntegerLC($l_2$)) ∈ $\mathcal{E}_{LCreal}$, if $l_1$ ∈ $\mathcal{E}_{LCreal}$ ∧ $l_2$ ∈ $\mathcal{E}_{LCinteger}$ } ),
( AddLC($l_1$,$l_2$) = { ConstRaelLC(ValueIntegerLC($l_1$) ⊕ ValueRealLC($l_2$)) ∈ $\mathcal{E}_{LCreal}$, if $l_2$ ∈ $\mathcal{E}_{LCreal}$ ∧ $l_1$ ∈ $\mathcal{E}_{LCinteger}$ }

StateTrans(($l_1$ $l_2$.S), E, (ADD.C), D, L);
.....
**End**; (* *ArithFun* *).

## 7. SEMANTICS OF A_SECD MACHINE LANGUAGE

In this section we will give some examples of the denotational semantics definitions using the object-oriented modifications for some valid language construction of A_SECD machine language.

Constructing Operations. Constructing operations form some structure object (lists or pairs), using the top stack items. We define, for example, a semantic class for the A_SECD machine operation CONS, which forms a pair of two top stack items, with:

**Class** *ConstructOper*
**Mapping** ConsLC : $\mathscr{E} \times \mathscr{E} \rightarrow \mathscr{E}_{LCpair}$
.....
**Rules** ConsLC($l_1 : \mathscr{E}, l_2 : \mathscr{E}$) : $\mathscr{E}_{LCpair} \equiv$
$Eval_{A\_SECD} [\![ ConsLC(l_1,l_2) ]\!] \rho =$
$\{'(' \odot Eval_{A\_SECD} [\![ l_1 ]\!] \rho \odot '.' \odot Eval_{A\_SECD} [\![ l_2 ]\!] \rho \odot ')'\} \in \mathscr{E}_{LCpair}$,
$Eval_{A\_SECD} [\![ l_1 ]\!] \in \mathscr{E}$,
$Eval_{A\_SECD} [\![ l_2 ]\!] \in \mathscr{E}$,
$Sem_{LC} [\![ l_1 ]\!] \in \mathscr{E}$,
$Sem_{LC} [\![ l_2 ]\!] \in \mathscr{E}$,
StateTrans($(l_1\ l_2.S)$, E, (CONS.C), D, L);
.....
End; (* *ConstructOper* *), where $\odot$ is operation of concatenation of characters or strings, and the class of *StateTrans* is defined by:

**Object** *StateTrans*
**Mapping** StateTrans : $\mathscr{E} \times \mathscr{E} \times \mathscr{E} \times \mathscr{E} \times \mathscr{E} \rightarrow \mathscr{E} \times \mathscr{E} \times \mathscr{E} \times \mathscr{E} \times \mathscr{E}$
**Rules** StateTrans($l_1 : \mathscr{E}, l_2 : \mathscr{E}, l_3 : \mathscr{E}, l_4 : \mathscr{E}, l_5 : \mathscr{E}$) : $\mathscr{E} \times \mathscr{E} \times \mathscr{E} \times \mathscr{E} \times \mathscr{E} \equiv$
**Case** HeadLC($l_3$) **Of**
ADD: $'(' \odot Stack_{Push}(Stack_{Pop}(l_1) \oplus Stack_{Pop}(l_1), l_1) \odot ',' \odot$
$l_2 \odot ',' \odot Stack_{Pop}(l_3) \odot l_4 \odot ',' \odot l_5 \odot ')'$
CONS: $'(' \odot Stack_{Push}('(' \odot Stack_{Pop}(l_1) \odot '.' \odot Stack_{Pop}(l_1) \odot ')', l_1) \odot ',' \odot$
$l_2 \odot ',' \odot Stack_{Pop}(l_3) \odot l_4 \odot ',' \odot l_5 \odot ')'$
.....
End; (* *StateTrans* *).

Selecting Operations. The class of selecting operations of A_SECD machine, in terms of denotational semantics, in the object-oriented approach is defined with:
**Class** *SelectOper*
**Mapping** HeadLC : $\mathscr{E} \rightarrow \mathscr{E}$
.....
**Rules** HeadLC($l_1 : \mathscr{E}$) : $\mathscr{E} \equiv Eval_{A\_SECD} [\![ x ]\!] \rho \in \mathscr{E}$
if $l_1 = (x.y) \wedge x \in \mathscr{E}$
$Eval_{A\_SECD} [\![ HeadLC(l_1) ]\!] \rho = \{$
*Error*, otherwise
$Eval_{A\_SECD} [\![ l_1 ]\!] \in \mathscr{E}$,
$Sem_{LC} [\![ l_1 ]\!] \in \mathscr{E}$,
StateTrans($(l_1\ l_2.S)$, E, (CAR.C), D, L);
.....
End; (* *SelectOper* *).

In all above definitons, the variables S, E, C, D and L are global, and $Eval_{A\_SECD}[\![ I ]\!] \in \mathcal{E}$ where $I \in \{S, E, C, D, L\}$.

## 8. CONCLUSION

A completely new approach to the definition of denotational semantics, combined with operational semantics and based on object-oriented methods, is introduced. This method has an advantage in strict mathematical combination of denotational and operational view of defining semantics of any programming language.

This approach is very suitable for the application in axiomatic definitions of a programming language, because it is easy to construct and prove a lot of theorems, which hold in that programming language.

## REFERENCES


[Bulv89] Budimac Z., Ivanović M., "New Data Type in Pascal", Proc. of the DECUS Europe Symposium, The Hague, Holland, pp. 192-199., 1989.

[Bulv90a] Budimac Z., Ivanović M., "An Implementation of Functional Language Using S-Expressions", 14th Information Technologies Conference 'Sarajevo-Jahorina 1990', Sarajevo, pp. 111-1-111-8, 1990.

[Bulv90b] Budimac Z., Ivanović M., "A Useful Pure Functional Interpreter", Proceedings of 12. International symposium "Computer at the University", Cavtat, 3.17.1-3.17.6, 1990.

[He80] Henderson P., Functional Programming, Prentice Hall, New York, 1980.

[Je92a] Jerinić Lj., "A_LispKit Lisp-Description and Implementation", Informatica, (in print), 1992.

[Je92b] Jerinic Lj., "Functional Programming Language A_LispKit Lisp", Rew. of Res. Ser. Mat. (in print), 1992.

[La64] Landin P. J.,"The mechanical evaluation of expressions", Computer Journal, Vol. 6, pp. 308-320, 1964.

[Me90] Meyer B., Introduction to the Theory of Programming Languages, Prentice Hall, New York, 1990.

[St83a] Stojković V., Stojmenović I., Jerinić Lj., Mirčevski J., Kulaš M., "Dynamic Memory Management For the Usage and Implementation of Programming Languages" (in serbian), Proceedings of 5. International symposium "Computer at the University", Cavtat, 135-142, 1983.

[St83b] Stojković V., Stojmenović I., Jerinić Lj., Mirčevski J., Kulaš M., "Implementation of the Simulator of the SECD Machine" (in serbian), Proceedings of 27. Conf. of ETAN, Struga, Vol. IV, 337-344, 1983.

[St84] Stojković V., Mirčevski J., Jerinić Lj., Stojmenović I., "LispKit Lisp language-version ARL" (in serbian), Bulletin of the Region Department of Informatic Novi Sad, pp. 55-61, 1984.

[Sto84] Stojmenović I., Stojković V., Jerinić Lj., Mirčevski J., "On implementation of a translator from LispKit Lisp language into the SECD machine language in FORTRAN" (in serbian), Informatica, Vol. 1, pp. 57-64, 1984.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 69 - 76

# Prover LEIBNIZ

Vladan Krstic & Milena Radnovic

Lenjinova 18, 26000 Pancevo, YU, 013/45-195
T.Koscuska 66, 11000 Beograd, YU, 011/180-601

## ABSTRACT

Program LEIBNIZ is a prover for any finitely axiomatized formal theory and specially predicate and propositional calculus. The proof of a predicate or propositional formula we derive indirectly - the negation of the formula is translated into a formal theory and we deduce contradiction from it. Equality is also included.

**KEY WORDS:** formal theory, proof, shortened proof, depth of proof, equality, inequality, depth of agreements, 'and-then' form

## 1. INTRODUCTION

Program LEIBNIZ is a prover for any finitely axiomatized formal theory and specially predicate and propositional calculus. It is written in Prolog (Arity Prolog V5.1) and its size is about 8k. It consists of preparing and working part.

Preparing part (only for predicate and propositional formulae):

- negate the formula,
- translate to 'and-then' form to quantifiers for minimizing arity of Skolem functions,
- push the negation to atomic formulae,
- rename variables,
- translate to prenex normal form,
- skolemize (replace existential variables with Skolem functions),
- replace universal variables with prolog variables,
- translate into 'and-then' form and delete conjunctions which are surplus (propositional),
- add contrapositions,
- delete surplus specific for predicate case,
- transform into formal theory (productional system),
- find predicates which can lead to contradiction.

Working part (MPPM_EIE - Modus Ponens Prolog MACHINA with Eqality and InEquality) duces the proof or, in the predicate and propositional case, produces the contradiction from the nal theory.

## 1. PROBLEM DESCRIPTION

Suppose we have a formal theory T with finite number of axioms and finite number of inference rules. Any rule is of the form F1, ..., Fn -> G, where F1, ..., Fn, G are formulae of the theory T. Our basic problem is to decide whether some given formula F is a theorem of the theory. We will use the following definition of proof, which differs a little from the standard one:

Proof is a finite sequence A1, ..., Ak, F1, ..., Fm of formulae of theory T, satisfying the conditions:

- A1, ..., Ak are all axioms of the theory T
- formulae Fi are deduced by an application an inference rule to some preceding formulae from the sequence
- an instance of a formula from the sequence can not be placed after that formula.

Theorems are last members of proofs. Subsequence F1, ..., Fm is termed the shortened proof. The axioms are excluded from it. It is clear that any usual proof can be transformed into a proof satisfying this definition.

## 2.SEARCH FOR A SOLUTION

Suppose we have a finite formal theory T and its formula F. The question if F is a theorem of the theory T could be discussed as follows:

1) First, we check if F is an axiom. If it is, the process is finished with the answer YES. Else, we go to the step 2.

2) We look for the first rule whose conclusion G can be unified with F. Then, if there is no such a rule, the process is finished with the answer NO. Else, let such a rule be F1, ..., Fk -> G. For further discussion of provability of the formula F, now we will discuss provability of formulae F1, ..., Fk (we assume that some variables have been instantiated because of unification). Then, if all formulae F1, ..., Fk are proved, the process is finished with the answer YES. Else, for proving the formula F, we go to the step 2, but we are looking now for the next rule whose conclusion can be unified with the formula F.

It is clear that this process (with some more details) is indeed the standard prolog mechanism. Thus, it has some well known prolog 'insufficiencies'. Here are some examples.

**EXAMPLE 1:**

Suppose that, in proof of C, we need a proof of a formula B, and a rule B1, ..., Bn -> B is used for it. Suppose also, for a proof of a premise Bi, a proof of the formula B is needed again. It is clear that in this case, proving of C will never be finished. For eliminating this insufficiency, in our program we draw up a record of the formulae which we have to prove. If we need a proof of already recorded formula F, then we will try to prove F in some other way.

**EXAMPLE 2:**

Suppose we have to use the rule f(g(X)) -> f(X). It is clear that consecutive application of this rule might never finish. To deal with such cases, we limit DEPTH OF PROOF. It means that we leave a rule which has not lead to a proof up to the given depth, and we try with the next rule whose conclusion can be unified with the given formula.

DEFINITION:

If the formula F is an axiom or if it is already proved (it is placed in the part of the proof built so far), then we consider it proved with any depth $\geq 0$.

If the formula F is proved by a rule of the form F1, ..., Fm -> F, then we consider F proved with the depth $N > 0$ if formulae F1, ..., Fm are proved with the depth N-1.

ILLUSTRATION:

Suppose we have the formal theory T = { a, b <- (c, d), c <- a, d <- c }. It is possible to prove b with the depth 2. Indeed: To prove the formula b with the depth 2, by the rule b <- (c,d), we need a proof of the formulae c and d with the depth 1. To prove c with the depth 1, by the rule c <- a , we need a proof of a with the depth 0, which we have because a is an axiom. To prove d with the depth 1, by the rule d <- c , we need a proof of c with the depth 0, which we have because c is placed in the part {a, c} of the proof which we already have.

With limited depth of proof, the process of proving will be finished, because the number of axioms and rules is finite. If this process ends with the answer YES, then the formula is a theorem. If it ends with the answer NO, then we only know that the formula has no proof of the given depth.

LEMMA:

For any theorem F, there exists a number N such that F can be proved with the depth N.

PROOF OF LEMMA:

Suppose we have a theorem F of a theory with k axioms. We are going to prove the lemma by general induction with respect to the length of the proof of F.

If the length of the proof is k, then F is an axiom. Thus, it is proved, for example, with the depth 0. Let $n > k$ be given, and suppose the lemma holds for all theorems whose length of proof is less then n.

If the length of the proof for F is n, then F is deduced by some rule F1, ..., Fm -> F. The lengths of proofs of formulae F1, ..., Fm are less than n. Thus, by the induction hypothesis, the formulae are proved with depths d1, ..., dm respectively. Then the formula F is proved with the depth, for example, max { d1, ..., dm } + 1.

EXAMPLE 3:

Suppose, during the process of proving, we had proved an instance F' of some formula F (F' is F with variables replaced by terms). But, suppose also that it turned out it was impossible to complete the proof using F'. Then, we find another instance F" of F. If F" is an instance of F', we will not even try to use it, because we have already had more general case and we did not succeed.

It is allowed to bind the depth of the proof for any particular premise of a rule.

## 3. FORMULAE WITH EQUALITY

For theories with equality, we could use the usual axioms and rules of reflexivity, transitivity and agreement with operations and relations of the language. But, it turns out that the number of different possibilities for proving in that case is too large. Thus generation of even simple proofs is very complicated. That's the reason why equality is implanted in the process of proving. In that way, the number of deduction tries of a given equality formula is decreased.

Let there be given a theory with equality, without axioms and rules of inference of equality logic (because they are implanted in the process of proving). Suppose we are trying to prove a formula in the theory, we have built a part F1, ..., Fk of the proof, and now we have to prove an equality U = V with the depth n.

If U and V can be unified, we consider the equality proved with any depth (reflexivity).

If the equality U = V or V = U can be unified with Fi, for some i, we consider it proved with any depth (symmetry).

So, we consider reflexivity and symmetry obvious, and they don't appear explicitly in a proof. But we consider transitivity and agreement rules as built in the system. For easier control, we introduce depth of application of agreement rules which is independent of depth of proof. We have depth of agreement of equality with operations and with relations separately. From now on, depth will be denoted by (n, no, nr) where n is the depth of proof as defined above, and no and nr are depths of agreement with operations and relations. We can not apply agreement rule if the corresponding depth is 0.

U = V is proved by transitivity rule to the depth (n, no, nr) if U = Z is proved with the depth (n-1, no, nr) but not by transitivity rule, and Z = V with the depth (n-1, no, nr).

Transitivity defined in this way is correct, because if there is a transitive linking U with V, then there exists Z which is directly linked with U, and transitively with V.

f(X) = f(Y) is proved by the rule of agreement with the operation to the depth (n, no, nr) if X = Y is proved to the depth (n, no-1, nr).

p(X) is proved by the rule of agreement with the relation to the depth (n, no, nr) if X = Y and p(Y) are proved to the depth (n, no, nr-1).

If there exists a proof for U = V in equality logic, then there exists a proof in this system too, to some depth (n, no, nr), because the depths of agreement are depths of applications of agreement rules. The statement obviously follows from Lemma. Also, if we proved something in this system, the same proof is a proof in equality logic. This means that proving of an equality in this way is equivalent to proving in equality logic.

In this way, the number of possibilities for proving is greatly decreased, specially for transitivity, and because of agreement depth, work with long terms is limited too.

## 4. EXTENSION TO THE PREDICATE AND PROPOSITIONAL CALCULUS. PROBLEM OF NEGATION

It is possible to use this prover (MPPM_EIE) for proving formulae of the predicate and propositional calculus. Proving is indirect - we try to deduce contradiction from the negated formula.

Thus, we negate the formula, and we eliminate the quantifiers by prenexing, skolemizing and replacing variables by prolog variables. The formula transformed in that way is then translated to 'and-then' form.

DEFINITION:

A formula is in 'and-then' form if it is a conjunction and all its conjuncts are either literals (i.e. atomic formulae or negation of atomic formulae) or implications whose consequents are literals and antecedents are conjunctions of literals.

Any formula can be reduced to an 'and-then' form. 'And-then' form is not unique.

From the 'and-then' form we delete tautologies since they are of no use in proving. These are:

(1) ... and P and ... then P,
(2) ... and A and ... and not A and ... then B.

Also, we arrange brackets to make the association to the right and replace conjuncts of the form A and A by A to avoid repetition.

Formulae can contain negation, but it is not the usual negation, because the connection between implication and negation is missing (contraposition). This is the only incompleteness of negation because the only axiom which regulates this connection is the contraposition. Consequently, we add contrapositions, and with negation we will work like with any other predicate.

In the case of predicate calculus, together with the formulae of the form (1) and (2), we also delete formulae of the form:

(2') ... and A and ... and not C and ... then B,

where A and C can be unified, but not formulae of the analogous form (1'), because this would decrease the possibility of proving in the formal theory which we are going to describe.

Then, we consider implications as inference rules, and literals as axioms. We will try to derive contradiction in this formal theory.

There is a problem when working with equality, because agreement and transitivity are not implications, but built-in rules, and contraposition does not work for them, so we add contrapositions of agreement and transitivity.

As these rules are of general character (for example: not (X = Y) <- not (f(X) = f(Y)), where f is any function), it is necessary to say in advance which operations and relations can be used, i.e. it is necessary to declare operations and relations which could be in an inequality. With this, work with negation is completed.

In particular, when we translate a formula of propositional calculus to 'and-then' form and we delete all tautologies from it, if there is not any conjunct left, then the given formula is a tautology, otherwise it is not. All true conjuncts of an 'and-then' form must be of the form (1) or (2). It follows what validity of a formula of propositional calculus could be checked in that way too.

## 5. THE QUESTION OF COMPLETENESS OF THE PROCESS

The possibility of proving a theorem of given formal theory is equivalent to the possibility of proving with LEIBNIZ. Direction <= is obvious. Direction => follows from Lemma.

Specially, in case of formulae of propositional or predicate calculus, the procedure used in LEIBNIZ is not complete.

If 'and-then' form of a formula is consistent, then the formal theory derived from it is consistent too. It follows that we can not deduce contradiction from the negation of a non-valid formula. Thus, if we deduce contradiction, the formula is valid indeed.

If the 'and-then' form is inconsistent (this will always happen when the formula is valid), the derived formal theory doesn't have to be inconsistent (for example, if it has no axioms, or the axioms have no connection with the inference rules).

If A <=> B it still may happen that the theory derived from A differs from the theory derived from B.

**EXAMPLE:**

If we apply the preparing part of the prover to formula p then p, theory { not p, p } will be derived, and it is possible to prove contradiction from it.

If we apply the preparing part of the prover to formula p iff p, theory { not p <- p , p <- not p } will be derived, but it is not possible to prove anything from it, because it has no axioms. But, (p iff p) <=> (p then p).

It is desirable, when proving an equivalence, to prove separately the two implications, because that way we remove partly incompleteness, as in example 2 of next paragraph.

## 6. EXAMPLES

LEIBNIZ is a prover of axiomatic - productional type. That's why proving of some formulae (especially with equality) is not so simple as it seems. Between intuitively close problems could exist great combinatory difference (for example between PA |- 1+1=2 and PA |- 2+2=4 ).

LEIBNIZ can prove theorems in formal theories with low degree of combinatory complexity (on the average, it processes 5000 - 15000 formulae per hour - AT 16 MHz).

If we have a predicate formula of few lines in length, such that it does not describe any problem of a formal theory type, then it is almost always possible to prove the formula, generally for few minutes at most. Classical (bookish) examples of valid formulae are proved almost immediately.

PA |- 2 + 2 = 4

We will take from Peano arithmetic only what we estimate is enough for proving the equality. If we take axiom schema X + 0 = X, then it is practically impossible to produce a proof because of the huge world of formulae which the axiom generates. The equality 1 + 1 = 2 is proved after 148 processed formulae only, because the proof needs lower depth, and thus the world of accessible formulae is a lot smaller. But, when proving 2 + 2 = 4, this world becomes enormous. This equality can be deduced elegantly, because we only need 2 + 0 = 2 :

Axioms :

```
one :: 0 prim = 1
two :: 1 prim = 2
thr :: 2 prim = 3
fou :: 3 prim = 4
+_p :: X1 + X2 prim = (X1 + X2) prim
2+0 :: 2 + 0 = 2
```

Proved : 2 + 2 = 4 td [6,2,0] <- 15

The number of formulae passed through MPPM_EIE is 39.
The time used [0,0,23]. ([h,m,s] on AT 16 MHz)

Shortened proof :

1 :: 2 + 2 = 2 + 1 prim <- agree to (ref , sym(two))
2 :: 2 + 1 = 2 + 0 prim <- agree to (ref , sym(one))
3 :: 2 + 1 = (2 + 0) prim <- trans to (2 , +_p)
4 :: (2 + 1) prim = (2 + 0) prim prim <- agree to 3
5 :: (2 + 0) prim prim = (2 + 0 prim) prim <- agree to sym(+_p)
6 :: (2 + 0) prim = 2 prim <- agree to 2+0
7 :: 2 + 0 prim = 2 prim <- trans to (+_p , 6)
8 :: (2 + 0 prim) prim = 2 prim prim <- agree to 7
9 :: 2 prim prim = 3 prim <- agree to thr
10 :: 2 prim prim = 4 <- trans to (9 , fou)
11 :: (2 + 0 prim) prim = 4 <- trans to (8 , 10)
12 :: (2 + 0) prim prim = 4 <- trans to (5 , 11)
13 :: (2 + 1) prim = 4 <- trans to (4 , 12)
14 :: 2 + 1 prim = 4 <- trans to (+_p , 13)
15 :: 2 + 2 = 4 <- trans to (1 , 14)

## IF TWO EQUIVALENCE CLASSES INTERSECT THEN THEY ARE EQUAL

all x : p(x,x) and
all (x, y) : (p(x,y) then p(y,x)) and
all (x, y, z) : (p(x,y) and p(y,z) then p(x,z))
then
all (k, n) : (ex a : (p(k,a) and p(n,a)) then all x : (p(k,x) then p(n,x)))

Here is proved only: the class k is a subset of the class n. Converse can be proved similarly. After the preparing part, we have a formal theory from which we will prove a contradiction of the form ( not p(X,Y), p(X,Y) ).

Axioms :

ax_1 :: not p(g2,g3)
ax_2 :: p(g1,g3)
ax_3 :: p(g2,g4)
ax_4 :: p(g1,g4)
ax_5 :: p(X1,X1)

**Inference rules :**

**pi_1 :: not p(X1,X2) <- not p(X2,X1)**
**pi_2 :: p(X1,X2) <- (p(X1,X3) , p(X3,X2))**
**pi_3 :: p(X1,X2) <- p(X2,X1)**

**Possible contradictory predicates : p / 2**

**Proved : not p(g2,g3) td 0 <- ax_1**
**p(g2,g3) td 3 <- 3**

**The number of formulae passed through MPPM_EIE is 8.**
**The time used [0,0,2].**
Shortened proof :

1 :: p(g4,g1) <- pi_3 to ax_4
2 :: p(g4,g3) <- pi_2 to (1 , ax_2)
3 :: p(g2,g3) <- pi_2 to (ax_3 , 2)

## 7. CONCLUSION

The formal theory derived from a formula of predicate or propositional calculus is very "natural", i.e. relative positions of subformulae which have sense for us, are mostly preserved. Thus the proof can be easily understood (it is relevant for the formula).

If it is possible to prove a formula of propositional or predicate calculus by both LEIBNIZ and semantic tableau method, then our prover is almost always more efficient.

Arrangement of axioms and inference rules, and estimated depth of proof have great influence on efficiency of proving.

The question of completeness for predicate and propositional calculus is open.

## REFERENCES

- dr Petar Hotomski i dr Irena Pevac : Matematicki i programski problemi vestacke inteligencije u oblasti automatskog dokazivanja teorema
- Mario Radovan : Programiranje u PROLOGU

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 77 - 84

# The Implementation of PROG Mechanism in Pure Functional Programming Language

Nenad Mitić
Matematički Fakultet Univerziteta u Beogradu,
Studentski trg 16, POB. 550, YU-11001 Beograd
email: xpmfm23@yubgss21.bitnet

Key Words: functional programming, PROG mechanism, pure functional programming language, SECD machine, Lispkit Lisp language

**Abstract**

The conception of pure functional (or application) programming languages prohibits side effects, including the assignment of values to variables. For this reason the program cycle statement doesn't exist in application languages. The lack of this statement doesn't affect program execution. But, using the programming cycle statement and variables assignments combined with printing are very useful for debugging.This paper presents an original program implementation of PROG mechanism - the statement of the programming cycle in pure functional programming language Lispkit Lisp. The manner of implementation provides preserving the structure of Lispkit Lisp as a pure functional programming language. The program implementation is done by an extension of the the SECD machine simulator and a modification of the Lispkit Lisp language compiler into the machine language of SECD machine.

## 1. Introduction

Along with the recent advances in VLSI technology and new computer architectures, a significant development of various new programming methodologies has taken place. They had to provide more efficient using of new hardware. J. Backus gave a significant contribution in developing the methodology called ***functional programming*** with his paper [1]. This style of programming, also known as applicative programming, is characterized as "programming without the assignment statement". Functional languages in which assignment statements don't exist are called pure functional languages. Total absence of assignment statements and side effects (which are results of it) eliminate needs for variables in programs. Programs in functional programming are functions that map objects in objects. These functions are not depending on "outer" data (like global variables in non-functional programming). The programs become simpler and easier for understanding. Another advantage of functional programming languages is the existance of the simple mechanism for modification and combination of existing programs. This mechanism is enabled by using high-order functions which increase the level of abstraction in programming.

## 2. SECD machine and Lispkit Lisp language

SECD machine [3,4] represent one of the first models of computers on which some ideas of functional programming can be applied. The name SECD comes from the names of four main machine registers:

- *s* - the stack. It is used to hold intermediate results when computing the values of expressions.
- *e* - the environment. It is used to hold the values bound to variables during evaluation.
- *c* - the control. It is used to hold the machine-language program being executed.
- *d* - the dump. It is used as a stack to save values of other machine registers on calling a new function.

Each of the registers can be considered as a stack or a list (S-expression). When the simulator of the SECD machine is used for evaluation of pure functional language Lispkit Lisp [3], input and output data for the SECD machine are well-formed expressions as described in [3].

The implementation of the simulator of the SECD machine and Lispkit Lisp compiler was made, with some extensions, according [3], on PL/I language in an MVS/ESA TSO environment (on IBM 3090-17T).

## 3. PROG mechanism in Lispkit Lisp language

Loop statement, assignment statement (which implicitly introduce variables), goto, return and label statements introduce side effects in a pure functional programming language (and it isn't a pure functional language any more). On the other side, there is a rule that introducing any new instruction in Lispkit Lisp must preserve pure functionality. This condition can be satisfied if every PROG function is considered as a user defined elementary function. In this case, the programs contains only PROG functions. Assignment statement, variables used in assignment statements, goto, return and label statements are defined only inside the body of PROG function; every using of these statements outside of the body of PROG function is marked as a syntax error. The implementation contains:

1. PROG(ram) function. This is a base function and elements of PROG mechanism can be used only inside its body. Characteristics of PROG functions are:
   - PROG is n-ary function (n≥2).
   - The first argument of a PROG function is a list of variables. The list of PROG variables can be empty (NIL). All variables are initially set to NIL.
   - The other arguments of PROG function are evaluated, beginning with the second, according to the following rules:
     a. If the argument is an atom it must be a label and its value isn't evaluated. Labels can be numerical or symbolic atoms.
     b. If the argument of a function is GO, RETURN or SETQ its value is evaluated according to rules 2-4.

c. Otherwise, the rules of evaluation Lispkit Lisp function are applied.

- The value of PROG function is the value of the RETURN function, if it is evaluated. Otherwise, the value of PROG function is equal to the value of its the last argument.

- PROG function can be nested in another PROG function. In this case you can jump only within the PROG function that consists this GO statement.

2. RETURN function. RETURN is an unary function whose argument is a well-formed expression. The value of the function is the value of the argument. Evaluating of RETURN results in assigning its value to the PROG function (and finishing the evaluation of the PROG function).

3. GO function. GO is an unary function whose argument is a well-formed expression. The value of the function is the value of the argument. The effect of evaluating the GO function is a jump to the statement below the label which is equal to the value of the GO function (or error if such a label doesn't exist).

4. SETQ(uote) function. SETQ is a binary function. The first argument is an atom (the name of the variable). The second argument is a well-formed expression. The value of SETQ function is the value of the second argument. The side effect is changing the value of the first argument which is set to the value of the second one.

## 4. Prog mechanism - program implementation

### 4.1 Compiling into machine language

Functions of PROG mechanism are compiling into machine instructions which must produce side effects. These machine instructions need additional program support which is not initially included in the simulator of a SECD machine. For these reasons, translation of PROG mechanism functions will be described in two ways: by formal description using transactions like in [3] and by description of side effects of each machine instruction.

The translation to machine language has to satisfy following conditions:

- Translation of any PROG mechanism function or a label has to contain new machine instruction(s) (because the PROG mechanism function can't be expressed neither using existing SECD machine instructions nor using existing Lispkit Lisp functions).

- Translation of the PROG function has to contain information about the beginning and end of the PROG function (because of the possibility of the existance of nested PROG functions and identical labels within them).

- Translation to machine language must contain information about the number of arguments of translated PROG function.

According to this, the translation of expression (PROG arg $exp_1$ $exp_2$ ... $exp_n$) is

(PROG arg $exp_1$ $exp_2$ ... $exp_n$)*n = (bprog)|arg*n|$exp_1$*n|...|$exp_n$|(eprog)

where | is a symbol for concatenation, and exp*n is the translation of the expression exp with namelist n. The previous expression has the net-effect property. If PROG function has no argument, the instruction bprog (begin PROG) is evaluated in the following way:

s e (bprog 0.c) d save → s e c d ((s e c.d).save)

If Prog function has more than zero arguments brpog is evaluated as

s e (bprog $c_1$.c) d save → s ($c_1$.e) c d ((s e c.d).save)

In both cases there is an additional side-effect, which is not obvious in the formal description of the transaction. Separate field is used for storing

- value of s register
- value of e register
- value of d register
- return address from PROG function (which implicitly depends on register c).
- list of each labels with addresses existing in the PROG expression (excluding labels from nested PROG expressions).

This field is generated for every PROG expression and stands for a base of the mechanism which allows using identical label names in nested PROG expressions. Using an additional register *save* eliminates possible errors during garbage collection. Register *save* can be modified only by brpog and eprog instructions.

Instruction eprog (end PROG) is evaluated in the following way:

($s_1$.s) e (eprog.c) d ($save_1$.save) → ($s_1$.s') e' c d' save

The side effect of eprog instruction is reflected in taking values of the saved registers (s', e', d') from the field where they were stored during evaluation of bprog instruction. After that, the field related to the finished PROG expression is released. The net-effect property is provided by restoring general registers (from the field filled by bprog instruction) and saving the current value at the top of the register *s*.

Translation of an atom that denotes label is label * *n* = (lbl) | label while the effect of executing an instruction lbl is s e (lbl l.c) d → s e c d

Instruction lbl is used only in preprocessing PROG function because of fixing the label address. Instruction lbl is skipped during execution. Preprocessing is needed because a value of a label must be evaluated before code execution.

Translation of the expression (RETURN e) is (RETURN e) * *n* = e * *n* | (ret), while the effect of executing instruction ret is ($s_1$.s) e (ret.c) d → ($s_1$.s) e (eprog.c') d

where c' is the code that is the continuation of a program code after the code of a PROG function. The side effect of ret instruction is reflected in taking the return address (of PROG expression) from the field in which it is stored.

Translation of the expression (GO e) is (GO e) * *n* = e * *n* | (go), while the effect of executing instruction go is ($s_1$.s) e (go.c) d → s e c' d

where $s_1$ is a label, and c' is the code that is the continuation of a program code after the label whose value is $s_1$. The side effect is reflected in taking address from where the execution continues.

Translation of the expression (SETQ $e_1$ $e_2$) is (SETQ $e_1$ $e_2$) * *n* = $e_2$ * *n* | (setq) | loc

where loc = locate ($e_1$ *n*). The effect of executing instruction setq is

$$(s_1.s)\ e\ (setq\ c_1.c)\ d \rightarrow (s_1.s)\ e'\ c\ d$$

where $s_1$ is the value of the second argument of SETQ function, e' is a environment with the substituted value at the place which relates to variable $e_1$. This is the side effect of this instruction.

Appendix A contains the statements that extend the translator. Instructions setq, bprog, eprog, go, ret, lbl have codes 32,40,41,42,43,44.

### 4.2 Modifications of simulator of the SECD machine

A basic idea: when code 40 is found during the execution of the program, searching for a code 41 is started. All instructions between codes 40 and 41 are analyzed. Names and addresses of found labels are saved in dynamic variables (PL/I allocation variables). Analysis takes care about the level of nesting PROG functions. Every code 40 must have its pair, code 41. Every pair (40, 41) determinates a set of labels used in instructions between them. It is possible to jump to a label only from the instruction which is in the same PROG function, and doesn't belong to any nested PROG function. Code 41 denotes the address of the exit from PROG function. In the next step the values of registers *s,e,c,d* are saved in order to provide the net-effect property (restoring at the end of PROG function). It is done by using PL/I stack (not the stack of the SECD machine). Rules for program execution are:

- The code of instruction lbl and value that follows it in register *c* are ignored.
- When the instruction go is found the corresponding value of label (the value at the top of the stack *s*) is taken and the list LABEL is searched for address from which the execution will be continued.
- When the instruction ret is found the corresponding return address (of corresponding code 41) is taken.
- When the instruction eprog is found the registers *s*, *e*, *c*, *d* would be restored, valid variables released and all changes on the *save* register eliminated, which provide net-effect property.

Modifications in the SECD machine program simulator are given in appendix B.

References

[1] Backus, John - Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, CACM 21/8, August 1978, p. 613-641

[2] Darlington, J., Hendreson, P., Turner, D. A. - Functional programming and its applications, Cambridge University Press, Cambridge, 1982

[3] Henderson, Peter - Functional Programming: application and implementation, Prentice-Hall International, inc., London, 1980.

[4] Landin P.J. - The mechanical evaluation of expression, Computer Journal, Vol. 6, pp 308-320 (1964)

[5] Landin P.J. - The Next 700 Programming Languages, Comm of the ACM, Vol. 9, No. 3, March 1966

[6] OS PL/I Version 2 Programming Guide, SC26-4307-2, IBM

[7] OS PL/I Version 2 Language Reference, SC26-4308-2, IBM

## Appendix A: The statements for extending the translator

```
(IF (EQ  (CAR E) (QUOTE SETQ))
    (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 32)
            (CONS (LOCATION (CAR (CDR E)) N) C)))
 (IF (EQ (CAR E) (QUOTE GO))
     (COMP (CAR (CDR E)) N
           (CONS (QUOTE 42) C))
 (IF (EQ (CAR E) (QUOTE RETURN))
         (COMP (CAR (CDR E)) N
               (CONS (QUOTE 43) C))
 (IF (EQ  (CAR E) (QUOTE PROG))
    (LETREC (IF (ATOM (CAR (CDR E)))
                (IF (EQ (CAR (CDR E)) (QUOTE NIL))
                    (CONS (QUOTE 40)
                          (CONS (QUOTE 0)
                                (PROGNIL (CDR (CDR E)) N)
                          )
                    )
                    (QUOTE ERROR1)
                )
                (CONS (QUOTE 40)
                      (CONS (PRAZNO (CAR (CDR E)))
                            (PROGNIL (CDR (CDR E)) (CONS (CAR (CDR E)) ))
                      )
                )
            )
            (PRAZNO LAMBDA(X)
               (IF (EQ  X (QUOTE NIL))
                   (QUOTE NIL)
                   (CONS (QUOTE NIL)
                         (PRAZNO (CDR X) ) ) ) )
            (PROGNIL LAMBDA(E N)
             (IF (EQ E NIL)
                 (QUOTE ERRORPROG)
                 (IF (EQ (CDR E) (QUOTE NIL))
                     (IF (ATOM (CAR  E) )
                         (QUOTE ERROR10)
                         (COMP (CAR E) N
                               (CONS (QUOTE 2)
                                     (CONS (QUOTE NIL)
                                           (CONS (QUOTE 41) C)
                                     )
                               )
                         )
                     )
                     (IF (ATOM (CAR E) )
                         (CONS (QUOTE 44)
                               (CONS (CAR E) (PROGNIL (CDR E) N) )
                         )
                         (COMP (CAR E) N (PROGNIL (CDR E) N))
                     )
                 )
             )
            )
    )
```

## Appendix B: Modification in SECD machine program simulator

- In main program register SAVE is declared as

  declare save binary fixed(31,0);

- In procedure MARK the register SAVE is marked.

- Modifications in procedure EXEC are:

```
exec : proc(fn,args) returns( bin fixed(31));
       .
       .
       .
declare 1 prog ctl,
          2 s1 bin fixed(31),
          2 e1 bin fixed(31),
          2 d1 bin fixed(31),
          2 r1 bin fixed(31),
          2 return bin fixed(31),
          2 labela ptr;

 /* control promenljiva prog je osnovni stek. svaka generacija prog promenljive
pridružena je jednoj prog funkciji (bilo da se izvršava ta prog funkcija ili neka prog
funkcija ugnježdena u njoj. ona sadrži adrese tekućih vrednosti registara s, e, d kao
i adresu povratka iz te prog funkcije return. labela je pokazivač na niz labela koje
su pridružene tekućoj prog funkciji. */
```

```
declare 1 label based(sada),
          2 name char(15) var,
          2 adress bin fixed(31),
          2 next ptr;
 /* label je element liste koja sadrži ime i adresu labele                */
declare (sada,prvi,preth) ptr;
declare null builtin;
declare errp1 char(42)
              init('SETQ used outside PROG. Command terminated');
declare errp2 char(40) init('GO used outside PROG. Command terminated');
declare errp3 char(15) init('Undefined label ');
declare errp4 char(19) init('. Command terminated');
declare errp5 char(44)
              init('RETURN used outside PROG. Command terminated');

     .
     .
     .
save=nil;
     .
     .
     .
do until(ivalue(car(c))=21);
   select(ivalue(car(c)));
     .
     .
     .
    when(32) do;              /*  SETQ   */
      if br_prog=0
         then do;
                put file(sysprint) edit(errp1) (skip(2),a);
                stop;
              end;
      /* određuje se mesto promenljive u listi imena  */
      w=e;
      do i=1 to ivalue(car(car(cdr(c))));
         w=cdr(w);
      end;
      w=car(w);
      do i=1 to ivalue(cdr(car(cdr(c))));
         w=cdr(w);
      end;
      car(w)=car(s);
      c=cdr(cdr(c));
    end;
    when(40) begin;                    /* BPROG */
      declare help bin fixed(31);      /* čuva adresu od koje treba
                                          početi pretraživanje       */
      declare kraj bit(1) init('1'b);
      declare br_prog bin fixed(15);  /* koliko je nađeno kodova 40 */
      prvi=null;                     /* pokazivač na početak liste labela */
      alloc prog;                    /* alocira se element koji čuva
                                        informacije o tekućoj prog funkciji */
      prog.s1=s;                          /* čuvanje opšteg registra s  */
      prog.e1=e;                          /* čuvanje opšteg registra e  */
      prog.d1=d;                          /* čuvanje opšteg registra d  */
      save=cons(cons(s,cons(e,cons(c,d))),save);
     /* eliminisanje mogućnosti greške pri radu garbage collector-a */
      if ¬isnumb(car(cdr(c)))
         then e=cons(car(cdr(c)),e);
                 /* ako ima argumenata u prog funkciji njihove
               početne vrednosti (tj. nil-ovi) se dodaju okolini */
      help=cdr(c);
      c=cdr(cdr(c));                 /* odakle se nastavlja izvršavanje */
      br_prog=1;       /* inicijalizacija broja pročitanih kodova 40 */
      do while(kraj);
         help=cdr(help);
         if isnumber(car(help))
            then select(ivalue(car(help)));
                              /* traženje instrukcija go, ret i lbl */
                 when(40) br_prog=br_prog+1;
                 when(41) do;
                             br_prog=br_prog-1;
                             if br_prog=0
                                then do;
                                       kraj='0'b;
                                       return=help;
                     /* ako je ret od tekućeg a ne od unutrašnjeg
                          prog-a postavlja se adresa povratka */
                                     end;
                          end;
                 when(44) do;
                             if br_prog=1
                                then do;   /* postavljanje labele */
                                       alloc label;
                                       if prvi=null
                                          then prvi=sada;
                                          else preth->next=sada;
```

```
                                        preth=sada;
                                        next=null;
     /* dodeljivanje imena labele */    if isnumber(car(cdr(help)))
/* name je oblika char(15) var i zato */   then name=tostring(
/* se broj pretvara u char pomoću toostring */ ivalue(car(cdr(help))));
                                           else name=stringstore(
                                            ivalue(car(cdr(help))));
                                        adress=cdr(cdr(help));
                                     end;
                                  help=cdr(help);
                               end;
                        other;
                     end;  /* select */
       end; /* do while(kraj) */
          /* postavljanje pokazivača na listu labela za tekući prog */
       if prvi=null then labela=null;
                    else labela=prvi;
      end; /* begin kod when 40 */
      when(41)  begin;              /*  eprog  */
         /* restauracija opštih registara sa efektom čistog rezultata */
                s=cons(car(s),prog.s1);
                e=prog.e1;
                d=prog.d1;
                free prog;
                save=cdr(save);
                c=cdr(c);
      end;
      when(42)  begin;              /* go */
                declare povratak char(15) var;
                      /* ime labele na koju se prenosi izvršavanje */
                declare kraj bit(1) init('1'b);
                if br_prog=0
                   then do;
                           put file(sysprint) edit(errp2)(skip(2),a);
                           stop;
                        end;
                if isnumber(car(s))
                   then povratak=tostring(ivalue(car(s)));
                   else povratak=stringstore(ivalue(car(s)));
                s=cdr(s);
                if labela=null
                   then do;
                           put file(sysprint) edit(errp3,povratak,
                                                   errp4)(skip(2),a,a,a);
                           stop;
                        end;
                   else sada=labela;
                do while(kraj);
                   if name=povratak
                      then do;
             /* ako je labela nađena izvršavanje se  prenosti i i tu
                naredbu pomoću dodeljivanja njene adrese c registru */
                           c=adress;
                           kraj='0'b;
                        end;
                     else if next=null
                           then do;
                                   put file(sysprint) edit(errp3,
                                    povratak,errp4)(skip(2),a,a,a);
                                   stop;
                                end;
                           else sada=next;
                        end;
      end;
      when(43)  begin;              /*  ret */
                if br_prog=0
                   then do;
                           put file(sysprint) edit(errp5)(skip(2),a);
                           stop;
                        end;
                c=return;                /* izvršavanje se prenosi na
                                            adresu naredbe sa kodom 41 */
      end;
      when(44)                      /*  lbl  */
        c=cdr(cdr(c));    /* ignoriše se kod za labelu i sama labela.
            Labela je važna samo u predprocesiranju prevoda PROG f-je */
       .
       .
       .
 end; /* do until */
 return(car(s));
end; /* exec */
```

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA 92
NOVI SAD, OCTOBER 29-31, 1992., PP. 85 - 92

# ON COMPILATION OF PATTERN MATCHING IN SASL LANGUAGE*

Branislav Nikolajević[†], Zoran Budimac[‡]

[†]University of Novi Sad, Computing Center
Trg D. Obradovića 5, 21000 Novi Sad

[‡]University of Novi Sad, Faculty of Natural Sciences and Mathematics,
Institute of Mathematics, Trg D. Obradovića 4, 21000 Novi Sad

e-mail: {brana,zjb}%unsim@yubgef51.bitnet

**ABSTRACT:** A transformation of SASL equations into their non-equational equivalent is described. Equations of the source language are transformed only into usual operators of a target (functional) language, and not into special constructs. This transformation scheme is therefore more useful, because it can be applied on a broader range of target languages.
**Key-words:** Implementation of Functional Languages, SASL, Pattern Matching

## 1 INTRODUCTION

The research in the field of functional (or applicative) programming has been of growing interest to computer scientists since its origin in the 1950's. Functional (or applicative) programming languages, especially pure ones, are of great importance for the future of computing, because they could solve the so-called "software crisis." Functional programs are short, concise, easy to maintain, and their (in)correctness is easily proved formally. They also offer a natural approach to parallelism and parallel programming languages.

Purely functional languages lack everything that is essential to procedural (or imperative) languages: statements, explicit sequencing and side effects. Purely functional languages are referentially transparent, insensitive to evaluation order, have strong mathematical basis and a small set of built-in features. Therefore, they are easy to learn and follow. Furthermore, all program identifiers are lexically scoped (i.e. bound at compile-time and not at run-time) and all functions are first-class objects (i.e. have the same rights like other data types). Finally, most purely functional languages have non-strict semantics (i.e. expressions are evaluated only when necessary), which gives those languages the potential of dealing with infinite data structures. For more details and an overview of functional programming languages and style see for example [1].

The key issue in functional programming research is the definition of purely functional languages and implementation of their processors. Recent functional languages have many syntactic enhancements such as conditional expressions (or "guards"), list comprehension (or ZF-expressions) or pattern matching, which all lead to more readable programs. Since its introduction in functional languages SASL and NPL, pattern matching became almost standard feature of most modern functional languages. Intuitively, pattern matching means that functional program consists of a set of equations, where certain equation is applied if its left-hand side is matched against the current state of function evaluation. Usually only the pattern of the left-hand sides is identified (hence the name pattern matching). In Fig. 1 is displayed a typical example of a function definition written in equational form. Function app appends two lists.

```
app nil s = s
app (x:s) t = x : (app s t)
```

**Fig. 1** An example of an equational functional program

* This work is supported by Science Fund of Serbia

Since the set of equations is not a "natural form" of a functional program, it has to be transformed into some intermediate code based on lambda-calculus or into some existing functional language to be (efficiently) executed. In most cases known from the literature (for example [2,4]) equations are transformed into special constructs or operators, developed exclusively for pattern matching compilation. However, we show, in this paper, that equations of a functional language can be successfully transformed into a set of "classic" operators found in all functional languages, if the source language has only built-in data types. The paper deals with a compilation of pattern matching of SASL language, into a corresponding Scheme or LispKit LISP expression with the same semantics.

The rest of the paper describes a compilation of pattern matching of SASL language into a subset of some standard (functional) language. Section 2 defines the patterns, pattern matching and its semantics, section 3 shortly introduces SASL language and section 4 describes our implementation of SASL pattern matching with respect to the set of standard (or "common") operators, found in all functional languages. Section 5 concludes the paper.

## 2 PATTERNS AND PATTERN MATCHING

The concept of data constructor is closely related to the definition and concept of patterns and pattern matching. We proceed with a short introduction to data constructors.

### 2.1 Data Constructors

Data constructors can be observed as a special functions that assist "to construct or bind together data" [2]. The only difference between data constructors and "ordinary" functions is that constructor functions don't possess associated rules (for transformation or reasoning about them). Data constructors can be built-in (in which case they can take the form of constructor operators) or introduced by the user in the form of algebraic data types (user-defined data types). For example, a data constructor that can be found in all functional languages in the form of explicit constructor or constructor operator is the pair of constructors CONS and NIL, which together serve to build lists of data. For example, the following call: CONS(1, CONS(2, CONS(3, NIL))) builds a list of data containing numbers 1, 2 and 3. If more than one data constructor is used for building a single data type, it is often called sum-constructor, otherwise it is called product-constructor.

### 2.2 Patterns

Pattern is in functional languages defined as:

- a variable or
- a constant or
- an infix constructor operator pattern of the form $p_1\ o\ p_2$, where $p_1$ and $p_2$ are patterns and $o$ is binary constructor operator.
- a constructor pattern of the form $c\ p_1\ p_2\ \ldots\ p_n$, where $c$ is constructor of arity n and $p_1$, $p_2$, ..., $p_n$ are patterns.

### 2.3 Pattern Matching

Patterns can be used in the place of every argument on the left-hand sides of equations that constitute a function definition. Patterns are used in case analysis during function evaluation in the following way: when actual arguments are matched against patterns on the left-hand side of the equation, corresponding right-hand side is selected for evaluation. In implementation of pattern matching, there are several issues to be concerned of:

- overlapping patterns, which means that in function definition exists at least one pattern that can be applied in more than one case. In case of overlapping patterns the order of equation

in function definition is significant.

- non-exhaustive sets of equations, which means that equations not necessary define all possible cases that can appear in actual arguments.

- repeated variables, which means that if in equation are allowed same variable names, then they denote the same values of actual arguments.

### 2.4 Transformation rules

Function definitions involving patterns can be transformed into lambda-calculus that is enhanced with appropriate reduction rules or special built-in functions. To define formally semantics of pattern matching, we'll transform the set of equations into lambda-calculus enhanced with lambda abstractions depending on patterns (and not only on variables) and with built-in constant called FAIL. We'll assume that there is already transformation function T, which transforms the constructs of a source language into the lambda-calculus with constants (and built-in functions). For possible definitions of function T see [2,3,4].

Function definition which consists of multiple equations is transformed in the following way:

$$
\begin{aligned}
&T[\![\, f\ p_1^1\ p_1^2 \ldots p_1^n = E_1 \\
&\quad f\ p_2^1\ p_2^2 \ldots p_2^n = E_2 \\
&\quad \ldots\ldots\ldots \\
&\quad f\ p_m^1\ p_m^2 \ldots p_m^n = E_m\ ]\!] \;=\; f = \lambda x_1 \lambda x_2 \ldots \lambda x_n .\ ( \\
&\qquad ((\lambda T[\![p_1^1]\!] \lambda T[\![p_1^2]\!] \ldots \lambda T[\![p_1^n]\!] . T[\![E_1]\!])\ x_1\ x_2 \ldots x_n) \\
&\qquad \square\ ((\lambda T[\![p_2^1]\!] \lambda T[\![p_2^2]\!] \ldots \lambda T[\![p_2^n]\!] . T[\![E_2]\!])\ x_1\ x_2 \ldots x_n) \\
&\qquad \ldots\ldots \\
&\qquad \square\ ((\lambda T[\![p_m^1]\!] \lambda T[\![p_m^2]\!] \ldots \lambda T[\![p_m^n]\!] . T[\![E_m]\!])\ x_1\ x_2 \ldots x_n))
\end{aligned}
$$

where $p_j^i$ are patterns, $x_i$ are new variable names which does not occur free in arbitrary expressions $E_j$, for $i=1,\ldots,n$ and $j=1,\ldots,m$. For example, Fig. 2 displays the effect of described transformation scheme on two simple SASL functions.

```
T[ f x = 2*x ]  =  f = λx.* 2 x
T[ fac 0 = 1
   fac n = n*fac(n-1) ]  =  fac =
                λx.( ((λ0.1)x)
                   □ ((λn.* n (fac (- n 1))) x) )
```

**Fig. 2 Transformation by function T**

### 2.5 Semantics of lambda-calculus enhancements

Semantics of the enhancements to the lambda-calculus will be described by giving the values of applications of lambda abstractions to arguments. The following is the definition of the constant lambda abstraction, for any constant **c** and arbitrary expressions **E** and **a**:

$$
\begin{aligned}
&Val[\![\lambda c.E]\!]\ a = Val[\![E]\!],\ a = Val[\![c]\!] \\
&Val[\![\lambda c.E]\!]\ a = FAIL,\ a \neq Val[\![c]\!]
\end{aligned}
$$

Intuitively, the value of the **λc.E** applied to **a** is in fact the value of **E**, if **a** evaluates to constant **c**. Otherwise, the value of the constant lambda abstraction is **FAIL**. For example, **(λ3.+ 4 5) (+ 1 2)** evaluates to 9, because **(+ 1 2)** evaluates to **3**, while **(λ3.+ 4 5) 2** evaluates to **FAIL**, because **2** does not evaluate to **3**.

The semantics of patterns involving data constructors is defined in the following way, for every constructor **c** and patterns $p_i$, $i=1,\ldots,n$:

$Val[\![\lambda(c\ p_1\ p_2\ \ldots\ p_n).E]\!]\ (c\ a_1\ a_2\ \ldots\ a_n) = Val[\![\lambda p_1 \lambda p_2 \ldots \lambda p_n.E]\!]\ a_1\ a_2 \ldots a_n$
$Val[\![\lambda(c\ p_1\ p_2\ \ldots\ p_n).E]\!]\ (k\ a_1\ a_2\ \ldots\ a_n) = FAIL,\ c \neq k$

Intuitively, if constructors match then individual lambda abstractions will be applied to data components. If constructors do not match, then the value of application of such lambda abstraction is FAIL.

For example, the following are two possible evaluations:

(λ(CONS x y).+ x y) (CONS 2 3) ---> (λxλy.+ x y) 2 3 ---> (λy.+ 2 y) 3 ---> + 2 3 ---> 5
(λ(CONS x y).+ x y) NIL ---> FAIL

Lambda abstractions depending on constructor operators have the same semantics like data constructors -the only difference between the two is syntactical one. The semantics of variable lambda abstractions on variables is the usual semantics of lambda abstraction (see for example [2, 4]).

Operator □ [7] is defined as follows:

x □ y = x
FAIL □ x = x

Intuitively, operator □ forces the evaluation of its first argument. If it evaluates to FAIL, then the evaluation of the second argument is forced.

### 2.6 Possibilities for Implementation

Semantics described in previous section can be implemented either directly like described, or by further transformations of enhanced lambda calculus into the "ordinary" one containing special built-in functions. It is usually the case that above semantics is implemented by its transformation into the family of CASE operators, which inspect the structure of its argument and "jumps" to the appropriate branch.

## 3 SASL AND PATTERNS IN SASL

Functional programming language SASL (short for: *Saint Andrews Static Language*) was the first one whose programs has taken the form of the set of equations and employed pattern matching. Furthermore, SASL was also the first functional language which was with non-strict semantics and implemented by graph reduction. By standards of best known functional language representatives of today, SASL is regarded as untyped language with non-strict semantics, without possibilities to introduce new data types except built-in ones. For more details about SASL and its implementation see [5,6].

Since in SASL there is no introduction of new data types, patterns in SASL are simpler than patterns in general. Pattern in SASL can be:

- a variable or
- a constant or
- an infix constructor operator pattern of the form $p_1 : p_2$, where $p_1$ and $p_2$ are patterns and : is binary constructor operator; a nullary constructor operator ( ). Operator : is an infix shorthand for CONS, while ( ) is a shorthand for NIL.

```
length () = 0
length (a:x) = 1 + length x

fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

**Fig. 3** Two function definitions in SASL

In Fig. 3 are displayed definitions for a function **length** which returns the length of a list and function **fib** which returns nth fibonacci number.

From the implementation point of view:

- patterns in SASL can be overlapping, so that the order of equation is significant (equations are in SASL tried top to bottom).
- set of equations can be non-exhaustive, which means that the implementator have to take care of all uncovered cases in user's program during the compilation of equations.
- repeated variables are not allowed and will be considered as semantic error in user's program.

## 4 TRANSFORMATION OF SASL EQUATIONS

The goal of our SASL implementation is to translate the program in source (SASL) language into some existing language whose language processors are already available. LispKit LISP, Scheme or any other purely functional subset of LISP has been chosen as an target language for translation of SASL source, because of the broad availability of those processors. Since they don't implement semantics described in 2.5 nor does not possess special built-in functions, implementation had to be done using only common operators.

### 4.1 Semantics of a "Common" Primitive Operations

For translation of SASL equations into already existing language, only several "common" operators of the target language are needed: **cond**, **&**, **=**, **list**, **head** and **tail**. In this section, a semantics of those operators will be defined:

For every $b_1, \ldots, b_n$ of logical type, $c_1, \ldots, c_n$, $k_1, \ldots, k_n$ of character type, $x, x_1, x_2, \ldots, x_n, y, y_1$ of any type, where **true** and **false** are logical constants, and **nil** and **:** (list) are data constructors, the following holds:

Val[ **if true then** $x$ **else** $y$ ] = Val[ $x$ ]
Val[ **if false then** $x$ **else** $y$ ] = Val[ $y$ ]

Val[ **cond** $(b_1\ x_1)\ (b_2\ x_2)\ \ldots\ (b_n\ x_n)$ ] =
Val[ **if** $b_1$ **then** $x_1$ **else** (**if** $b_2$ **then** $x_2$ **else** (...(**if** $b_n$ **then** $x_n$ **else** *error*)...)) ]

Val[ $b_1$ **and** $b_2$ ] = Val[ $b_2$ **and** $b_1$ ]
Val[ **false and** $b_1$ ] = **false**
Val[ **true and** $b_1$ ] = Val[ $b_1$ ]

Val[ $b_1$ & $b_2$ & ... & $b_n$ ] = Val[ ($b_1$ **and** ($b_2$ **and** (... **and** ($b_n$ **and true**)))) ]

Val[ **true = true** ] = **true**
Val[ **false = false** ] = **true**
Val[ "$c_1c_2...c_n$" = "$k_1k_2...k_m$" ] = **true**, if n=m and $c_i=k_i$, i=1,...,n
Val[ **nil = nil** ] = **true**
Val[ $(x{:}x_1) = (y{:}y_1)$ ] = Val[ $(x=y)$ & $(x_1=y_1)$ ]
Val[ $x = y$ ] = **false**, otherwise

Val[ **list** $(x{:}x_1)$ ] = **true**
Val[ **list** $x$ ] = **false**, otherwise

Val[ **head** $(x{:}x_1)$ ] = Val[ $x$ ]
Val[ **tail** $(x{:}x_1)$ ] = Val[ $x_1$ ]

Note that in most functional languages there are equivalents of above operations with exactly

the same semantics as described.

### 4.2 Transformation Rules

The following are the rules for transformation of SASL equations into the set of common operators of any functional language:

$$T[\![ f\ p^1\ p^2 \ldots p^n = E ]\!] \quad = \quad f = \lambda x_1 \lambda x_2 \ldots \lambda x_n . E'$$

$$\begin{aligned} T[\![ & f\ p_1^1\ p_1^2 \ldots p_1^n = E_1 \\ & f\ p_2^1\ p_2^2 \ldots p_2^n = E_2 \\ & \ldots\ldots\ldots \\ & f\ p_m^1\ p_m^2 \ldots p_m^n = E_m ]\!] \end{aligned} \quad = \quad f = \lambda x_1 \lambda x_2 \ldots \lambda x_n . (\text{cond } (\text{condition}_1\ T[\![ E'_1 ]\!]) \; (\text{condition}_2\ T[\![ E'_2 ]\!]) \; \ldots\ldots \; (\text{condition}_m\ T[\![ E'_m ]\!]))$$

where $p^j$, $p_i^j$ are patterns, $x_j$ are identifier names which do not occur free in expressions E, $E_i$, while $\text{condition}_i$ and E', $E'_i$, are built by consecutive applying of following rules, for i=1,...,m and j=1,...,n:

1. E' and $E'_i$, i=1,...,m are obtained from E and $E_i$ respectively such that every variable pattern $p_i^k$, $k \in \{1,\ldots,n\}$ is replaced with appropriate $x_k$. In standard notation of lambda-calculus, $E'_i = [x_k/p_i^k]E_i$ for every $k \in \{1,\ldots,n\}$ for which $p_i^k$ is a variable.

2. $\text{condition}_i$, i=1,...,m are built as $(x_k = p_i^k) \& \ldots \& (x_l = p_i^l)$, for every constant pattern $p_i^k \ldots p_i^l$, $k,\ldots,l \in \{1,\ldots,n\}$.

3. $\text{condition}_i$, i=1,...,m are built as $(x_k = \textbf{nil}) \& \ldots \& (x_l = \textbf{nil})$, for every $p_i^k \ldots p_i^l$, $k,\ldots,l \in \{1,\ldots,n\}$ which is equal to ( ).

4. $\text{condition}_i$, i=1,...,m are built as $(\textbf{list}\ x_k) \& \ldots \& (\textbf{list}\ x_l)$, for every $p_i^k \ldots p_i^l$, $k,\ldots,l \in \{1,\ldots,n\}$ which are of the form *x*:*y*. E', $E'_i$ are obtained from E, $E_i$ by replacing all occurrences of *x* by **head** $x_k$ and *y* by **tail** $x_k$, for every k for which $p_i^k$ is of the form *x*:*y*. In standard notation of lambda-calculus, $E'_i = [\textbf{tail}\ x_k/y]([\textbf{head}\ x_k/x]E_i)$ for every k for which $p_i^k$ is of the form *x*:*y*.

5. $\text{condition}_m$, (m > 1) is built as **true** if all $p_m^k$ $k \in \{1,\ldots,n\}$ are variables (in correct programs equation with all variable patterns have to be the last one).

It can be easily formally proved this scheme exactly implements semantics of pattern matching lambda abstractions given in 2.5.

### 4.3 Notices on Implementation

The implementation of transformation rules described in previous section is done using attribute grammars and compiler generator, in which the transformation of SASL equations is the most important part. SASL equations are made of **namelist** (left-hand side) and corresponding **expression** (right-hand side), and both of them have associated attribute. The internal structure of the function definition with respect to **namelist** and **expr** is of the following form (written in Modula-2-like syntax):

```
GroupOfEquations = RECORD      /* function definition */
                     Name: ARRAY OF CHAR;     /* function name */
```

```
                    NoOfEqus: CARDINAL;      /* number of equations */
                    NoOfParms: CARDINAL;     /* number of parameters */

                    NoOfConds: CARDINAL;     /* number of conditions */
                    TrueCond: BOOLEAN;       /* true condition built? */
                    LHS: ARRAY OF Namelist;  /* left hand sides */
                    RHS: ARRAY OF Expr;      /* right hand sides */
                  END;
Namelist = RECORD                    /* structure of the left hand side */
            Parms: ARRAY OF ARRAY OF CHAR;    /* parameters */
            NoOfConsts: CARDINAL;             /* number of constants */
            Consts: ARRAY OF Cst;             /* structure of constants */
            NoOfLists: CARDINAL;              /* number of lists */
            Lists: ARRAY OF Lst;              /* structure of lists */
           END;
Cst = RECORD
       Place: CARDINAL;              /* position of parameter in the
                                     equation that contains constant */
       Value: ARRAY OF CHAR          /* actual value of the constant */
Lst = RECORD
       Place: CARDINAL;              /* position of parameter in the
                                     definition that contains list */
       NoOfMembs: CARDINAL;          /* number of members of the list */
       Membs: ARRAY OF ARRAY OF CHAR;    /* members themselves */
      END;
Expr = RECORD
       Conditions: Internal Structure of Expressions;
       Right-hand: Internal Structure of Expressions;
      END;
```

After all equations are examined and all attributes "filled in", next step is to determine the groups of the equations with the same name. Then, every group is processed by the following procedure:

```
;
WITH GroupOfEquations DO
     FOR i:=1 TO NoOfEqus DO
          Determine new identifiers which do not occur free in function
          definition
          FOR j:=1 TO NoOfParms DO
                    Search RHS[j].Right-hand for LHS[i].Parms[j] and
                    replace it with new identifiers, according to rule 1
                    (of section 4.2)
          END;
          FOR j:=1 TO LHS[i].NoOfConsts DO
                    Build or update RHS[i].Condition according to rule 2.
                    and 3. (of section 4.2), and update NoOfConds
          END;
          FOR j:=1 TO LHS[i].NoOfLists DO
                    Build or update RHS[i].Condition according to rule 4
                    (of section 4.2), and update NoOfConst
                    FOR k:=1 TO LHS[i].NoOfMembs DO
                              Search RHS[i].Right-hand for
                              LHS[i].Lists[j].Membs[k] and replace it
                              by calls of appropriate funcions,
                              according to rule 4 (of section 4.2)
                    END;
          IF ( RHS[i].NoOfConst=0 AND RHS[i].NoOfLists=0 AND
                         i=NoOfEqus ) THEN
                    Build RHS[i].Condition as "true" condition,
                    accoording to rule 5 (of section 4.2), and update
                    TrueCond
          END
     END
END;
IF NoOfConds <> 0 THEN
     IF NOT TrueCond THEN
          Build "true" condition, indicating an error
     END;
```

```
        "Surround" conditions by cond operator
END;
Write transformed expression
```

The weakest point in described implementation is a choice of "strange enough" parameter names - if function definition contains free variable of the same name, semantics of resulting function definition will bi significantly changed. However, "occurs check" could be to expensive to perform during transformation.

### 4.4 Examples

Examples displayed in Fig. 3 are by rules described in previous two sections transformed into two LispKit LISP functions displayed in Fig. 4 (equivalents in other similar languages would look almost the same). In both examples, prefix operator eq is equivalent with (in section 4.1) defined infix =, not atom with list, car with head, cdr with tail, and ('NIL) with nil.

```
(length lambda (newvar1)
  (cond
    ((eq newvar1 ('NIL)) ('0))
    ((not (atom newvar1))
     (add ('1 )
          (length (cdr newvar1)))
    )
    (('T) ('error))
) )

(fib lambda (newvar1)
 (cond
   ((eq newvar1 ('0)) ('1))
   ((eq newvar1 ('1)) ('1))
   (('T) (add (fib (sub newvar1 ('1)))
              (fib (sub newvar1 ('2))))
)) )
```

**Fig. 4 LispKit LISP equivalent of two examples**

## 5 CONCLUSION

The transformation scheme given in this paper is different from those found in literature, because it translates equations into a set of operators which can be found in every functional language. Since no special operators are needed for transformation, languages involving pattern matching can be implemented by translation into some other language, which is broadly available or efficiently implemented. However, this scheme can only be used to translate untyped languages, i.e. languages which does not allow introduction of new data types.

Current implementation of the described transformation can be improved in many ways. For example, the more efficient execution of transformed program can be achieved by grouping and nesting of cond operators such that the number of needed tests is kept minimal.

### References


1. Budimac, Z., Ivanović, M., Putnik, Z. and Tošić, D., *LISP by Examples*, Institute of Mathematics, Novi Sad, 1991. (in Serbian).
2. Field, A.J. and Harrison, P.G., *Functional Programming*, Addison Wesley, 1988.
3. Ivanović, M. and Budimac, Z., *A Definition of an ISWIM-like Language via Scheme*, SIGPLAN Notices, to appear.
4. Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice Hall, New York, 1987.
5. Turner, D.A., *The SASL Language Manual*, University of St. Andrews, 1976.
6. Turner, D.A., *A New Implementation Technique for Applicative Languages*, Softw. Pract. Exper. 9(1979), 31-49.
7. Turner, D.A., *Aspects of the Implementation of Programming Languages*, Ph.D. Thesis, University of Oxford, 1981.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 93 - 98

# Modification of the Prolog-control in a Smalltalk environment

*Radivoj Protić* *Dušan Tošić*

*Matematički fakultet*
*Studentski trg 16*
*11000 Beograd*
*Yugoslavia*

**Abstract.** The problem of efficient writing the Prolog programs in the Smalltalk environment is considered. Starting with the famous Kowalski's equation:

A(lgorithm) = L(ogic) + C(ontrol)

the component C is analyzed and according to that analysis some new predicates are proposed to built into Prolog/V. These predicates might be useful in any Prolog language where the component Control is significant in solving some type of problems.

**Keywords:** Prolog, Smalltalk, Control, Built-in predicates, Environment.

## 1. Introduction

In this paper we suggest how to improve the efficiency of Prolog programs incorporated into a Object Oriented environment. The problem is how to amalgamate successfully two different programming paradigms: Logic and Object Oriented. The similar problems are studied in [5], but instead of Smalltalk, C++ is considered. In [1] and [2] the problem integration of imperative and logic paradigms is studied. So, the problem of integrating two different paradigms is very actual. The good starting point is Smalltalk/V environment that includes a Prolog interpreter (Prolog/V). We can conclude that a kind of integration of Object-Oriented and Logic paradigm is realised in this environment. Moreover, Prolog/V-interpreter is a simple interpreter written in Smalltalk/V which power is based on the using of Smalltalk-expressions. In a lot of applications it is suitable to have better control-mechanisms in Prolog itself. The connection between Prolog and Smalltalk gives opportunity to solve previous problem in the efficient way.

## 2. How to enhance the control in Prolog

In foreword of [3] Robinson writes: "Today Logic programming is a standard paradigm in the methodology of computing. Its attractions are immediate. Kowalski's apothegmatic equation:

A(lgorithm) = L(ogic) + C(ontrol)

sums up the most striking of them: the clean separation of the knowledge required to solve a problem from the way this knowledge is to be deployed to solve it." According to [4] it is central to the idea of logic programming, and Prolog in particular, that we be prepared to alter the declarative component L to obtain desired problem-solving

behaviour A. The component C is usually fixed in Prolog. Moreover, in same article, it is admitted the possibility "of changing A by changing only C" with the conclusion "Logic programming is concerned with the possibility of changing both L and C."

In an Object-Oriented environment, with Prolog available for the changing, it is very interesting to modify C-component in Kowalski's equation. Because of that we analyze this component in more details. The main elements of the control in Prolog are:

- Backtracking
- Unification and
- Built-in predicates.

The interpreter of Prolog/V is written in Smalltalk/V and we could change all of these elements of the control. Moreover, our attention is on built-in predicates. We propose some new logical predicates enhancing the control in Prolog/V. These predicates may be added to any Prolog working environment.

## 3. Prolog-classes in Smalltalk/V

To explain how to enhance Prolog/V, it is necessary to inspect the place of Prolog/V in Smalltalk environment. Prolog/V interpreter is highly integrated into Smalltalk/V hierarchy. The structure of classes related to Prolog/V is:

```
Object          <-- Top of hierarchy
.....
Logic           <-- Key control mechanism
  Prolog        <-- Basic predicates
    .           <-- Application classes
    .            (with predicates
    .             written in Prolog/V)
```

All methods in classes Logic and Prolog are written in Smalltalk/V and could be changed. If we intend to add new predicates, their methods should be written into the class Prolog.

## 4. New control-predicates in Prolog/V

In this article our attention is concentrated to the generalization and creation of logical predicates. These predicates may be useful in changing of the control of Prolog programs.

The initial idea for the building new control-predicates was born after modification of the predicate or. Namely, the existing definition of the predicate or in Prolog/V is (see [6]):

```
or: assoc
   "Disjunction predicate."
   assoc key size = 2 ifFalse: [^self].
   self doGoal: (self first: assoc) continue: assoc value.
   self doGoal: (self second: assoc) continue: assoc value
```

This is standard definition of or-predicate. In Prolog notation it is:

```
or(x, y) :- x.
or(x, y) :- y.
```

Sometimes it is useful to make the following modification of previous definition:

```
or(x, y) :- x, !.
or(x, y) :- y.
```

In Smalltalk/V code it is enough to change the line:

```
self doGoal: (self first: assoc) continue: assoc value.
```

into the line:

```
self doGoal: (self first: assoc) continue: [^assoc value].
```

A generalisation of the modified or-predicate (we call it gor) in Prolog description could be:

$gor(p_1, p_2,\ldots,p_k,\ldots,p_m)$ :- $p_1$.
.....
$gor(p_1, p_2,\ldots,p_k,\ldots,p_m)$ :- $p_k$, !.
.....
$gor(p_1, p_2,\ldots,p_k,\ldots,p_m)$ :- $p_m$.

This is clumsy construction and may be changed by a built-in predicate:

$gor(k, p_1, p_2,\ldots,p_k,\ldots,p_m)$

by the following Smalltalk code:

```
gor: assoc
    "General disjunction predicate."
    | m n aList i |
(m := assoc key size ) <= 2 ifTrue: [^self].
aList := assoc key.
n := aList head.
aList := aList tail.
i := 0.
n < 1 ifTrue: [^self].
[ aList isEmpty ]
    whileFalse: [
        i := i + 1.
        i = n
          ifTrue: [
            self doGoal: (aList head) continue: [^assoc value]
                ]
          ifFalse: [
            self doGoal: (aList head) continue: assoc value
                ].
          aList := aList tail
        ]
```

Sometimes we put the cut-predicate in several positions of clauses as in the following example:

```
gorl(p1,p2,p3,p4,p5) :- p1.
gorl(p1,p2,p3,p4,p5) :- p2,!.
gorl(p1,p2,p3,p4,p5) :- p3.
gorl(p1,p2,p3,p4,p5) :- p4,!.
gorl(p1,p2,p3,p4,p5) :- p5.
```

Practically, we need a generalized gor-predicate, that could be called gorl. In this case we should know a list of positions of the cut predicates. For the previous example we could write it in the following way:

```
gorl([2,4],p1,p2,p3,p4,p5).
```

The general form for the gorl-predicate is:

```
gorl(List,p1,p2,...,pm).
```

where List contains positions of the cut-predicates.

The gorl-predicate is implemented in Smalltalk by slightly changing of the Smalltalk code for the gor-predicate as follows:

```
gorl: assoc
    "General disjunction predicate with List."
    | m n aList i tmpList |
(m := assoc key size ) <= 2 ifTrue: [^self].
aList := assoc key.
tmpList := aList head.
( tmpList isKindOf: List ) ifFalse: [ self error: ' not a list ! ' ].
aList := aList tail.
i := 0.
[ aList isEmpty ]
   whileFalse: [
      i := i + 1.
      (tmpList hasObject: i )
       ifTrue: [
         self doGoal: (aList head) continue: [^assoc value]
              ]
       ifFalse: [
         self doGoal: (aList head) continue: assoc value
              ].
       aList := aList tail
     ]
```

In the following example we present a part of the Prolog program with the gorl-predicate:

```
.....
form(List),
.....
gorl(List, feature_1(X), feature_2(X),...,feature_m(X)),
checkProfit(X),
.....
```

An interpretation of this part of Prolog code might be the following one. List contains the positions of very important features of some material. If a material with at least one very important feature is not profitable, the searching for the rest materials with other features is stopped.

In the similar way we may generalize and-predicate. In Prolog/V, and-predicate is realised by the following Smalltalk method ([6]):

```
and: assoc
    "Try to satisfy a list of goals."
    | aList |
    aList := assoc key.
    [aList isEmpty] whileFalse: [
        (self doOneGoal: aList head)
            ifTrue: [aList := aList tail]
            ifFalse: [^self]].
    ^assoc value
;
```

The corresponding Prolog notation of the and-predicate is:

$$and(p_1, p_2,...,p_m) :- p_1, p_2,...,p_m, !.$$

We suggest more general and-predicate by the following Prolog (pseudo)definition:

$$gand(k,n,p_1,...,p_k,...,p_n,...,p_m) :- p_k,...,p_n,!.$$

where is:

$$1 \leq k \leq n \leq m.$$

We call this predicate from a Prolog program by:

$$gand(k, n, p_1,... p_m).$$

It is clear that for k=1 and n=m we have and-predicate.
This generalized and-predicate is realised by the following Smalltalk-method:

```
gand: assoc
    "Try to satisfy a list of goals from position m to n "
    | m n aList i |
    aList := assoc key.
    m := aList head.
    aList := aList tail.
    n := aList head.
    aList := aList tail.
```

```
m <= n ifFalse: [ ^self ].
i := 1.
[ aList isEmpty not and: [ i < m ]]
    whileTrue: [ i := i+1.
                 aList := aList tail
               ].
(aList isEmpty )
  ifFalse: [
             [ aList isEmpty not and: [ i <= n ]]
               whileTrue: [
                   (self doOneGoal: aList head)
                     ifTrue: [ aList := aList tail.
                               i := i+1]
                     ifFalse: [^self]].
           ]
  ifTrue: [ ^self ].
^assoc value
```

## 5. Conclusion

This paper has touched upon a number new predicates related to the control in Prolog. It is unproductive to create these predicates in a Prolog application. An acceptable solution is the implementation of these predicates in Smalltalk. The specific organisation of Smalltalk provides the inheritance of those predicates from the others Prolog applications. The usefulness of integration of two programming paradigms (Logic and Object-Oriented) is evident.

## 6. References

[1] Boyd J.L. and Karam G.M. *PROLOG in 'C'*, ACM SIGPLAN Notice, Vol. 25, No. 7, 1990, 63-71.

[2] Delrieux C, Azero P. and Tohme F. *Toward Integrating Imperative and Logic Programming Paradigms: a WYSIWYG approach to PROLOG Programming,* ACM SIGPLAN Notice, Vol. 26, No. 3, 1991, 35-44.

[3] Deville Y. *Logic Programming, Systematic Program Development,* Addison-Wesley Pub. Company, 1990.

[4] Kowalski R. A. *The Early Years of Logic Programming,* Comm. of ACM, Vol. 31, No. 1, 1988, 38-43.

[5] Shawm-inn We *Integrating Logic and Object-Oriented Programming,* ACM OOPS Messenger, Vol. 2, No. 1, 1991, 28-37.

[6] *Smalltalk/V Tutorial and Programming Handbook,* Digitalk inc., 1986.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 99 - 106

# DEPENDENCY ANALYSIS IN AN UNTYPED FUNCTIONAL LANGUAGE: AN EXERCISE IN FUNCTIONAL PROGRAMMING*

Zoran Putnik, Zoran Budimac, Mirjana Ivanović

University of Novi Sad, Faculty of Natural Sciences and Mathematics
Institute of Mathematics, Trg D. Obradovića 4, 21000 Novi Sad

e-mail: {putnik,zjb,ivanovic}%unsim@yubgef51.bitnet

ABSTRACT: A possible solution of a dependency analysis problem is given by this paper. It can be used as a useful part of most optimization schemes used in implementation of functional programming languages. Implementation is given in ISWIM - an untyped functional programming language, using higher-order functions, leading to a short and abstract solution. It can be used as a description of a dependency analysis algorithm and as a prototype to a more efficient implementation.

**Key-words:** Functional Programming, Dependency Analysis, Higher-Order Functions, Strongly Connected Components of Graph

## 1 INTRODUCTION

Functional programming represents one of the most important programming paradigms, next to procedural, object-oriented, logic etc. Together with logic programming style, functional programming is considered as declarative, by which the problem is described (i.e. *what* is to be solved), and not the recipe for its solution (i.e. *how* the problem is to be solved).

It is usually estimated that functional programming will have the most important role in the future of computing and solving of the so-called "software crisis." This claim is based on the following three characteristics of functional programs:

- Functional programs are short, concise, and easier to read and maintain than their counterparts in any other programming paradigm (including logic).
- Features and (in)correctness of functional programs can be formally proved.
- Functional programs can be naturally and easily implemented on parallel architectures, without introduction of any additional language constructions and concepts.

Functional paradigm has the mentioned three characteristics because of its powerful fundamentals in mathematics (lambda-calculus). Because of that, functional programs do not have statements (especially assignment statements), side-effects and explicit sequencing. Functional programs are characterized also by static binding of identifiers and treatment of functions as "first class citizens." Besides that, some functional programming languages have a possibility of the so-called lazy evaluation, which enables a dealing with infinite data structures.

Currently there are many research directions in the field of functional programming. Among the most interesting and the most proliferable is the application of functional programming style in

* This research is supported by Science Fund of Serbia

different fields of "every-day" programming. An aim of this direction is to prove practically the usefulness of functional programming paradigm and its advantages. For example, the small part of investigations and implemented software systems based on functional programming paradigm is: general remarks about tools and "procedures" applicable in many areas [5, 6, 9], purely functional operating systems [7, 12], implementation of other programming paradigms [4], implementation of some mathematical algorithms [11, 18, 19], classical software engineering tools and methods [15, 16] etc.

This paper presents a solution of a dependency analysis that is a useful part of most optimization schemes used in implementation of functional programming languages. The analysis is implemented in ISWIM - an untyped functional programming language, using higher-order functions, which directly lead to a short and abstract solution, (mostly) independent of concrete data structures. Described implementation can be used as a description of a dependency analysis, as well as an executable specification of an algorithm. If the implementation of ISWIM is not efficient enough, then the presented solution at least can be used as a prototype for more efficient implementations.

The rest of the paper is organized as follows: section 2 contains a short introduction to a functional programming language ISWIM, while section 3 contains a description of importance and a verbal description of a common algorithm for dependency analysis. Section 4 in full details describes ISWIM implementation of an analysis. Section 5 concludes the paper.

## 2 SHORT COMMENTS ON ISWIM

ISWIM (short for: If you See What I Mean) was intended to be a set of purely functional languages with common basis. It was introduced in [13] and represented an important step in the development of functional languages. Perhaps the two most important improvements were: the strong influence of lambda-calculus to the design and features of the language and the implementation technique via (virtual) SECD machine [14]. It can be said that ISWIM was the first declarative language and a predecessor of the so-called modern functional languages whose best known representatives today are Miranda (trademark of Research Software Ltd.) and Haskell [8].

From the point of view of LISP dialects (the only language with similar characteristics at the time), ISWIM introduced infix operators, parentheses-free syntax, and local definitions in the form of let or where blocks.

ISWIM is a functional language that satisfies all major criteria for a functional language [2,8]: it lacks statements, explicit sequencing and side effects, treats functions as first-class citizens and is lexically scoped (i.e. statically binds its identifiers). Moreover, ISWIM is:

- the language with strict semantics, which means that it is incapable for lazy evaluation (although that can be easily changed).
- the untyped language, which means that it has not a notion of type, type checking nor the introduction of user-defined types in any way.
- the language in which the functions returning functions are defined by "anonymous" functions or explicit definitions of curried functions.

```
//insertion sort of list l
{ sort
    where rec
       sort(l) = l=nil -> nil;
                 insert(hd l, sort(tl l))
   and insert(a,l) = l=nil -> [a];
                     a<=hd l -> a:l;
                     hd l:insert(a,tl l)
}
```

Fig. 1 An example of ISWIM program

- the language that does not posses the so-called special syntax enhancements such as guards, list comprehension and pattern-matching. In ISWIM all expressions must be defined explicitly, using primitive operators of the language.

Fig. 1 displays an illustrative example of ISWIM programming - an insertion sort of list 1. An extended version of an ISWIM language is implemented at the Institute of Mathematics in Novi Sad [3, 10]. This version conforms to the major ideas of Landin's original language. The main feature of ISWIM that will be explored in this paper is its lack of typing, i.e. impossibility to introduce new data types. Instead of data types, higher order functions, in some cases, will be used.

## 3 DEPENDENCY ANALYSIS

The dependency analysis takes place when a set of identifiers is bound by a set of (potentially) mutually recursive expressions in functional programs. Such constructions are often large and can be rearranged in smaller and nested sets such reflecting the dependency among defined identifiers. Moreover, there are often definitions that are non-recursive and need not be defined in the same set with mutually recursive ones. The purpose of dependency analysis is to rearrange the set of definitions such that: i) the set of mutually recursive definitions is minimal and as nested as possible and ii) to separate the sets of mutually recursive definitions from non-recursive ones.

If dependency analysis is not performed, functional program is inefficiently executed and in some cases impossible to type check (if functional language is strongly typed).

### 3.1 LETREC Blocks

The set of definitions in many functional languages and intermediate forms is introduced by the so-called **let** or **letrec** blocks. Both blocks are of the same (syntactic) form and can be represented as in Fig. 2. **Let(rec)** blocks replace expressions $def_i$ by the identifiers $x_i$, i=1,...,n in the expression *exp*. The only difference between **let** and **letrec** blocks is in the scope of introduced identifiers: in **letrec** blocks identifiers $x_i$ can occur in *exp* as well as in $def_i$, while in the case of **let** expressions identifiers $x_i$ can occur only in *exp*.

```
let[rec]
  x1 = def1
  x2 = def2
    ....
  xn = defn
in exp
```

**Fig. 2** Let[rec] block

According to mentioned rules, **letrec** block is a suitable mechanism for defining (mutually) recursive definitions, while **let** block is appropriate for "ordinary" definitions. **Let** blocks can be (a much) more efficiently implemented. **Let** and **letrec** blocks (or their equivalents) are common for most functional languages and represent the way of expressing recursion and/or term sharing.

### 3.2 An Algorithm for Dependency Analysis

A starting point for dependency analysis is the set of definitions defined in a **letrec** block. Generally, analysis proceeds as follows (for more details see, for example [17]):

1. For each **letrec** block construct a directed graph (called dependency graph) where the nodes are identifiers bound in **letrec** block. Node $x$ is connected to node $y$ if $y$ occurs free in the definition of $x$. Identifiers $x$ and $y$ (i.e. their definitions) are mutually recursive if there is a two-way path between corresponding nodes in the graph (all nodes with such feature belong to the so-called strongly connected component of a graph, later on: SCC).
2. Find all SCCs of the dependency graph [1].

3. Sort SCCs of the dependency graph into dependency order. This can be achieved in step 2 by choosing appropriate algorithm for finding SCCs or by coalescing all SCCs into single nodes and then by performing topological sort of such coalesced graph.
4. Produce a new arrangement of definitions, based on sorted coalesced graph. All singleton components of a graph not pointing to themselves can be embodied into let blocks, while all the other nodes can be transformed into letrec blocks. The order of let and letrec blocks depends on sorted coalesced graph.

### 3.3 An Example

The work of dependency analysis will be illustrated on the following (although very simple) example. Let the four identifiers (multi, T1, x and T2) are bound by the following letrec block (in which all definitions are written as they would be in ISWIM). Note that function arguments are local to the particular definitions and are not considered during analysis:

```
letrec  multi(n) = x*n
        T1(n) = n>1000 -> true; T2(multi(n))
        x = 10
        T2(n) = n>1000 -> false; T1(n+1) in T1(80)
```

The dependency graph of the block is displayed in Fig. 3.a and topologically sorted coalesced graph is displayed in Fig. 3.b. Based on the latter graph, the following rearrangement of original definitions is produced:

```
let     x = 10 in
let     multi(n) = x*n in
letrec T1(n) = n>1000 -> true; T2(multi(n))
       T2(n) = n>1000 -> false; T1(n+1) in T1(80)
```

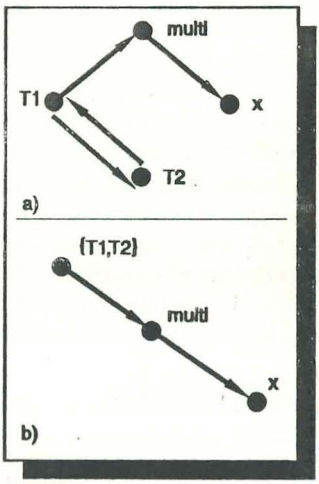



Fig. 3 Dependency and coalesced graph

Resulting definitions can be (considerably more) efficiently executed and are appropriate for some type checking algorithms. They are also much easier to read.

## 4 AN IMPLEMENTATION IN ISWIM

In the following sections an implementation of dependency analysis in ISWIM is described. The only feature of ISWIM essential to implementation is that it is untyped language. Only untyped languages can accept general lists as their arguments, and they usually "force" the programmer to use higher-order functions instead of introduction of new data types.

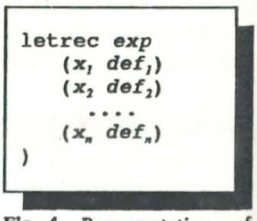



Fig. 4 Representation of let(rec) block

### 4.1 Graph Representation

Let and letrec blocks will be represented as list as displayed in Fig. 4, where *exp* is also a list and $def_i$, i=1,...,n consist of sequences of elements which can also be lists. List is most suitable data structure for letrec blocks, because in practice in many languages and intermediate forms let and letrec blocks are already represented as lists (various dialects of Lisp, Scheme, etc...). That way, no special preparation for let(rec) blocks is needed prior to analysis. The language which could recognize

general lists as its data structure has to be untyped, because any strongly typed language couldn't cope with lists containing numbers, strings and other lists in the same time.

Graph will be represented simply by function **g(x,y)**, returning **true** if there is an arc from x to y (later on: x --> y) and **false** otherwise. List of lists could also have been chosen, but solution like this would be clumsy and not in the spirit of functional programming style.

Fig. 5 displays three functions for dealing with graphs. Function empty represents a graph with no arcs because for every two nodes it returns **false**. Function **link** "puts" arc **i --> j** in graph (function) **g**, such that it returns a new graph (function) of two arguments **x** and **y** which to the old function **g** adds a new case which is to be tested.

```
empty(x,y) = false
link(i,j,g)(x,y) =
  (i=x)&(j=y) -> true; g(x,y)
unlink(i,j,g)(x,y) =
  (i=x)&(j=y) -> false; g(x,y)
```

**Fig. 5** Functions dealing with graphs

Similarly, function **unlink** removes the arc **i --> j** from graph **g**. Note that functions **link** and **unlink** are higher-order functions which take function **g** as their argument and return function as their result. Functions returning functions are in this case defined as curried functions which is in ISWIM done by writing two sets of arguments **((i,j,g)** and (x,y)) next to the function identifier.

Using these three functions any oriented graph can be constructed. For example the following ISWIM definition of function **gr** designates the function (graph) which contains arcs 1 --> 2 and 2 --> 3: `gr(x,y) = link(1,2,link(2,3,init))(x,y)`. Note also that in some implementations of functional languages (especially lazy ones) arguments **(x,y)** need not to be quoted on either side of such a definition.

One of possible versions of a function which would construct graph **g(x,y)**, from a **letrec** block is given in Fig. 6. The function is called with the following parameters: program (in the form of a list and containing **letrec** block), list of names (identifiers) defined in **letrec** block, same list again (for more convenient function definition) and the function **empty** (i.e. empty graph). The result of a function is constructed graph.

```
g(block, names, copy, graph) =
 block=nil -> graph(x, y);
 names=nil -> g(tl block, copy, copy, graph)(x, y);
 memball(hd name, tl hd block) ->
    g(block, tl name, copy,
      link(hd hd block, hd name, graph))(x, y);
 g(block, tl name, copy, graph)(x, y)
```

**Fig. 6** Function for creating a graph

According to a dependency analysis algorithm, a function for construction of a reversed graph is needed. As a reversed graph of graph **g**, we consider the graph in which **x --> y**, if **y --> x** in graph **g**. In ISWIM, that function can be for example one given in a Fig. 7. Basically, this function uses two copies of an original graph, tests whether two nodes were originally linked, and if they were, in a second graph unlinks first link and then links reversed one. Parameters of the function `revg` are list of names (three

```
revg(nam1, nam2, nam3, graph, revgraph) =
  nam1 = nil -> revgraph;
  nam2 = nil ->
    revg(tl nam1, nam3, nam3, graph, revgraph);
  graph(hd name, hd nam2) ->
    revg(nam1, tl nam2, nam3, graph,
         link(hd nam2, hd name,
              unlink(hd name, hd nam2, revgraph)));
  revg(nam1, tl nam2, nam3, graph, revgraph)
```

**Fig. 7** Function for creating a reversed graph

times) and graph (two times). Third copy of a list of names is needed to initialize recursive call of **revg** when **nam2** becomes **nil**.

### 4.2 Steps of Algorithm

Function dfs (Fig. 8) performs depth-first search on a graph finding all paths between graph nodes and create sets of connected components. **Dfs** then finds the "order of importance" of those sets (for more details see [1]). Function is called with two parameters, list of all defined names and higher-order function **graph**. After counting number of connected components for each node and sorting them in descending order, as a result we gain a sorted, associative, list of all nodes with "number of importance" assigned to it.

```
dfs(name, graph) =
  inssort(count(find(name, graph, name)))
// Finds all paths from nodes given in name
and find(name, graph, copy) =
  name = nil -> nil;
  reverse(remove(search(hd name, copy,
                  nil, graph))) :
  find(tl name, graph, copy)
// Finds a path for a single node
and search(vertex, names, path, graph) =
  names = nil -> vertex : path;
  graph(vertex, hd names) ->
    search(hd names, names,
         vertex : path, graph) ++
    search(vertex, tl names, path, graph);
  search(vertex, tl names, path, graph)
```

**Fig. 8** Function for performing depth-first search

Function **scc** (Fig. 9) reverse the original graph and performs depth-first search of reversed graph according to sorted sets from previous step. Only parameter used is **letrec** block, but "middle" functions use list of names and graph (created from **letrec** block). As a result, list of sets of SCCs is gained.

**Function move** rearranges original **letrec** block according to SCCs found in previous step of algorithm. Function changes **letrec** block into **let** blocks (when allowed) and changes an order of **letrec** blocks according to sets of SCCs. Together with this function, functions for finding a definition which is to be rearranged and function for removing that definition from a rest of a **letrec** block are used.

```
scc(block) =
  mid1(names(tl tl block), block,
     g(tl tl block, names(tl tl block),
       names(tl tl block), empty))
// Finds sets of SCC's
and mid1(name, block, graph) =
  mid(sort(name, dfs(name, graph), nil, nil),
      block, graph)
// Finds path for nodes of a reversed graph
and mid(name, prog, graph) =
  remove(find(name,
         revgr(name, name, name, graph, graph),
           name))
```

**Fig. 9** Function for finding SCC's

That part of a program is given in Fig. 10. Function **move** is called with the following parameters: **letrec** block, set of SCCs, expression of a **letrec** block and constant **true** or **false**. Last parameter is needed for deciding when to put that expression into resulting block.

### 4.3 The Program and Additional Functions

The functions defined above are organized in a function **depan**, which is called in the following way:

```
depan(block) = move(tl tl block, scc(block),
                    [hd block, hd tl block], true)
```

In the whole program several additional functions are used. There can be easily implemented and we only enumerate their names and meaning: function **names(block)** for a given **letrec** block returns the list of all defined identifiers in that block; function **memball(x,l)** returns **true** if **x** is element of **l** and all its sublists, otherwise returns **false**; function **reverse(l)** returns reversed list **l**; function **inssort(l)** sorts a list **l** into an ascending order; function **remove(l)** removes multiple occurrences of elements of a list **l**; function **count(l)** counts number of elements for each list in a list of lists **l**; function **sort(name, count(l), aux, newl)** which, using list of names and **count(l)**, creates **newl** - sorted list of SCCs.

```
move(block, scc, exp, needed) =
 scc = nil -> nil;
 #(hd scc) = 1 ->
   ["let",
   move(remdef(hd scc, block, nil),
          tl scc, exp, needed),
   (hd hd finddef(hd scc, block, nil) :
   4!(hd finddef(hd scc, block, nil)))];
 needed -> exp ++
         finddef(hd scc, block, nil) ++
           move(remdef(hd scc, block, nil),
           tl scc, title, false);
 finddef(hd scc, block, nil) ++
 move(remdef(hd scc, block, nil),
      tl scc, exp, needed)
// Finds a definition in a letrec block
and finddef(def, block, res) =
 block = nil -> res;
 hd hd block in def ->
   finddef(def, tl block, hd block : res);
 finddef(def, tl block, res)
// Removes a definition from a letrec block
and remdef(def, block, res) =
 block = nil -> res;
 hd hd block in def ->
   remdef(def, tl block, res);
 remfun(def, tl block, hd block : res)
```

Fig. 10 Function for rearranging letrec blocks

## 5 CONCLUSION

Higher-order functions, proved to be an "elegant" and abstract way of solving some of the problems which may arose during dependency analysis. However, in this analysis some data structures could not be replaced by higher-order functions. For example, some intermediate results in finding SCCs of a graph cannot be (simply) represented via functions.

Program written in ISWIM (and described here) will be used as an prototype for an implementation in procedural language. It will constitute the part of a compiler of functional languages currently developing at the Institute of Mathematics. It will also be used for improving an algorithm of dependency analysis given in this paper.

## References


1. Aho, A., Hopcroft, J., Ullman, J., *Data Structures and Algorithms*, Addison-Wesley (1985)
2. Budimac, Z., Ivanović, M., Putnik, Z. and Tošić, D., *LISP by Examples*, Institute of Mathematics, University of Novi Sad, Novi Sad, 1991. (in Serbian).
3. Budimac, Z., Ivanović, M. and Živkov, S., *Strict ISWIM Compiler, version Nui/SECD*, Authoring Agency for Serbia, Copyright no. S-63/92, July 1992.

4. Carlsson, P., *On Implementing PROLOG in Functional Programming*, New Generation Computing, 2(1984), 347-359.
5. Fairbairn, J., *Making Form Follow Function: An Exercise in Functional Programming Style*, Software - Practice and Experience, 6(1987), 379-386.
6. Grimley, A., *Natural Constructions in Functional Programming*, UKC Computing Laboratory Report No. 53, University of Kent, Canterbury, 1988.
7. Henderson, P., *Purely Functional Operating Systems*, in "Functional Programming and its Applications" (D.A.Turner, ed.) Cambridge University Press, 177-192, 1985.
8. Hudak, P., *Conception, Evolution, and Application of Functional Programming Languages*. ACM Comp. Surveys 3(1989), 359-411.
9. Hughes, R.M.J., *Why Functional Programming Matters*, The Computer Journal, 2(1989), 98-107.
10. Ivanović, M., Budimac, Z., *A Definition of an ISWIM-like Language via Scheme*, SIGPLAN Notices, (to appear)
11. Jones, G., *Deriving the Fast Fourier Algorithm by Calculation*, in "Functional Programming: Proceedings of the 1989 Glasgow Workshop" (J. Hughes, ed.), Springer-Verlag, London, 1990, 80-102.
12. Jones, S.B. and Sinclair, A.F., *Functional Programming and Operating Systems*, The Computer Journal, 2(1989), 162-174.
13. Landin, P.J., *The Next 700 Programming Languages*. Comm. of ACM 3(1966), 137 - 164.
14. Landin, P.J., *The Mechanical Evaluation of Expressions*. Comp. Journal 6(1964), 308 - 320.
15. Putnik, Z., Budimac, Z. and Ivanović, M., *Some Improvements of a LispKit LISP Syntax Analyzer*, Proc. of I International DECSYM '92 Symposium, Side-Antalia, Turkey, 243-252, 1992.
16. Peyton Jones, S.L., *YACC in SASL - an Exercise in Functional Programming*, Software - Practice and Experience, 8(1985), 807-820.
17. Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Addison-Wesley, New York, 1987.
18. de Vries, F.-J., *A Functional Program for the Fast Fourier Transform*, ACM SIGPLAN Notices, 1(1988), 67-74.
19. Wainwright, R.L. and Sexton, M.E., *A Study of Sparse Matrix Representations for Solving Linear Systems in a Functional Language*, J.of Func.Prog. 1(1992), 61-73.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 107 - 114

# CONSTRUCTION OF THE TRANSLATOR FROM ROBOTIC PROGRAMMING LANGUAGES

Miloš Racković[1], Institute of matematics
Trg Dositeja Obradovića 4, Novi Sad, Yugoslavia

Keywords:Robotic programming language, translator, compiler-compiler.

Abstract:A new robotic language comprising some basic structures of the robotic language Pasro and some commands of the modified robotic language of the robot ROBED-01 is described. Construction and methodology of implementation of a translator from this new language into the modified robotic language of the ROBED-01 are discussed.

## 1. INTRODUCTION

One of attractive research fields in robotics is the robot programming. A number of compilers for robot-oriented programming languages have been constructed, viz. SRL, PASRO, AL, AML, VAL, etc. [1]. The methodology of compiler construction used for general-purpose programming languages can also be applied for these languages.

In [2-4], construction and the implementation methodology of a translator for robotic languages with the aid of the compiler-compiler Coco-2 [5] have been described. This translator has been intended for translating the basic structure of the robotic programming language Pasro [6] into the robotic language of the ROBED-01 robot. This paper is concerned with a new robotic language (NRL), composed of basic commands of the programming language Pasro and of some commands of the modified robotic language of the robot ROBED-01, invented in Institute Mihajlo Pupin in Belgrade (RLMP). Basic methodology of construction and implementation of a translator from NRL into RLMP is discussed.

## 2. NEW ROBOTIC LANGUAGE

In NRL are adopted some basic components of the robotic programming language Pasro. The syntax structure of NRL is similar to thet of Pasro and the basic commands of strctured programming (while, if then else,...) are adopted. Also it is adopted the concept for the robot operation (motion commands, frame concept,...).

---

[1]This paper is supported with Serbian Fondation of Science

However, because of the limitation of the RLMP, the complete structure of Pasro has not been adopted.

In order the new language would have all the advanteges of RLMP, some commands from RLMP which are not supported in Pasro language were also built-in in NRL. The syntax of these commands has been adapted to the syntax of the NRL language.

A detailed description of the NRL structure via EBNF recording, which is a part of the input file for Coco-2, will be given in Section 3.

## 3. CONSTRUCTION OF THE TRANSLATOR

In Fig. 1 is presented a part of the input file for Coco-2, describing the syntax structure of the NRL, as well as the semantic actions which translate the NRL structures into the corresponding structures of RLMP (after the official word SEM). These actions are realized by the procedures written in programming language Modula-2 [4].

```
RULES
  Prog    =  LOCAL <<VAR spix:INTEGER;>>  SEM <<InitDat;>>
   "program" ident <<spix>> ";"  Block "." SEM <<CloseDat;>>.
  Block   =  LOCAL <<VAR val,spix,lab:INTEGER;>>
      [ "var"  SEM << InitVar;>>  Variables { Variables } ]
      { "procedure" ident <<spix>> ";" SEM <<NewProc(spix,lab);>>
        "begin" Statement { ";" Statement } "end" ";"
            SEM <<CloseProc(lab);>> }
     "begin" Statement { ";" Statement } "end".
  Variables  = LOCAL <<VAR spix,ind,tyval,val,val1:INTEGER;
                     VAR defvar:ARRAY [1..100] OF INTEGER;>>
   ident <<spix>> SEM <<ind:=1;
                        NewVar(spix,defvar,ind);
                        val:=0;
                        val1:=0;>>
   {"," ident <<spix>> SEM <<ind:=ind+1;
                              NewVar(spix,defvar,ind);>> }
      ":" Type <<tyval,val,val1>>
              SEM <<SetType(defvar,ind,tyval,val,val1);>> ";".
 Statement = LOCAL <<VAR ind,spix,val,hlab,hlab1,stval:INTEGER;
             VAR markvar,markvar1,inval,zero,tyval1:INTEGER;
             VAR reval:REAL;
             VAR indic:BOOLEAN;
             VAR arrspi,arrpar,arricon,arrval,arrtyval:ARRAY
                 [1..10] OF INTEGER;
             VAR arrcon:ARRAY [1..10] OF REAL;
             VAR tx:ARRAY [0..100] OF CHAR;>>
 [ Variable <<spix>>   SEM <<val:=0; indic:=FALSE;>>
  [ "[" UnsignedInteger <<val>> "]" SEM <<CheckArr(spix,val);
                                          indic:=TRUE;>> ]
  ":=" Expression <<tyval,inval>> SEM <<CheckIndic(indic,spix);
                                  SetVal3(spix,tyval,inval,val);>>
 | RobotVar <<tyval1>> "[" UnsignedInteger <<val>> "]"
```

```
  SEM <<CheckVal(val);>>  ":=" Expression <<tyval,inval>>
      SEM <<SetValJoint(tyval,inval,val,tyval1);>>
| "speedfactor" ":=" UnsignedNumber <<tyval,inval,reval>>
      SEM <<SpeedCheck(tyval,reval);
            SpeedTrans(reval);>>
| ProcedureIdentifier <<tyval,spix>> SEM <<ind:=0;
                                          indic:=FALSE;>>
  [ ActualParameterList
     <<arrspi,arrpar,arricon,arrtyval,arrcon,ind>>
  SEM <<indic:=TRUE;>> ]  SEM <<IF tyval <> 0 THEN
    CheckParam(tyval,ind,arrspi,arrpar);
    TransProc(tyval,ind,arrspi,arrpar,arricon,arrtyval,arrcon)
  ELSE
    CheckParProc(indic,tyval);
    CallProc(spix)
  END;>>
| "write"   "("  SEM <<FOR ind := 0 TO 100 DO
                       tx[ind]:=' '
                     END;>>
    (  string <<tx>> SEM <<TextTrans(tx);>> | SEM <<ind:=0;>>
       WriteList <<ind,arrspi,arrpar,arrval>>
       SEM <<WriteTrans(ind,arrspi,arrpar,arrval);>> ) ")"
| "acq"  "(" ListGraf <<ind,arrtyval,arrval,arricon>>
  SEM <<CheckAcq(ind,arricon); FOR inval := 0 TO 100 DO
                                 tx[inval]:=' '
                               END;>>
       namedat <<tx>> "," UnsignedInteger <<val>> ")"
          SEM <<AcqTrans(ind,val,arrtyval,arrval,arricon,tx);>>
| "begin" Statement { ";" Statement } "end"
| "if" BoolExpression <<tyval,spix,markvar>>  "then"
      SEM <<IfStat(tyval,spix,markvar);
            Label(hlab);>>  Statement
      [ SEM <<Label1(hlab);>> "else" Statement ]
        SEM <<Label2(hlab);>>
| "while"  BoolExpression  <<tyval,spix,markvar>> "do"
   SEM  <<Label4(hlab);
          IfStat(tyval,spix,markvar);
          Label(hlab1);>> Statement   SEM <<JmpStat(hlab);
                                           Label2(hlab1);>>
| "repeat" SEM <<Label4(hlab);>> Statement { ";" Statement }
  "until"  BoolExpression <<tyval,spix,markvar>>
              SEM <<IfStat(tyval,spix,markvar);
                    Label3(hlab);>>
| "for" Variable <<spix>> ":=" SEM <<zero:=0;
                                     CheckVar(spix,zero);>>
   Expression <<tyval,inval>> SEM <<CheckExpr(tyval,zero);
                                    val:=0;
                                   SetVal3(spix,tyval,inval,val);
                                    PutStack(spix);>>
      Step <<stval>>  Expression <<tyval1,inval>> "do"
      SEM <<CheckExpr(tyval1,zero);
            SubInd(markvar,stval);
            Label4(hlab);
            IfStat1;
            Mark(markvar);
            JmpStat1(hlab1);>>
        Statement SEM<<IncDec(stval,spix,markvar);
                       JmpStat(hlab);
```

```
                          Label5(hlab1);>>  ].
Step <<VAR stval:INTEGER>> =  "to"          SEM <<stval:=1;>>
                            | "downto"      SEM <<stval:=2;>>.
Type <<VAR tyval,val,val1:INTEGER>> = ( OrdinalType <<tyval>>
  | "array" "[" UnsignedInteger <<val>> ".."
  UnsignedInteger <<val1>> "]" SEM <<CheckInd(val,val1);>>
  "of" OrdinalType <<tyval>> ).
OrdinalType <<VAR tyval:INTEGER>> = "integer" SEM <<tyval:=0;>>
  | "real"     SEM <<tyval:=1;>> | "boolean"   SEM <<tyval:=2;>>
  | "frame"    SEM <<tyval:=3;>> | "thetai"    SEM <<tyval:=4;>>
  | "rotation" SEM <<tyval:=5;>> | "rotmatrix" SEM <<tyval:=6;>>
  | "vector"   SEM <<tyval:=7;>>.
ListGraf <<VAR ind:INTEGER;
           VAR arrtyval,arrval,arricon:ARRAY OF INTEGER>> =
                        LOCAL <<VAR tyval,val,val1:INTEGER;>>
   VarGraf <<tyval,val,val1>>
           SEM <<ind:=1; arrtyval[ind]:=tyval;
                 arrval[ind]:=val; arricon[ind]:=val1;>>
   { VarGraf <<tyval,val,val1>>
           SEM <<ind:=ind+1; arrtyval[ind]:=tyval;
                 arrval[ind]:=val; arricon[ind]:=val1;>> }.
ActualParameterList
 <<VAR arrspi,arrpar,arricon,arrtyval:ARRAY OF INTEGER;
   VAR arrcon:ARRAY OF REAL;
   VAR ind:INTEGER>> =
   LOCAL <<VAR spix,tyval,tyval1,inval,val:INTEGER;
     VAR reval:REAL;>> SEM <<arrpar[ind+1]:=0;>>
 "(" ( Variable <<spix>>     SEM <<arrpar[ind+1]:=1;>>
     | RobotVar <<tyval1>> "[" UnsignedInteger <<val>> "]"
                           SEM <<arrpar[ind+1]:=2;
                                 CheckVal(val);>>
     | SignedNumber <<tyval,inval,reval>> ) SEM <<ind:=ind+1;
                                 IF arrpar[ind] = 1 THEN
                                   arrspi[ind]:=spix
                                 ELSIF arrpar[ind] = 2 THEN
                                   arrtyval[ind]:=tyval1;
                                   arrspi[ind]:=val
                                 ELSIF tyval=1 THEN
                                   arrcon[ind]:=reval
                                 ELSE
                                   arricon[ind]:=inval
                                 END;>>
     { ","      SEM <<arrpar[ind+1]:=0;>>
     ( Variable <<spix>>    SEM <<arrpar[ind+1]:=1;>>
     | RobotVar <<tyval1>> "[" UnsignedInteger <<val>> "]"
                    SEM <<arrpar[ind+1]:=2;
                          CheckVal(val);>>
     | SignedNumber <<tyval,inval,reval>> ) SEM <<ind:=ind+1;
                             IF arrpar[ind] = 1 THEN
                               arrspi[ind]:=spix
                             ELSIF arrpar[ind] = 2 THEN
                               arrtyval[ind]:=tyval1;
                               arrspi[ind]:=val
                             ELSIF tyval=1 THEN
                               arrcon[ind]:=reval
                             ELSE
                               arricon[ind]:=inval
                             END;>>  } ")" .
```

```
WriteList <<VAR ind:INTEGER;
            VAR arrspi,arrpar,arrval:ARRAY OF INTEGER>> =
LOCAL <<VAR spix,val1,val,tyval:INTEGER;>>
      SEM <<arrpar[ind+1]:=0;>>
( Variable <<spix>>  SEM <<arrpar[ind+1]:=1;>>
[ "[" UnsignedInteger <<val>> "]" SEM <<arrpar[ind+1]:=2;>> ]
| RobotVar <<tyval>> "[" UnsignedInteger <<val1>> "]"
  SEM <<CheckVal(val1);>> ) SEM <<ind:=ind+1;
                                  IF arrpar[ind] = 0 THEN
                                    arrspi[ind]:=val1;
                                    arrval[ind]:=tyval
                                  ELSE
                                    arrspi[ind]:=spix;
                                  IF arrpar[ind]=2 THEN
                                    arrval[ind]:=val
                                  END
                                END;>>
 { ","      SEM <<arrpar[ind+1]:=0;>>
 ( Variable <<spix>>      SEM <<arrpar[ind+1]:=1;>>
 [ "[" UnsignedInteger <<val>> "]" SEM <<arrpar[ind+1]:=2;>> ]
 | RobotVar <<tyval>> "[" UnsignedInteger <<val1>> "]"
 SEM <<CheckVal(val1);>> ) SEM <<ind:=ind+1;
                                  IF arrpar[ind] = 0 THEN
                                    arrspi[ind]:=val1;
                                    arrval[ind]:=tyval
                                  ELSE
                                    arrspi[ind]:=spix;
                                    IF arrpar[ind]=2 THEN
                                      arrval[ind]:=val
                                    END
                                  END;>>  } .
Expression <<VAR tyval,inval:INTEGER>>  =
     LOCAL <<VAR tyval1,tyval2,opval:INTEGER;>>
  Term <<tyval1,inval>>   SEM <<tyval:=tyval1;>>
  { AddOp <<opval>>  Term <<tyval2,inval>>
         SEM <<AddExpr(tyval,tyval1,tyval2,opval);>>  }.
AddOp <<VAR opval:INTEGER>>=    "+"    SEM <<opval:=1;>>
  | "-"    SEM <<opval:=2;>> | "or"   SEM <<opval:=3;>>.
Term <<VAR tyval,inval:INTEGER>> =
    LOCAL <<VAR tyval1,tyval2,opval:INTEGER;>>
  Factor <<tyval1,inval>>       SEM <<tyval:=tyval1;>>
  { MulOp <<opval>>  Factor <<tyval2,inval>>
         SEM <<MulExpr(tyval,tyval1,tyval2,opval);>>  }.
MulOp <<VAR opval:INTEGER>> =  "*"     SEM <<opval:=1;>>
  | "/"    SEM <<opval:=2;>> | "div"   SEM <<opval:=3;>>
  | "mod"  SEM <<opval:=4;>> | "and"   SEM <<opval:=5;>>.
Factor <<VAR tyval,inval:INTEGER>> =
         LOCAL <<VAR spix,val,tyval1:INTEGER;
                 VAR indic:BOOLEAN;
                 VAR reval:REAL;>>
  SignedNumber <<tyval,inval,reval>>
     SEM  <<SetVal1(tyval,inval,reval);>>
  | Variable <<spix>>  SEM <<val:=0; indic:=FALSE;>>
   [ "[" UnsignedInteger <<val>>  SEM <<CheckArr(spix,val);
                                        indic:=TRUE;>>
     "]" ]            SEM <<CheckIndic(indic,spix);
                            SetVal2(spix,tyval,val);>>
  | RobotVar <<tyval1>> "[" UnsignedInteger <<val>> "]"
```

```
                SEM <<CheckVal(val);
                      SetValJ(tyval,val,tyval1);>>
    | "(" Expression <<tyval,inval>>")"
    | "sqr" "(" Expression <<tyval,inval>>")"
      SEM <<SqrTrans(tyval);>>
    | "sqrt" "(" Expression <<tyval,inval>>")"
        SEM <<SqrtTrans(tyval);>>
    | "sin" "(" Expression <<tyval,inval>>")"
        SEM <<SinTrans(tyval);>>
    | "cos" "(" Expression <<tyval,inval>>")"
        SEM <<CosTrans(tyval);>>
    | "arctan" "(" Expression <<tyval,inval>>")"
        SEM <<ArTanTrans(tyval);>>
    | "succ" "(" Expression <<tyval,inval>>")"
        SEM <<SuccTrans(tyval);>>
    | "pred" "(" Expression <<tyval,inval>>")"
        SEM <<PredTrans(tyval);>>
    | "not" Factor <<tyval,inval>>
    | "true"   SEM <<tyval:=2; inval:=1;>>
    | "false"  SEM <<tyval:=2; inval:=0;>>.
VarGraf <<VAR tyval,val,val1:INTEGER>> =
     RobotVar <<tyval>> "[" UnsignedInteger <<val>> "]"
                SEM <<CheckVal(val);>>
     "," UnsignedInteger <<val1>> "," SEM <<CheckVal1(val1);>>.
RobotVar <<VAR tyval:INTEGER>> =
 "robotjoints" SEM <<tyval:=0;>> | "robotnom" SEM <<tyval:=1;>>
 | "robotcont" SEM <<tyval:=2;>>.
Variable <<VAR spix:INTEGER>> = ident <<spix>>.
ProcedureIdentifier <<VAR tyval,spix:INTEGER>> =
 "call" ident <<spix>>   SEM <<tyval:=0;>>
 | "makevector" SEM <<tyval:=1;>> | "vabs" SEM <<tyval:=2>>
      .  .  .
 | "anout" SEM <<tyval:=32;>>  | "anin"  SEM <<tyval:=33;>>
 | "time"  SEM <<tyval:=34;>>  | "swtch" SEM <<tyval:=35;>>
 | "awtch" SEM <<tyval:=36;>>  | "smask" SEM <<tyval:=37;>>
 | "amask" SEM <<tyval:=38;>>.
SignedNumber <<VAR tyval,inval:INTEGER;
               VAR reval:REAL>> = LOCAL <<VAR sign:INTEGER;>>
SEM <<sign:=1;>>  [ "+"  | "-"    SEM <<sign:=-1;>> ]
UnsignedNumber <<tyval,inval,reval>> SEM <<IF tyval=0 THEN
                                           inval:=sign*inval
                                         ELSE
                                          reval:=REAL(sign)*reval
                                         END;>>.
UnsignedNumber <<VAR tyval,inval:INTEGER;
    VAR reval:REAL>> =    LOCAL <<VAR inval1:INTEGER;>>
UnsignedInteger <<inval>>  SEM <<tyval:=0;>>
[ "." UnsignedInteger <<inval1>> SEM <<tyval:=1;
                                   RealNum(inval,inval1,reval);
                                   inval:=0;>> ].
BoolExpression  <<VAR tyval,spix,markvar:INTEGER>> =
        LOCAL <<VAR inval:INTEGER;
                VAR indic:BOOLEAN;
                VAR reval:REAL;>>
        Variable <<spix>>    SEM <<indic:=FALSE;>>
        [ "=" SignedNumber <<tyval,inval,reval>>
                    SEM <<IntType(tyval,spix,inval,markvar);
                          indic:=TRUE;>>
```

```
]                SEM <<IF NOT indic THEN
                       BoolType(spix);
                       tyval:=2
                     END;>>.
```

Fig. 1

The methodology of the translator implementation is identical to that described previosly [2-4]. Basic charateristics of this translator are that it functions as a one-pass translator (each command of NRL when scanned is translated immediately into the correspondig RLMP command), immediately when the variables in NRL are declared, the names of the corresponding variables in RLMP are reserved (as this language has the built-in variables, it has no declarations).

## 4. EXAMPLE

An illustrative exmple of the program translating from NRL into RLMP is given below.

program in NRL

```
program example;
var
  tet:thetai;
  i:integer;
  ok,ok1:boolean;
procedure next;
begin
  ok:=true;
end;
begin
  ok1:=true;
  swtch(248,1,next);
  tet[0]:=1.1;
  tet[1]:=0.5;
  tet[2]:=0.7;
  tet[3]:=1.0;
  while ok1 do
    begin
    if ok then
      begin
        ok:=false;
        jdrive(tet);
        gripopen;
        nullpos;
        gripclose;
        tet[0]:=tet[0]+0.1;
        tet[1]:=tet[1]+0.1;
        tet[2]:=tet[2]+0.1;
        tet[3]:=tet[3]+0.1;
        time(0,1000)
      end
    end
end.
```

program in RLMP

```
new
jmp 20
lab 10
sflg 0 1
ret
lab 20
sflg 1 1
wtch s 248 1 10
push  1.1
pop r0
push  0.5
lab 40
if f 0
jmp 60
jmp 70
lab 60
sflg 0 0
put tacka 0
put kretanje 0
move r0 r1 r2 r3 r4 r5
out s 2 1
out s 1 1
push r1
push 0.1
sum
pop r1
push r2
push 0.1
sum
pop r2
push r3
push 0.1
sum
```

```
pop r1
push  0.7
pop r2
push  1.0
pop r3
lab 30
if f 1
jmp 40
jmp 50
```

```
put tacka 0
put kretanje 0
move 0 0 0 0 0 0
out s 2 1
out s 1 0
push r0
push 0.1
sum
pop r0
```

```
pop r3
time 0 1000
lab 70
jmp 30
lab 50
end
```

## 5. CONCLUSION

The construction of the new robotic language, NRL, and a translator from this language into the RLMP enables the programming of the ROBED-01 robot at the level of a higher programming language. One of important properties of this translator is that the source language NRL can be, in a simple way, modified and expanded by new structures. This is achieved by adding the syntax structure and the corresponding semantic actions to the input file.

## 6. REFERENCES

[1] Blume, C., Jakob, W., Programing Languages for Industrial Robots, Springer-Verlag, 1986.

[2] Racković, M., Surla, D., "Construction of a Translator for Robotic Languages with the Aid of Compiler-Compiler Coco-2. Part I.", (submitted for XXXVI Yugoslav Conference ETAN, Kopaonik, 1992).

[3] Surla, D., Racković, M., "Construction of a Translator for Robotic Languages with the Aid of Compiler-Compiler Coco-2. Part II.", (submitted for XXXVI Yugoslav Conference ETAN, Kopaonik, 1992).

[4] Racković, M., Surla, D., "Implementation of a Translator for Robotic Languages with the Aid of Compiler-Compiler Coco-2.",

[5] Dobler, H., Pirklbauer, K., "Coco-2 - A New Compiler Compiler", SIGPLAN Notices, Vol. 25, 1990.

[6] Blume, C., Jakob, W., Pasro. Pascal for Robots, Springer-Verlag, 1985.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 115 - 119

# MODEL FOR SELECTION AND CREATION OF REUSABLE SOFTWARE

*Prof. dr Ivan Seder, mr Sasha Boshnjak ,Faculty of Economics*
*Department of Informatics,Mathematics & Statistics*
*Mose Pijade 12, 24000 Subotica*

## ABSTRACT

Reusability of products, processes and other knowledge is one of the key aspects to achieve rising productivity and quality in software production. Software life cycle enables entire reuse of software's objects on different abstraction levels. One of the crucial problems is selection and creation of reuse candidates and also the way of their adaptation to the new environment. In this paper, a model for selection and creation of reuse candidate by means of suitable process is described. The relevant attributes that characterize such objects are defined. Also, the methods of transformation from candidate to suitable object are mentioned and it's characteristics are pointed out. Moreover, the expirience base and reuse repositrory which contain the reuse object are described.

## *1. INTRODUCTION*

Requirements concerning software production, which are nowadays appreciated, are mainly related to the enhance of software quality and also to the time-saving production of software. The use of software development methodology that deals with reuse approach can make significant contribution to the higher quality and productivity. The reuse approach implies reuse of existing solutions during development of new software systems. In general, existing solutions are called reusable objects. They can present:

- *software experience*
- *tested products*
- *tested processes*
- *models for evaluation of software quality and productivity.*

Generally speaking, the reuse of existing experience is basic process in any discipline. If we define this new reuse methodology as systematic and disciplined approach to the software development, implementation and maintenance in which reuse is basic concept, than it means that the methodology covers the whole software life cycle. Reuse objects in this methodology are objects on different abstraction levels and are related to every phase of software life cycle. We need a model for selection and creation of reuse objects in order to apply this methodology for software development process.

## *2. ASSUMPTION ON REUSE OBJECTS APPLICATION*

Reuse-oriented software development assumes that, given the information requirement X' for object X, we consider reusing an already existing object Xk instead creating X from the beginning. The reuse process is realized through the following steps:

*step 1.*

Set of reuse candidate X1,X2...Xn is identified from an experience base which is simply called repository of reuse objects.

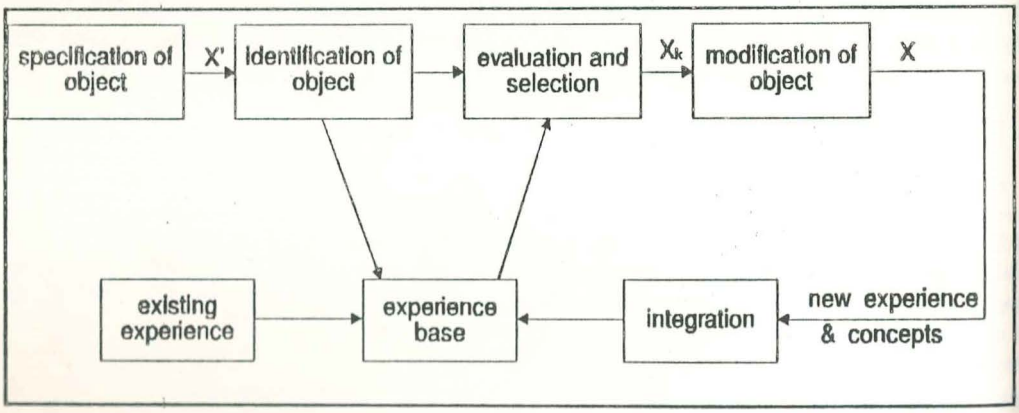
*step 2.*

Evaluation of identified candidates for satisfying requirements X' and selection of the best suitable candidate Xk.

*step 3.*

Modification of selected candidate Xk into required object X.

A knowledge about, the objects which have the characteristic of reuse object is growing with each new project. This is the way of establishing a better criterion for selection of reusable objects which are to be added to the experience base. Any kind of software experience, starting with any product belonging to any phase of software life




*Picture 1. Model of software procces*

cycle, and ending with the expert knowledge related to whole cycle is stored in the experience base.

Model of software development process that is based on described assumptions above is illustrated on picture 1.

Reusable methodology uses certain assumptions that are falling:

a) Instead of traditional view about using only concrete reuse object (e.g. part of source code or program design), we consider reuse object as all types of software experience and knowledge about software process. The product can be concrete document, created during software process or product model describing class of concrete documents or objects with common characteristics.

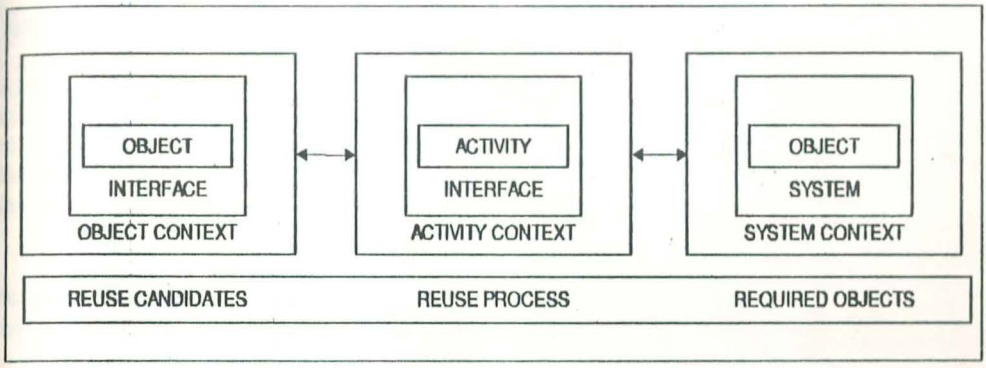
b) Software reuse mainly requires modification of reuse object which was chosen as the most convenient candidate into the required object X. The required object is integrated into the new software system. Reuse object modified in experience base remains there because of its suitable characteristics.

c) The question of applying reuse methodology requires detailed analysis. Decision whether existing experience is good or not can be brought only after the expected results have been analyzed.

d) Reuse methodology should be integrated in a specific software development model. If we consider waterfall model as development one, it is necessary than to check all steps in designing phase. Documents needed in this phase have to be stated for input and output. Results of reusable methodology application cannot be immediately identified and their valuation is very difficult.

## *3. MODEL FOR SELECTION AND CREATION OF REUSE OBJECTS*

Due to complexity of reuse objects application, within reuse methodology, it is necessary to define a suitable model of reuse objects. Such model demands criterion and attributes for reuse candidates, for reuse process of transforming candidate in suitable object. Such models should satisfy some basic requirements:




*Picture 2. Reuse process*

- application of all types of reuse objects;
- reuse candidate modeling should satisfy new environment's requirements
- modeling reuse process so that it can be applied to the selection of appropriate reuse candidates.

The above mentioned model (pic. 2.) was designed to satisfy assumptions defined in section 2. Transformation of reuse model is realized through set of activities based on reuse approach and is called reuse process. Integration of such activates into complete software development process is also part of reuse model.

Reuse process consist of four basic activates: identification, evaluation, modification and integration of reuse objects. Attributes of reuse candidates are given by following sheme:

- *name (e.g. prog.c, expen.pascal, input.prg)*
- *function or purpose (e.g. input-array, sort, expen-estimate)*
- *kind of usage (product, process, knowledge)*
- *type (source code, design, document, requirements document, tools and techniques)*

*- granularity (system level, component-package, program, function)*
*- representation (language, schemised template, set of information about process)*

Interface between required objects and experience base contains input/output parameters of the product that are needed for completing definition of object. Furthermore, the information about required object have to contain the historical picture about previous application, previous language and previous methods (e.g. waterfall life cycle). Basic distinct between reuse candidate and required object is in characteristics and the stated reuse requirements. Reuse candidate is potential object for reuse, while required object already has characteristics of the software system in which it will be integrated. The most important activity within transformation process is modification of reuse candidate into required object. Modification method can be deposed into four categories:

*a) Object without modification*

The type of object is black-box because it do not require any modification for building into new software system as required object.

*b) Manual modification*

Certain parts of objects need manual adaptation towards satisfying the new system requirements (white box). Insufficiency of methods can probably inputting of errors.

*c) Modification by fulfilling frame or template*

Frame or template contains constant parts that present their functionality. It also contains changeable parts which functionality is defined and that can be modified in accordance to the actual requirements. This method is simplier and safer than manual modification although the possibility inputting of errors still exist. This method is actual combination of black box and white box technique.

*d) Modification by parametrisation of objects*

Constants, variables or expressions as the elements of reuse objects may distinguish from contents and during process modification they are substituted with actual values that respond the parameter.

Experience base (pic. 1) that contains reuse object is described by suitable formal model that is called metasheme. Metasheme should contain all the attributes that identifying object specified as reuse candidate and that are described in this section. The most important problem about repository of reuse objects is the problem of their classification and systematization because with growth of attributes number their classification becomes more complex. The idea is to define such classification sheme that will provide quick insight, as well as selective approach to large amount of object. So far there are no satisfying and general solutions, except the partial mechanism for reuse candidate approach. The well-known procedure is information retrieval system and expert system approach. Neither of these procedures is intensively applied in software production supported by reuse methodology.

## *CONCLUSION*

This paper describes software development model that is based on reuse approach that enables each object to be used in software process. The model is general and it allows usage of the wide range of reuse objects such as software experience, products, processes and models for software quality and productivity evaluation. The application of this model can be

realized through classical approaches to the software development process: waterfall model, prototyping, object oriented approach. The selection of the most suitable candidate from the repository of reusable objects is one of the crucial tasks. The chosen object should respond to the required object specification, in order to be integrated into new system. The defined characteristics of reusable objects are the necessary background for specifying the repository of reusable objects. All of defined modification methods imply certain advantages and disadvantages. The practical testing of this method will be realized through building of reuse object repository in programming language C and that is the base for creation of new software products in accordance to the described principals and structures.

***REFERENCES***

*1. T. S. Biggerstaff, A. J. Perlis: "Software Reusability - Concepts and Models", Volume I, ACM Press, 1989;*
*2. T. S. Biggerstaff, A. J. Perlis: "Software Reusability - Applications and Experience", Volume II, ACM Press, 1989;*
*3. M. Lenz, H. A. Schmid, P. E. Wolf: "software Reuse through Building Blocks", IEEE Software July 1987, pp. 34-42;*
*4. V. R. Basili, H. D. Rombach: "Support for comprehensive Reuse", Software Engineering Journal, September 1991. pp 303 - 316;*
*5. T. S. Biggerstaff: "Reusability Framework, Assessment and Directions", IEEE Software 1987, 4, pp 41 - 49;*
*6. A. S. Peterson: "Comming to Terms with Software Reuse Terminology: A Model-Based Approach" ACM SIGSOFT, Software Engineering notes, Volume 16, No 2, pp. 45-51*

□

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 121 - 127

# On the Application of Back–Propagation Method for Solving Optimal Control Problem with Constraint on Control

Dušan Surla [1], Nataša Divljak [2]
Institute of mathematics
Trg Dositeja Obradovića 4
21000 Novi Sad, Yugoslavia

**Abstract**

The algorithmic structure for solving optimal control problem with constraint on control is given. The back–propagation rule is applied and penalty function method is used. The algorithm is illustrated by a numerical example.

**Key words:** neural network, back–propagation rule, optimal control.

## 1. Introduction

Neural network are suitable methods for solving optimization problems, including the optimal control problem. The methods for practical on–line and off–line learning of the mapping of inverse kinematics and dynamics of a robotic arm using back–propagation method has been proposed [1]. The application of static and dynamic back–propagation method for identification and control of dynamical system has been described [6]. In paper [2], the optimal control problem is converted to the two–point–boundary–value problem and then transformed into the problem of minimization of an error function. This problem is solved using Hopfield–Tank neural network. The modified Hopfield type network based on the conjugate gradient method has also been developed. A new nonlinear regulator design method that integrates linear optimal control techniques and nonlinear neural network learning methods has been presented [3].

Numerical methods of optimal control applied for the synthesis of nominal dynamics of robotic manipulators are given in [11], [4], [5]. These methods are based on classical theory of optimal control and complete dynamic model of robotic manipulators. Modification of these methods for solving optimal control problem using neural networks has been presented [8]. Input units were components of the state vector and output was the control vector. Weights were corrected on the basis of minimization of the Hamiltonian function using back–propagation rule.

In the present paper, the algorithms given in [8] and [9] are modified for solving the optimal control problem with constraint on control. This problem is solved using penalty function of equality and nonequality type.

[1]This research was partially supported by Science Fund of Serbia, grant number 0403, through Matematički Institut

[2]This research was partially supported by Science Fund of Serbia

# 2. Optimal control problem

## 2.1. Pontryagin's principle

Consider the following dynamic system:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t)), \quad \mathbf{x}(0) = \mathbf{x}_0 \tag{1}$$

where $\mathbf{x}(t) \in R^n$ is the state vector, $\mathbf{u}(t) \in \Omega \subset R^m$ is the control vector and $\mathbf{f} : R^+ \times R^n \times R^m \to R^n$ is a continuous function.

The optimal control problem is to find the control $\mathbf{u}$ which minimizes the optimality criterion

$$J(\mathbf{u}) = K(\mathbf{x}(T)) + \int_0^T L(t, \mathbf{x}, \mathbf{u})dt, \tag{2}$$

where $T$ is given.

By Pontryagin's minimum principle, optimal control satisfies the following set of equations:

$$\dot{\mathbf{x}} = \frac{\partial H(t, \mathbf{x}, \mathbf{u}, \mathbf{p})}{\partial \mathbf{p}} \tag{3}$$

$$\dot{\mathbf{p}} = -\frac{\partial H(t, \mathbf{x}, \mathbf{u}, \mathbf{p})}{\partial \mathbf{x}} \tag{4}$$

with the boundary conditions

$$\mathbf{x}(0) = \mathbf{x}_0 \tag{5}$$

$$\mathbf{p}(T) = \frac{\partial K(\mathbf{x}(T))}{\partial \mathbf{x}} \tag{6}$$

and

$$\min_{\mathbf{u} \in \Omega} H(t, \mathbf{x}, \mathbf{u}, \mathbf{p})$$

where

$$H(t, \mathbf{x}, \mathbf{u}, \mathbf{p}) = L(t, \mathbf{x}, \mathbf{u}) + < \mathbf{p}, f(t, \mathbf{x}, \mathbf{u}) > \tag{7}$$

is the Hamiltonian function.

## 2.2. Statement of the problem

The problem is to find optimal control $\mathbf{u}$, such that system (1) with constraint on $\mathbf{u}$, $|u^i| \leq \gamma_i, i = 1, ..., m$ is transferred from the initial state $\mathbf{x}(0) = \mathbf{x}_0$ into terminal state $\mathbf{x}(T) = \beta$ and minimizes the optimality criterion

$$J(\mathbf{u}) = \int_0^T L(t, \mathbf{x}, \mathbf{u})dt, \tag{8}$$

where $T$ is given and $\beta$, $\gamma$ are constant vectors.

This problem with constraints can be converted into the problem without constraints, putting

$$K(\mathbf{x}(T)) = \frac{1}{2\epsilon}\|(\mathbf{x}(T) - \beta)\| \tag{9}$$

and by minimizing the penalty function

$$s(u) = \frac{1}{2\epsilon}\sum_{j=1}^{m}[(\max(0, u^j - \gamma^j))^2 + (\max(0, -u^j - \gamma^j))^2] \tag{10}$$

when $\epsilon \to 0$, and $\|\cdot\|$ is the Euclidian norm.

For discretization of system (3)–(4) we use the following notation:

$$h = T/q$$

$$\left.\begin{array}{l} t_i = ih \\ \mathbf{x}_i = \mathbf{x}(t_i) \\ \mathbf{u}_i = \mathbf{u}(t_i) \\ \mathbf{p}_i = \mathbf{p}(t_i) \\ H_i = H(t_i, x_i, u_i, p_i) \\ s_i = s(\mathbf{u}_i) \end{array}\right\} i = 0, ..., q.$$

## 3. Learning method

We first describe the multilayered feedforward neural network used here. The following notation is introduced:

$o_i$ – output of the $i$th unit

$w_{ij}$ – weight of the connection from the $j$th to the $i$th unit

$b_i$ – bias of the $i$th unit

The output $o_i$ of each unit is a function of its net input $net_i$:

$$o_i = f(net_i)$$

$$net_i = \sum_j w_{ij}o_j + b_i$$

The activation function $f$ is

$$f(x) = \frac{1}{1 + e^{-x}}$$

in the hidden layer and the identity for output units

$$f(x) = x$$

Activations of the input units are set to values determined by the state vector $\mathbf{x}_i$ and the obtained values of output units are the values of control vector $\mathbf{u}_i$. Weights $w_{ij}$ and biases $b_i$ are initially set to random values between 0 and 1. Formally,

$$\mathbf{u}_i = \Phi(\mathbf{W}, \mathbf{x}_i, \mathbf{b}), \tag{11}$$

where $\mathbf{W}$ is the weight matrix, and $\mathbf{b}$ is the vector of biases.

The back–propagation algorithm is one of the most popular method for supervised learning [3]. It is based on the gradient descent method of square error measure between the target value and the obtained value for a given set of patterns. The object is to find a set of weights which minimizes this function. We adapted this method for solving the problem described in 2.2. Since we want to minimize functions $H(7)$ and $s(10)$, we

use function $H + s$ instead of the error function. Weights are modified according to the following rule:

$$\Delta w_{ij} = -\eta \sum_{l=0}^{q} \frac{\partial(H_l + s_l)}{\partial w_{ij}},$$

where $\eta > 0$ is the learning rate.

$\partial(H_l + s_l)/\partial w_{ij}$ is computed as in standard back–propagation rule:

$$\frac{\partial(H_l + s_l)}{\partial w_{ij}} = \frac{\partial(H_l + s_l)}{\partial net_{si}} \frac{\partial net_{si}}{\partial w_{ij}} = \frac{\partial(H_l + s_l)}{\partial net_{li}} o_{lj}$$

Therefore,

$$\Delta w_{ij} = \eta \sum_{l} \delta_{li} o_{lj},$$

where

$$\begin{aligned} \delta_{li} &= -\frac{\partial(H_l + s_l)}{\partial net_{li}} \\ &= -\frac{\partial(H_l + s_l)}{\partial o_{li}} \frac{\partial o_{li}}{\partial net_{li}} \\ &= -\frac{\partial(H_l + s_l)}{\partial o_{li}} f'(net_{li}) \end{aligned}$$

For the output units:

$$\delta_{li} = \left( \frac{\partial H_l}{\partial u_l^i} + \frac{1}{\epsilon}(\max(0, u_l^i - \gamma^i) + \max(0, -u_l^i - \gamma^i)) \right) f'(net_{li})$$

and for hidden units:

$$\delta_{li} = f'(net_{li}) \sum_{k} \delta_{lk} w_{ki}$$

So, the $n$th correction of the weights is described as

$$\Delta w_{ij}(n) = \eta \sum_{l=0}^{q} \delta_{li} o_{lj} + \alpha \Delta w_{ij}(n-1), \tag{12}$$

where $\alpha > 0$ is a momentum term which accelerate the learning.

# 4. Algorithm

On the basis of plant response $\mathbf{x}$, control $\mathbf{u}$ is determined from the neural network (11). For the integration of systems (3–4), the Euler method is applied.

The algorithm for determining optimal control problem described in 2.2 is of the following form:

input : $\mathbf{W}$, $\mathbf{x}_0$, $\beta$, $\gamma$, $\epsilon$, $\delta$, $k$, $H'$ ($\|H'\|$ – large value)
done=false;
repeat
  $\mathbf{u}_0 = \Phi(\mathbf{W}, \mathbf{x}_0, \mathbf{b})$
  for $i$=1 to $q$ do
    begin

$\mathbf{x}_i = \mathbf{x}_{i-1} + hf(t_{i-1}, \mathbf{x}_{i-1}, \mathbf{u}_{i-1});$
$\mathbf{u}_i = \Phi(\mathbf{W}, \mathbf{x_i}, \mathbf{b})$
end;
$\mathbf{p}_q = (\mathbf{x}_q - \beta)/\epsilon;$
for $i$=q-1 downto 0 do
$\mathbf{p}_i = \mathbf{p}_{i+1} + h\frac{\partial H_i}{\partial \mathbf{x}_i};$
for $i$=0 to $q$ do
$H_i = H(t_i, \mathbf{x}_i, \mathbf{u}_i, \mathbf{p}_i);$
if $(\|H\| - \|H'\| < k)$ then
if $(\|\mathbf{x}_q - \beta\| < \delta)$ then
done=true
else
$\epsilon = \epsilon/2;$
else
change of weigths according to (12);
$H' = H$
until done;

## 5. Numerical Example

Consider the system:

$$\begin{aligned} \dot{x}^1(t) &= x^2(t), \\ \dot{x}^2(t) &= u(t) \end{aligned}$$

with constraint on $\mathbf{u}(t)$, $|u| \leq 1$.

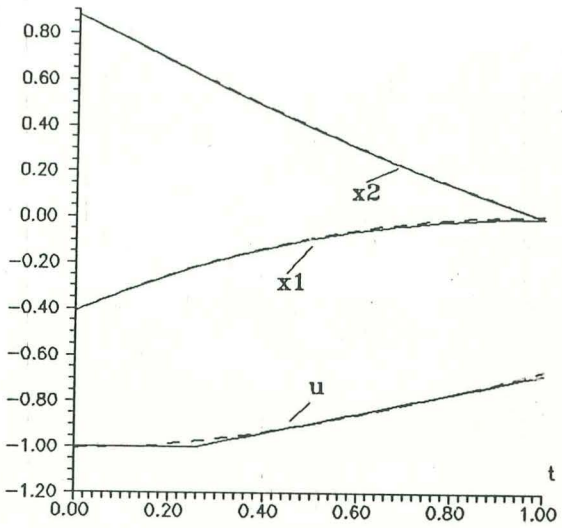



Fig. 1 *Numerical and analytical solution*

The initial state is $x^1(0) = -0.41, x^2(0) = 0.88$, terminal state $x^1(1) = 0.0, x^2(1) = 0.0$ and the optimality criterion

$$J(u) = \frac{1}{2}\int_0^1 u^2(t)dt$$

Numerical results are illustrated in Fig 1. Analytical solutions of $x^1, x^2$ and $u$ are represented by solid lines and numerical values are represented by dashed lines. Number of discrete points is $q = 50$, $\eta = 10^{-6}$, $\alpha = 0.95$, $k = 10^{-3}$, $\delta = 10^{-2}$. Three–layered neural network having 10 units in the hidden layer is used.

## 6. Conclusion

There are several numerical methods for solving optimal control problem based on classical optimal control theory. These methods can be modified for solving optimal control problem using neural networks. One possibility is to set the output state vector as input to the neural network and control vector as output. For modification of weights of connections in neural network, several rules can be applied. In this paper, simulation results of determination of optimal control using back–propagation rule are presented.

## References

[1] Bassi, D. F., *Connectionist Dynamic Control of Robotic Manipulators*, PhD thesis, Faculty of the Graduate School, University of Southern California, 1990.

[2] Biswas, S. K., *A Conjugate Hopfield Neural Network for Optimum Systems Control*, Proc. of the 29th Conf. on Desicion and Control, Honolulu, 1990, 1757-1762.

[3] Iiguni, Y., Sakai, H., Tokumaru, H., *A Nonlinear Regulator Design in the Presence of System Uncertainties Using Multilayered Neural Networks*, IEEE Trans. Neural Networks, Vol. 2, No. 4, 1991, 410-417.

[4] Konjović, Z., Vukobratović, M., Surla, D., *Energetic Analysis of Manipulation robots Nominal Dynamics*, Proc. of 2nd Project Workshop on CIM and Robotics Applications, Beograd, 1991.

[5] Konjović, Z. *Prilog automatskom planiranju trajektorija manipulacionh robota*, doktorska disertacija, Fakultet tehničkih nauka, Univerzitet u Novom Sadu, 1992.

[6] Narendra, K. S., Parthasarathy, K., *Identification and Control of Dynamical Systems Using Neural Networks*, IEEE Trans. Neural Networks, vol. 1, 1990, 4–27.

[7] Rumelhart, D. E., McClelland, J. L. and PDP Research group, *Parallel Distributed Processing: Explorations in the Microstructure of Congnition*, vol. I, MIT Press, Cambridge, Massachusetts, 1986.

[8] Surla, D., Divljak, N. *Back–Propagation Method for Solving Optimal Control Problem*, Bull. Appl. Math. (in print)

[9] Surla, D., Divljak, N. *On the Application of Back-Propagation Method for Solving Optimal Control Problem with Boundary Conditions*, Proc. of XIX SYM-OP-IS '92 (in print).

[10] Vukobratović, M., Stokić, D., *Scientifical Fundamentals of Robotics 2, Control of Manipulation Robots*, Monograph, Springer-Verlag, Berlin, 1982.

[11] Vukobratović, M., Konjović, Z., Surla, D., *Computer-aided Synthesis of Optimal Robot Trajectories by Using the Hamiltonian and Complete Dynamic Model*, MATHEMATICAL and INTELLIGENT models in system simulation, R. Hanus, P. Kool, S. Tzafestas (editros), J.C. Baltzer AG, Scientific Publishing Co., IMACS, 1991, 591-596.

PROCEEDINGS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE URA 52
NOVI SAD, OCTOBER 29-31, 1992., PP. 129 - 134

# Prolog within Smalltalk and the objects unification problem

*Dušan Tošić* *Radivoj Protić*

*Matematički fakultet*
*Studentski trg 16*
*11000 Beograd*
*Yugoslavia*

**Abstract.** In this paper we consider the problem of using the objects from Smalltalk within the Prolog programs. The Prolog variables now can have values in a set of objects defined in Smalltalk. The main arising problem is the unification when the domain of values of variables is spread on this way. The problem of unification is a central one in Prolog. By integration Prolog and Smalltalk we deduce this problem on the problem of equality two objects. As a working environment, Smalltalk/V (including Prolog/V) is used.

**Keywords:** Prolog, Smalltalk, Unification, Objects.

## 1. Introduction

The programming language Prolog is used in various problem solving areas, but some of its limitations are evident from the very beginning. There are a lot of papers with the proposals how to overcome those limitations. For example, in [3] an extension of Logic Programming is introduced, aimed at replacing the pattern matching mechanism of unification, as used in Prolog, by a more general operation called constraint satisfaction. In [4] Colmerauer describes the Prolog III programming language extends Prolog by redefining the fundamental process at its heart: unification. The article [1] is devoted to the generalisation of Prolog by universalising Horn clauses and by changing the process of unification. This direction to improvement of programming language Prolog is characterised by introducing a divers set of domains including: reals, booleans, trees, lists, etc. and by changing the process of unification. So, the main problem related to new-introduced domains is the unification problem. This problem is the subject of exploration for itself in many works. An abstract unification with an abstract domain is studied in [5].

An other direction in enlarging the productivity of Prolog is its integration into diverse programming environments. For example, in [8] the programming language LogiC++ is defined by mixing Prolog clauses and methods from C++. In [2] an integration of Prolog and 'C' is proposed. Also, the efforts of making an object oriented

Prolog are significant ( see: [7]), especially parallel versions of Prolog.

We experiment with Prolog/V into Smalltalk/V environment ( see: [9]). In this environment a kind of integration of Prolog and Smalltalk is realised. The power of Prolog is enlarged by the possibility of using of Smalltalk expressions. Meanwhile, it seems that the problem of using Smalltalk objects in Prolog programs is not solved in straight way. For a successful using of Smalltalk objects in Prolog, we recognize the unification problem as the central one. A conception of solution based on the redefinition of equality of objects is offered in this paper.

## 2. The Smalltalk objects and unification in Prolog

A Prolog unification algorithm is based on equation of two terms ( see [10]). If the domain of Prolog would be spread by Smalltalk objects, the power of Prolog could be increased meaningfully. It is possible to use the Smalltalk objects in Prolog/V, but if the objects are not standard Prolog structures, the obtain results from Prolog program are not always correct. Our aim is to spread the domain of Prolog/V without changing the backtracking mechanism and unification algorithm. We can do that with non-standard Prolog structures (Fig. 1.), i.e. with the objects from all classes not included in Prolog/V.

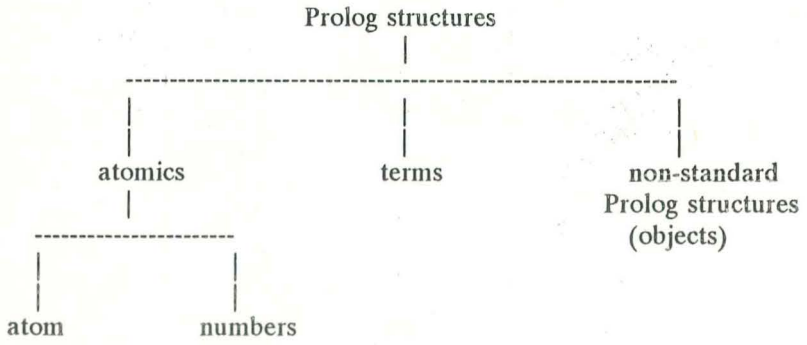



Fig. 1.

In this article we propose a way how to introduce these non-standard structures in Prolog/V, using some object-oriented features of Smalltalk.

## 3. The comparing of objects in Smalltalk

The unification algorithm implemented into Prolog/V interpreter is based on the method '=' imported from Smalltalk. The equality of two structures is based on that method and the arising problem is its implementation. Namely, two messages are used in Smalltalk for the comparing of two objects (see: [6] and [9]).

The message:

== <object>

returns 'true' when the receiver and <object> are identical i.e. when two objects are the same object. The message:

= <object>

returns 'true' when the receiver and <object> are equal. A common realisation of this method is checking of equality the values for all corresponding components. Those messages are implemented for all objects and the default implementation of '=' is the same as that of '=='. If redefinition of the message '=' is not made ( by default it is the same as the message '=='), we can not expect satisfactory results from Prolog program. For example, if we use the objects from the class **Bag** in a Prolog/V program, the unification of objects (7,8) and (8,7) can not be successful. Meanwhile, it is normal to expect the success of unification in this case. The same problem appears if we introduce any new class without redefying the message '='. In fact, the definition of message '=' in the class **IndexedCollections** is enough to support the process of standard unification in Prolog/V. In many classes the redefinition of the message '=' is required.

## 4. The unification based on the redefinition of message '='

The possibility of using objects in Prolog/V is based on the consistent implementation of the method '=', according to the semantics of classes. It means that we should modify this method in a lot of classes. Its modification in Smalltalk implies a modification of the unification algorithm in Prolog/V. We call this process: "Modification of the unification algorithm from outside".

How to modify the method '=' in Smalltalk ? The answer of that question depends on the nature of classes in Smalltalk. In other words, the solution is related to intrinsic features of objects from different classes.

In some classes the solution is very simple. For example, in the class **FixedCollection** the message '=' may be accepted without changing ( as it introduced by default). Meanwhile, for many other classes, the modifications are necessary. The modification could be realised by writing a new method '=' for the each Smalltalk class.

Let us consider an example. In the class **Bag** the message:

( 7, 8) = (8 , 7)

should return the value 'true'. So, the equality of two objects in this class does not depend on the ordering of the components in these objects. We reimplemented the method "=" for the class **Bag** according to the previous conclusion:

```
= aBag
" Equality of Bags "
(aBag class = (self class)) ifFalse: [^false].
self do: [ :i |
  ((aBag occurrencesOf: i) = (self occurrencesOf: i))
    ifFalse: [^false]
  ].
^true
```

The similar problem is appearing in the class **Set**. The method suggested for the class **Bag** could be used in this case too. Moreover, we implemented a new efficient

code for this class:

```
= aSet
 " Equality of Sets "
(aSetBag class = (self class)) ifFalse: [^false].
self do: [ :i | (aBag includes: i) ifFalse: [^false]].
^true
```

In general case the problem is quite complex because it is not clear how to make the modification of the method '=' for the other classes, especially for the new-introduced classes.

## 5. The unification of objects depending on semantics

The method '=' might be defined in different ways at a class. Let us consider a simple example.

The application class Man with two instance variables is defined on the following way:

```
Object subclass: Man
instanceVariableNames: 'age profession'
```

Two variables, Peter and Fred, of the class Man could be instanced in a Prolog program as follows:

```
Peter:                    Fred:
  age: 35                   age: 35
  profession: 'teacher'     profession: 'policeman'
```

Two men could be treated as equal if they are same age. In that case the unification in our example would success. This way is not convenient because it could not be applied generally.

Meanwhile, usually is required the equality of all instance variables. We suggest this way because it might be generalised for the new-introduced classes. In this case is possible to create a general algorithm for the method '='. It is necessary for all instance variables:

1. the possibility of getting the values outside of object and
2. the method '=' should be defined on adequate way.

The corresponding Smalltalk methods for the case 1. are:

```
inst1
 " value of instance variable inst1"
 ^inst1

inst2
 " value of instance variable inst2"
 ^inst2
```

```
....

instm
  " value of instance variable instm"
  ^instm
```

and for the case 2. the method is:

```
= aNewClass
  "Check equality of objects for a class NewClass "
(((self inst1 class) = (aNewClass inst1 class))
     and: [(self inst1) = (aNewClass inst1)])
   ifFalse: [^false].
(((self inst2 class) = (aNewClass inst2 class))
     and: [(self inst2) = (aNewClass inst2)])
   ifFalse: [^false].
....
(((self instm class) = (aNewClass instm class))
     and: [(self instm) = (aNewClass instm)])
   ifFalse: [^false].
^true
```

We have modified the method '=' for a few classes in Smalltalk/V. By modifying the method '=' in all classes of Smalltalk/V, we expect to get a new environment for Prolog/V.

## 6. Conclusion

The redefinition of the method '=' and systematic development of the Smalltalk classes makes it possible to use the objects from whole hierarchy classes in Prolog programs. We consider here a concrete software product where two different programming paradigms are integrated. Moreover, the similar problems appear in any programming environment where Prolog and Smalltalk are integrated.

## 7. References

[1] Aronsson A, Eriksson L.-H, Garedol A, Hallnas L. and Olin P, *The Programming Language GCLA - A Definitional Approach to Logic Programming,* New Generation Computing, Vol 7, No. 4, 1990, 381-404.

[2] Boyd J. and Karam G, *Prolog in 'C',* ACM Sigplan Notice, Vol. 25, No. 7, 1990, 63-71.

[3] Cohen J, *Constraint Logic Programming Language,* Comm. of ACM, Vol. 33, No. 7, 1990, 52-68.

[4] Colmerauer A, *An Introduction to Prolog III,* Comm of ACM, Vol. 33, No. 7, 1990, 69-90.

[5] Cortesi A. and File G, *Abstract Interpretation of Logic Programs: An Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis,* in Proc. "Symposium on Partial Evaluation and Semantics-Based Program Manipulation", Sigplan Notices, Vol. 26, No. 9, 1991, 52-61.

[6] Goldberg A. and Robson D, *SMALLTALK-80 the Language and its Implementation,* Addison-Wesley Pub. Comp. 1983.

[7] Mello P. and Natali A, *Objects as Communicating Prolog Units,* in Proc. "ECOOP '87 European Conference on Object-Oriented Programming" (eds. J. Bezvin, J.-M. Hullot, P. Cointe, H. Lieberman), Springer-Verlag, 1987, 181-191.

[8] Shawm-inn We, *Integrating Logic and Object-Oriented Programming,* ACM OOPS Messenger, Vol. 2, No. 1, 1991, 28-37.

[9] *Smalltalk/V Tutorial and Programming Handbook,* Digitalk inc., 1986.

[10] Sterling L. and Shapiro E, *The Art of Prolog, Advanced Programming Techniques,* The MIT Pres, 1986.


UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCE
INSTITUTE OF MATHEMATICS

# Abstracts of the VI Conference on Logic and Computer Science

LIRA '92

Novi Sad
October 29-31, 1992

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 137

# EXPERT SYSTEMS IN PROCESSING AND ANALYSING EMPIRICALLY COLLECTED DATA

*Zita Boshnjak, Faculty of Economics,*
*Department of Informatics, Mathematics and Statistics*
*Moshe Pijade 12, 24000 Subotica*

## EXTENDED ABSTRACT

The selection of the most appropriate statistical procedure which should be performed upon a set of data with the aim of their analyse, require not only the familiarity of the researcher with the large number of different statistical procedures, but an empirical knowledge and heuristics that alow the researcher to choose the procedure that would acquire the maximum amount of relevant information and derive the accurat conclusions. In this article we have described the expert system (ES) prototype that we named SSDAP (abbreviation from Selection of Statistical Data Analysing Procedures). SSDAP aids the selection of those statistical data analysing procedures which best fulfill the requests and aims of researcher, concearning the characteristics of chosen statistical procedure. The fulfillment of the aims of undertaken research depends on the degree of suitability of performed statistical procedure. This is the main reason why we consider the selection of the most appropriate SDAP an important problem, complex enough to require high level knowledge, i.e. expertise. To make a succesfull selection among large number of statistical tests, it is not enough to be qualified and familiar with all those procedures, but to have an experience in the field, as well as heuristics. Only the person who has such a background will be succesfull in selecting the right statistical procedures that should be performed. The right procedure allow the researcher to acquire the maximum amount of relevant information. SSDAP solves the problem of correct selection by giving an advice which SDAP is optimal to use, if such a procedure exists, or by givin an advice which procedures could be used, with equal, sufficient degree of adequacy.

The SSDAP prototype is implemented in expert system shell EXSYS Professional. We have used the production system technique to represent knowledge in the knowledge base. The inference mechanism is backward chaining. In order to enable the functionality of SSDAP, it was neccesary to determine the conditions under which the specific output was possible. In other words, we had to identify the conditions that impact the problem solution. In that purpose the relevant taxonomy was built and the decision tree drawn. The rules were written down directly from the branchings in decision tree. The ES prototype SSDAP consists of 55 rules and proposes the most suitable statistical procedure(s) from the initial set of 83 statistical data analysing procedures (SDAP). These procedures are the outputs of the system.

The results presented in this paper have left open several questions and have initiate a possible directions of further research. First of all, the question of knowledge acquisition technique selection has been opened. The problem that might occur during knowledge acquisition is the disagreement among experts regarding the set of necessary and sufficient SDAPs concerning the actual problem. The next question is the selection of knowledge representation technique. The production rules showed to be the most natural way of representation. However, the overall analyse of hybrid knowledge representation techniques convenience might result with different conclusions. The subject of further investigation should be ES SSDAP prototype evaluation and eventually its further development into a complete softver product (most probably on a different hardware platform and within different software environment), as well as its long term evaluation.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 138

# THE APPLICATION OF THE MUSICAL COMPUTERS IN THE MATHEMATICAL THEORY OF MUSIC

**Miloš Čanak**
**Faculty of Agriculture**
**Nemanjina 6, Zemun**

The mathematical theory of music is a scientific discipline, which has been developing from the time of Pythagoras till nowadays. The special ascent it got during the last 20 years by development of tone geometry, scale geometry, local compositions etc. In practice it has been shown that further development of the theory is not possible without application of musical computers. It is pointed to the developing targets of these computers and logic basis of the musical programming. The basic problems of the mathematical theory of music have been considered, starting from the geometrical parameters of the tone to the study of symmetry and structures of musical compositions and the logic of their solutions by application of the musical computers.

**Key-words:** Musical computers, mathematical theory of music

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 139

# A Comment on Amalgamation Property

**Milan Grulović**
**Institute of Mathematics, University of Novi Sad**

**Trg D. Obradovića 4**
**21000 Novi Sad**

**ABSTRACT.** We present a proof that Peano arithmetic does not have amalgamation property, making, in addition, a few remarks concerning this result.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 140

# EDI - Opening the Borders of Software Packages

**Borislav Jošanov**
**Viša škola za organizaciju rada**
**Novi Sad**

Electronic Data Interchange ( EDI ) is the exchange of business documents in standard formats.

Standardization is the key condition for the usage of EDI. Dominant standards for massage exchange are UN/EDIFACT, ANSI X.12 and SWIFT.

The results of analyzing the EDI are so positive that EDI is promoted as the most important technology nowadays.

Opening the boarders of information systems, influenced by EDI, makes the projecting of information systems much more complex.

In this paper the influences of EDI on projecting the software packages are defined. They are categorized in five basic groups - application of EDI standards, initial massage controls, the reactions of software packages, interactions between business partners and actions in which have to be done before or after the usage of software packages.

Every single influence in each of this groups is explained in detail.

**Key-words:** (EDI), (projecting), (influence)

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 141

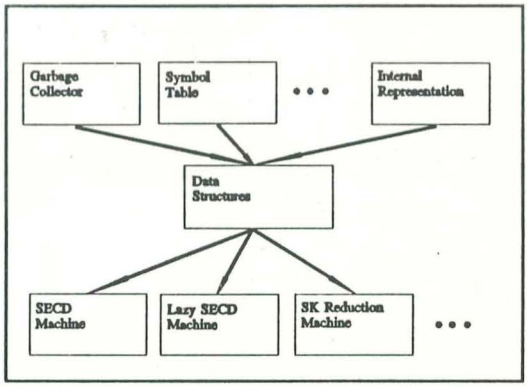
# SOME EXPERIENCES WITH SK-REDUCTION MACHINE - USER'S PERSPECTIVE

**Dragan Maćoš, Zoran Budimac**

**University of Novi Sad, Faculty of Natural Sciences and Mathematics,
Institute of Mathematics, Trg D. Obradovića 4,
21000 Novi Sad**

**ABSTRACT**. Functional languages are for the first time implemented by translation into the sequence of combinators (of combinatory logic [1]) in 1979 [3]. The resulting sequence of combinators is then represented in the form of a graph, which is then reduced according to appropriate rules for every combinator. The approach taken by SK reduction machine [3] (and many later improvements) soon became another standard approach in implementation of functional languages, besides previously established (1966) approach based on SECD machine and its variants [2]. The main difference between two machines (from the implementation point of view) is the way in which bound variables are handled. While in SECD machine based implementations, bindings of variables are stored in special data structure called *environment*, in combinator based implementations there is no variable bindings mechanism. This and other differences greatly affects program performances, but there is still no precise measurements or benchmarks which would rank two approaches with respect to time and space usage during program execution.

Presentation proposed by this abstract will show some results of the benchmark tests executed on both SK reduction machine and (lazy) SECD machine. Results are meant to help to an ordinary user (programmer in functional language) to choose appropriate machine for his needs and to learn something more about (in)efficiency of machine he currently use. The results will not be accompanied with deep analysis of the causes of machine performances.




The basis for benchmark tests is displayed in Figure. All three machines (SECD, lazy SECD and SK) are implemented using the same data structure and the same "building blocks" (garbage collector, symbol table etc.). The mentioned data structure is essentially binary tree with variable size nodes and is equally suitable for representation of all machine's internal structures: stacks in SECD machine and graph in SK machine. Both lazy SECD and SK machine support exactly the same primitive operators of the source functional language as well as the same (non-strict) semantics. Finally, implementations on which benchmarks are executed are the "classical" one and employ no optimization. Results obtained on just described basis therefore can be taken as accurate and reliable.

## References

1. Curry, H. and Feys, R., *Combinatory Logic*, North Holland, 1958.
2. Landin, P.J., *The Mechanical Evaluation of Expressions*. Comp. Journal 6(1964), 308 - 320.
3. Turner, D.A., *A New Implementation Technique for Applicative Languages*, Software - Practice and Experience, 9(1979), 31-49.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 142

# meLinda/S - A Simple Parallel Linda Programming Language

Dragan Mašulović
Institute of Mathematics, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia

## Abstract

meLinda/S is a simple programming language evolved from FORTH. FORTH background enforced slightly different syntactic attitude in comparison to conventional programming languages which led to simple solution to some basic problems arising within Linda model of parallel programming.

meLinda/S programming package consists of meLinda/S p-code compiler and corresponding abstract parallel machine. Abstract machine is built upon multi-computer/single-tasking model wich met our needs perfectly because meLinda/S does not support dynamic allocation of processes. Static allocation has been provided via special sytax constructs.

At this stage of developement meLinda/S does not support full Linda instruction set. Linda primitive *eval* has not been implemented (as stated above, meLinda/S provides static allocation of processes only).

Future work includes implementation of full Linda instruction set, along with switching to multi-computer/multi-tasking abstract machine model.

Keywords: Linda parallel paradigm

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 143

# Constructing a Language Interpreter Based on Denotational Semantic

Nenad Mitić
Matematički Fakultet Univerziteta u Beogradu,
Studentski trg 16, POB. 550, YU-11001 Beograd
email: xpmfm23@yubgss21.bitnet

Key Words: denotational semantic, functional programming language, interpreter construction

If a language has a complete denotational semantic (*J.E.Stoy, Denotational Semantics: the Scott-Strachey Approach to Programming Languages Theory, MIT Press, 1977*) an interpreter for that language can be constructed based on that semantic. Denotational semantic is given over semantic domains and semantic functions. All continuations functions are curried (high order functions). Because of that, a programming language with high order functions support was selected for interpreter construction.

As a test of the practical value of denotational semantic for constructing a language processor I have implemented an interpreter for language SMALL (*M. J.C. Gordon, The Denotational Description of Programming Language, Springer-Verlag, 1979*). This implementation has been realized in functional programming language Lispkit Lisp (*P. Henderson, Functional Programming: Application and Implementation, Prentice-Hall International, inc., London, 1980*) on IBM 3090 in an MVS/ESA TSO environment. Semantic domains were implemented over base Lispkit Lisp domains. Semantic functions were implemented, directly from semantic clauses, as high order functions in Lispkit Lisp.

The interpreter constructed this way can be simply extended by adding new functions into the Lispkit Lisp program. The interpreter consists of:

- Program, realized on PL/I language, which has two functions:
  - syntax analysis of SMALL programs, and
  - translation of SMALL programs into S-expressions by adding brackets.
- Lispkit Lisp program which interprets previously generated S-expressions.

The syntax analysis of SMALL programs is based on syntactic clauses from the denotational description of the language. The Lispkit Lisp program was constructed by using semantic clauses from denotational semantic.

From a user point of view, the interpreter can be called either in foreground TSO environment or in background MVS batch environment. Communication between users and programs can be done either interactively or through datasets.

During the testing period, the performance of the constructed interpreter was very good, regardless the Lispkit Lisp is an interpreter itself.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA 92
NOVI SAD, OCTOBER 29-31, 1992., PP. 144 - 145

# A TABLEAUX RELATED METHOD FOR THE FULL FIRST ORDER LOGIC

**- extended abstract -**

**Zoran Ognjanović, Dragan Uroševič†, Vladimir Božin**
**Mathematical Institute Belgrade & Mathematical Faculty Belgrade (†)**

**ket words: theorem proving, tableaux, resolution**

## 1. Introduction

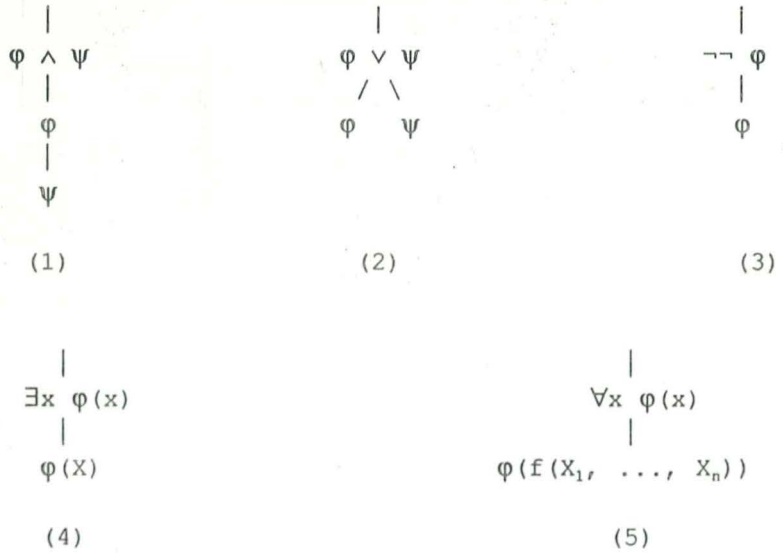
The dual tableaux method for the full first order classical logic is presented. It is based on dual tableaux for the classical logic [2] and procedure dual to the classical resolution [1].

In the method described in [2] the use of the rule for the ∃ could introduce sets of the similar sets (dual clauses) of sentences. The clauses differ only in the constants they contain. We discuss here the changes of quantifires rules that reduce the set to a single clause. As a consequence, we have to work with atomic formulas that contain variables, and we can not use the dual resolution procedure for the propositional logic as was done in [2]. Instead, we introduce a procedure dual to the first order resolution [1].

## 2. Formal system

Suppose that the full classical logic is defined as usual (over the language: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$, relation symbols, functional symbols, variables and auxiliary symbols ",", "(" and ")"). During the tableau construction we introduce so called dummy variables (or dummies). We denote them by X, $X_1$, $X_2$, ... Formulas that contain dummies we call basic formulas.

A dual modal tableau is constructed by the following rules:




The rules apply as follows. For instance, a branch with $\exists x\varphi(x)$ is extended by adding $\varphi(X)$, where X is a new dummy. In the rule (5) f is a symbol of a new Skolem function. Formulas $\neg(\varphi \vee \psi)$ and $\neg(\varphi \rightarrow \psi)$ behave according to the rule (1), formulas $\neg(\varphi \wedge \psi)$ and $(\varphi \rightarrow \psi)$ behave according to the rule (2), formula $\neg\forall x\varphi(x)$ behaves according to the rule (4) and formula $\neg\exists x\varphi(x)$ behaves according to the rule (5).

For an interpretation I, we define $I^*(\varphi)$ as a set of all terms over functional symbols of the examined formula $\varphi$ and introduced Skolem functional symbols from the formula's tableau. If there are no constants in the formula, we introduce a new initial constant symbol $\lambda$ such that $I^*(\varphi)$ contains $\lambda$. We extend the usual definition of satisfaction under the interpretation to cover new kinds of formulas containing dummies. Let $\omega$ be a mapping from set of dummies to $I^*(\varphi)$. We call it replacement.

Let I be an interpretation and $\omega$ a replacement. Then:

a) $I(f(t_1,...,t_n)_\omega) = I(f)(I(t_{1\omega}),...,I(t_{n\omega}))$,
b) $I(R(t_1,...,t_n)_\omega) = I(R)(I(t_{1\omega}),...,I(t_{n\omega}))$,
c) $I(f(t_1, ..., t_n)) = \{I(f(t_1,...,t_n)_\omega) : \omega \text{ is a replacement}\}$,
d) $I(R(t_1, ..., t_n)) = \{I(R(t_1,...,t_n)_\omega) : \omega \text{ is a replacement}\}$.

I satisfies a basic atomic formula $R(t_1, ..., t_n)$ if there is a satisfied element in $I(R(t_1, ..., t_n))$. If $\varphi$ and $\psi$ are basic formulas, I satisfies $(\varphi \wedge \psi)$ iff there is a replacement $\omega$ such that $I(\varphi_\omega) = I(\psi_\omega) = \top$. If $\varphi$ is a basic formula, I satisfies $\neg\varphi$ iff there is a replacement $\omega$ such that $I(\varphi_\omega) = \bot$.

**Lemma 1.** Let *T* be a tableau whose root contains formula $\Phi$. Formula $\Phi$ is valid iff for each interpretation I exists at least one branch of the tableau whose conjunction of all basic atomic formulas is satisfied under the interpretation I.

A dual clause is a conjunction of basic atomic formulas. To establish that a set of clauses is valid, we use the dual resolution procedure. The dual resolution rule is:

(DR) If $C_1$ and $C_2$ are dual clauses, $L_1 \subset C_1$, and $L_2 \subset C_2$, and $\mu$ is the most general unifier such that $\mu(L_1) = \neg\mu(L_2) = \{L\}$, where L is an atomic formula $R(C_1,C_2,L_1,L_2) = \mu((C_1 \setminus L_1) \cup (C_2 \setminus L_2))$

**Lemma 2.** A set of dual clauses is valid iff the empty dual clause could be inferred by the dual resolution rule.

Let *T* be a tableau for the formula $\Phi$. Then Clause(*T*) is the set of all clauses corresponding to the tableau *T*. The tableau *T* is the ***proof*** for the formula $\Phi$ iff the empty clause belongs to the closure of the set Clause(*T*) under the dual resolution rule.

**Completeness theorem.** A formula $\Phi$ is valid iff it has a proof in the system of dual tableaux.

## References


[1] Robinson J.A, A Machine-oriented logic based on the resolution principle, Journal of the ACM, Vol 12 (1965), pp. 23-41.
[2] Kapetanović M. and Krapež A, A proof procedure for the first order logic, Publications de L'Institute Matematique, vol 59 (1989), pp. 3-5.


## Acknowledgement


This work was supported by Science Fund Serbia, grant number 0403, through Mathematical Institute, Belgrade.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 146

# Isomorphisams between $\omega_1$-saturated models of complete theories

Žikica Perović

*Key words:* $\eta_1$*-ordered set,* $\omega_1$*-saturated model, real closed field, Hahn embedding*

It is well known that saturated models of a complete theory are isomorphic iff they have the same power. If we consider $\omega_1$-saturated models, of a complete theory, of cardinality $c$ we can not use this theorem unless we have CH . Could we avoid use of CH trying some other proofs. The answer is negative for a few well known theories. We present a theorem collecting some old and some new results.

**Theorem.** The following are equivalent:

i) CH

ii) Every two $\eta_1$ real closed fields of size c are isomorphic

iii) Every two divisible $\eta_1$ linearly ordered Abelian groups of size c are o-isomorphic.

iv) Every two $\eta_1$ linearly ordered sets of size c are similar.

v) Every two atomless $\omega_1$-saturated Boolean algebras of size c are isomorphic.

The equivalence i)⇔ii) answers a question of Erdös, Gillman and Henriksen.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 147

# Galois theory of Boolean algebras

Žikica Perović

*Key words: Galois extension, relatively complete extension, independent part, reduced set of generators*

We consider Galois extensions of Boolean algebras, the concept previously considered by S.Kopelberg, D.Monk,.... We present a few results of ours:

**Theorem 1.** Let C<B . B is a Galois extension of C iff it is a relatively complete finite extension of C having independent set of generators.

This theorem answers a question of D.Monk.

**Theorem 2.** Let B be a rc finite extension of C with the heigth sequence $(n_1, \ldots, n_k)$ . The followig statements are equvalent:

i) B is a pseudo-galois extension of C

ii) There exists a group G which transitively embedes into permutation groups $Sn_1, \ldots, Sn_k$.

iii) There exist ireducible polinomials of powers $n_1, \ldots, n_k$ with the same Galois group.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 148

# PROBABILITY PROPOSITIONAL LOGIC

**Miodrag Rašković**
**Prirodno-matematički fakultet**
**Radoja Domanovića 12, Kragujevac**

**Key-words:** logic, probability, completeness

The aim of the talk is to introduce probability propositional logic. Here we investigate the propositional calculus, extended by a new unary connectives $P_{r1}$, ... ,$P_{rm}$ where {r1, ... rn} $\subseteq$ [0,1].

The following type of model is relevant for us.

Def. A model is an ordered triple m = <w, $\{\mu_x : x \in W\}$, $\|$-> where:

1) $W \neq 0$ is the set of possible words x, y, z, ...

2) Each $\mu_x$ is finite additive probability measure on W

3) (the forcing) is a relation between words and formulas. So, for example, m $\|$- $P_r \varphi$ iff $(\forall x \in W)(\mu_x\{y \in W : y \| - \varphi\} \geq r)$.

Finally, we prove a completeness theorem for this logic.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 149 - 150

# INTERPRETER FOR APPLICATION OF MATHEMATICAL SPECTRA

Miomir Stanković, Jovan Madić and Predrag Stanimirović.
Filozofski fakultet, Univerzitet u Nišu
Cirila i metodija 2, 18000 Niš, Yugoslavia

Abstract. *In this paper was described a programming package implementing the interpreter of programming language which is an extension of LISP for manipulation of mathematical spectra. For this purpose several new data types and new kind of functions (called spectral data types and spectral functions) are defined and implemented.*

Keywords. *Interpreter, internal form, spectrum, spectral expressions, spectral functions, spectral constants.*

## 1. DATA TYPES

Lisp data types: Lists, symbols, strings, integers, fixed-point numbers, rational numbers, file-pointers, NIL, T, LISP-functions

Spectral data types: Vectors, spectral constants, spectral functions, spectral expressions

### 1.1 Spectral constants

A spectral constant represents the notation of the spectrum with uniform rhythm. This notation begins and ends with symbol '$'. The sign of the spectrum follows first symbol '$', and could be omitted. Strips are separated using character '|'.

### 1.2 Vectors

vector is a sequence of numbers between square brackets.

## 2. SPECTRAL FUNCTIONS

The interpreter evaluates all arguments of the spectral functions. Arguments of these functions could be spectral expressions or LISP expressions whose values are numbers, vectors or spectral constants.

We can extract several different functions types.

A. Functions for the implementation of spectral operations:
spectral addition ($+); spectral multiplication ($*); spectral subtraction ($-); right lengthening ($rlen); left lengthening ($llen); left condensation ($lcon); conversion of a spectrum to the vector ($eval); effective value of a strip ($efval); inverse spectrum ($sinver);

B. Forming new spectra:
Spectral generation ($gensp); generation of the spectrum whose strips are 1 ($1); generation of the spectrum whose strips are 0 ($0);

C. Spectral relation functions:
Equality of two spectra ($=); identity of two spectra ($ident); comparison of absolute values of two spectra ($>, $<)

### 2.1 Functions $+, $- and $*

*Description.* ($+ x y) ($- x y) ($* x y).
Both arguments are spectral expressions.

### 2.2 Function $eval

*Description.* ($eval x)
$x$ must be a spectral expression. If the value of x is the spectral constant then the result is vector whose elements are effective values of the spectral strips, otherwise, if the value is a vector, the result is the same vector.

### 2.3 Function $gensp

*Description.* ($gensp x) or ($gensp x y).

If one argument is taken, it must be a spectral expression. The result is the spectral constant of the array which is contained in the given vector or the same spectral constant given as argument.
If the function is called using two arguments then first argument is an expression whose value is integer $n$, and the second is an expression whose value is number $b$. The result is the spectral constant of $n$ spectral strips whose nominal values are $b$.

2.4 Functions $rlen i $llen

*Description.* ($rlen x y) ($llen x y).
First argument should evaluate to a spectral constant or a vector. The second argument is an expression whose value is an integer $n$. The result is spectral constant equal to the previous constant, whose strips are lengthened on the right or on the left for $n$ figures.

2.5 Function $lcon

*Description.* ($lcon x).
$x$ must be a spectral expression. If $x$ evaluate to a vector it is transformed into corresponding spectrum. The result is spectral constant equal to the previous constant, obtained after condensation on the left minimizing the number of figures.

2.6 Function $sinver

*Description.* ($sinver x).
$x$ must be a spectral expression. The result is spectral constant representing inverse spectrum of the spectrum given after evaluation of $x$.

2.7 Functions $1 i $0

*Description.* ($1 x) ($0 x).
The argument is an expression whose value is integer $n$. Values of these expressions are the spectral constants defined in this way: the rhythm is one, and contains $n$ spectral strips whose nominal values are one (zero).

2.8 Function $efval

*Description.* ($efval x y).
The first argument must be a spectral expression and the value of the second expression must be integer $n$. The result is the n-th element of the vector or the effective value of the n-th strip of the given spectral constant.

2.9 Functions $ident, $=, $>, $<

*Description.* ($ident x y) ($= x y) ($> x y) ($< x y).
Both arguments must be spectral expressions. The vector given as value of any argument is converted into corresponding spectral constant. The result is T if the spectral constants satisfy the given relation, otherwise NIL.

3. SPECTRAL EXPRESSIONS

Spectral expressions could be derived in three groups:
1. Spectral expressions whose values are spectra; 2. Spectral expressions whose values are vectors; 3. Spectral relation expression.

The first group contains following expressions: spectral constants, symbols bounded with spectra, calls of the spectral functions $+, $-, $*, $rlen, $llen, $gensp, $1, $0.

The second group contains: vectors, numbers as single-element vectors, symbols bounded with vectors, calls of the spectral functions $+, $-, $*, $eval, $efval.

Elements of the third group evaluate to the LISP constants NIL (as false) or T (as true). This group of spectral expressions contains: NIL, T, any expression whose value is T or NIL, calls of the spectral functions $=, $>, $<, $ident.

THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA 92
NOVI SAD, OCTOBER 29-31, 1992., PP. 151

# U-RANK AND THE NUMBER OF COUNTABLE MODELS

Predrag Tanovic
Matematicki institut
Knez Mihajlova 35, Beograd

We discuss the relationship between the height of the fundamenthal order of a countable, stable theory T which admits finite coding and the number of its nonisomorphic countable models.

**Theorem 1** $I(T,\aleph_0)\geq\aleph_0$

**Theorem 2** If T is superstable locally modular and $U(T)\geq\omega^\omega$ then $I(T,\aleph_0)=2^{\aleph_0}$.

**Theorem 3** If T is trivial and $I(T,\aleph_0)<2^{\aleph_0}$ then T is $\omega$-stable.

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 152

# SET-VALUED FUNCTIONS, BIO-COMPUTING AND FREQUENCY MULTIPLEXING

Ratko Tosic
Institute of Mathematics, University of Novi Sad
21000 Novi Sad, Yugoslavia

## Abstract

Set-valued functions play an important role in the study of bio-circuits and of circuits based on frequency multiplexing.

Let R = {0,1,...,r-1} be a finite set referred to as the set of fundamental values, and P(R) the set of subsets of R. The set P(R) is a Boolean algebra when equipped with the set-theoretical operations union, intersection and complementation.

Set-valued functions F: P(R)^n --> P(R) can be regarded as functions over a logic with 2^n values which provides a rich collection of functions over logics with very high radix. Few of these functions are Boolean functions, that is functions that can be constructed from constants and variables, using unions, intersections and complementation.

In the technology of bio-circuits every j, 0 <= j <= r-1, represents a pair substratum-enzyme (assuming that we deal with r distinct enzymes). Set-valued logic-networks can be used to attack the interconnection problem in highly parallel architecture. The basic concept here is "logic-value multiplexing" which means the simultaneous transmission of logic values represented by optical wavelengths through a waveguide. This concept enables us to realize simultaneous execution of several binary operations in a single module.

We investigate some problems concerning set-valued functions and Boolean collections. A non-empty collection of sets C, where C is a subset of P(R)^n is a Boolean collection if there is a set-valued function f which is Boolean function and f(X) = 0 iff X is an element of C.

It appears that the combinatorial classification of non-Boolean set-valued functions based on the maximal Boolean collections might present interest for implementations of hybrid circuitry (biological and digital, for example).

ABSTRACTS OF THE VI CONFERENCE ON
LOGIC AND COMPUTER SCIENCE LIRA '92
NOVI SAD, OCTOBER 29-31, 1992., PP. 153

# Description of the software package for robots programming by PASRO programming language

Ervin Varga
Institut za računarstvo, automatiku i merenje
Trg D. Obradovića 6, Novi Sad

**Key words:** PASRO, robot programming, translator, interpreter, source code, intermediate code

Until now, many programming languages for robots programming are developed. Many of them are based on some high level general-purpose programming languages. PASRO is one among these programming languages. PASRO is the language which is very suitable for robots programming since based on Pascal keeps all its nice characteristics. Besides, PASRO contains a set of default robot procedures and functions as well as a number of functions supporting the frame concept, which enables to describe position and orientation of the gripper in a very easy way.

The software package contains two global modules: translator and interpreter. The translator reads the source code written in PASRO and translates it into intermediate code which is, then, executed by the interpreter. The source code, written by user contains the description of the robot's task. When writing the source code, user is not expected to take care about the particular robot which is supposed to execute the task. That problem is resolved by the interpreter which will execute the code on the particular robotics system.

The translator, also, contains the complete users interface (manu system, multi-text editor, help system) providing the user with an friendly environment for writing the programs and using the software package itself.

The results achieved in this paper are evaluated on the PUMA configuration robot developed at the Robotics Department of the Institute "Mihajlo Pupin" Belgrade.

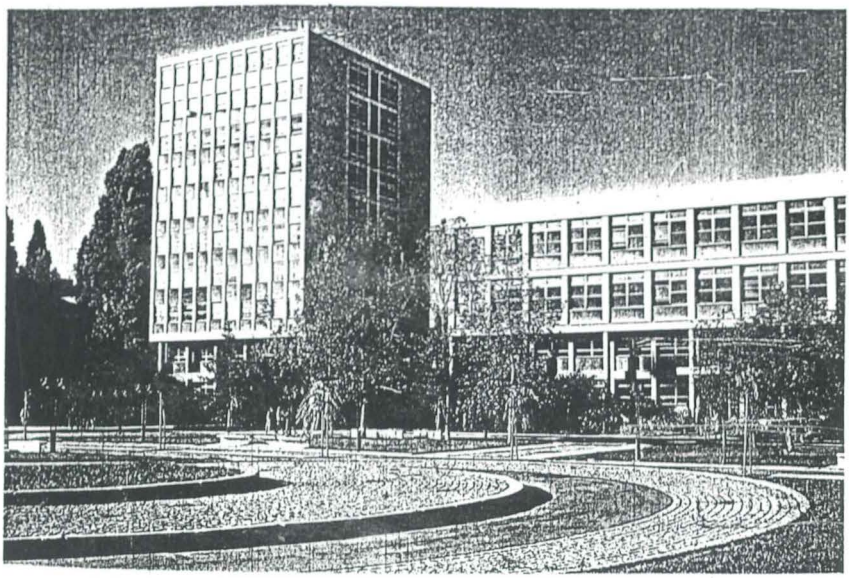

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCE
INSTITUTE OF MATHEMATICS

# Sponsors of the VI Conference on Logic and Computer Science

LIRA '92

Novi Sad
October 29-31, 1992

UNIVERSITY OF NOVI SAD
Trg Dositeja Obradovića 5
21000 Novi Sad
Tel: (+38 21) 350-622, 350-760, 350-163
Fax: (+38 21) 611-725
E-mail: rektorat%uns@yubgef51.bitnet



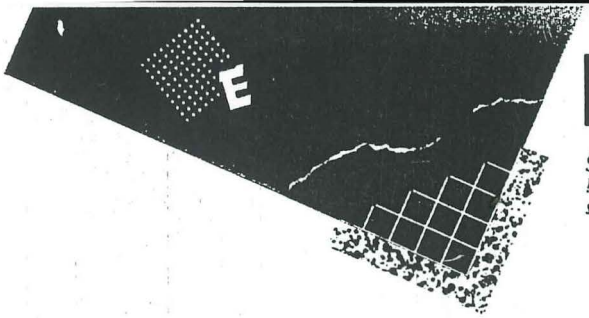
THE UNIVERSITY OF NOVI SAD (FOUNDED IN 1960) COMPRISES TWELVE FACULTIES LOCATED IN:

Novi Sad
FACULTY OF PHILOSOPHY
FACULTY OF AGRICULTURE
FACULTY OF TECHNOLOGY
FACULTY OF LAW
FACULTY OF ENGINEERING SCIENCES
FACULTY OF SCIENCE
FACULTY OF MEDICINE
ACADEMY OF ART
FACULTY OF PHYSICAL EDUCATION

Subotica
FACULTY OF ECONOMICS
FACULTY OF CIVIL ENGINEERING

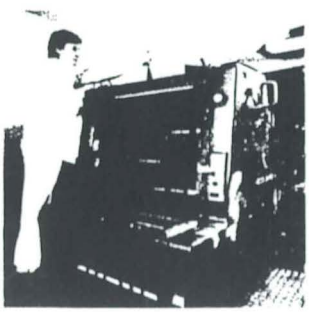
Zrenjanin
FACULTY OF ENGINEERING "MIHAJLO PUPIN"

The University of Novi Sad is the well-known university centre with good reputation in education of young people for different professional sectors. It has over 20.000 thousand students every year. It is also successful in scientific and research work by realization of a considerable number of projects, participation in professional and scientific conferences, publication of scientific results, etc. All its educational, scientific, cultural, artistic, sports and extension activities are supported and performed by some 2.000 professors, associate professors, assistants and other university staff.
For years the University of Novi Sad has been developing successful and diversified international university cooperation in all sectors of its competence.
The University of Novi Sad with its programmes and activities is completely oriented towards the future, development and prosperity.



**EFEKT**

*ŠTAMPARSKO GRAFIČKO IZDAVAČKO*
*MEŠOVITO DRUŠTVO EXPORT-IMPORT*
*sa ograničenom odgovornošću sa p. o. Beočin*



DOBITNIK PLAKETE SREBRNI PEČAT
I MAKARIJEVE NAGRADE
NA XII MEDJUNARODNOM SAJMU GRAFIČKE
I PAPIRNE INDUSTRIJE 1990. GODINE

21300 BEOČIN Dunavska bb Yugoslavia Tel:021/461-096, 461-370, 51-867, 51-349 Fax:021/461-096, 51-349 Br.žiro rač. 65700-603-8334 SDK Novi Sad

## ELEKTROVOJVODINA
## PUBLIC COMPANY FOR ELECTRIC DISTRIBUTION
## NOVI SAD

The Elektrovojvodina supplies high quality electrical power for citizens and companies in the whole territory of the Autonomous Province of Vojvodina. It also deals with design, construction and maintenance of the electrical power facilities and plants.

It was founded on June 28, 1958, by integration of few electric distribution companies that at that time existed in Vojvodina.

The Status of the Public Enterprize it was given by the Decision of the Assembly of the SAP Vojvodina in January, 1990, and from the beginning of 1992 it is within the united electric industry of the Republic - Public Corporation "Elektoprivreda Srbije".

The Elektrovojvodina is now composed of 13 companies and the Administration. The headquarters of the electric distribution are in Sombor, Vrbas, Subotica, Senta, Kikinda, Zrenjanin, Pančevo, Ruma, Sremska Mitrovica and Novi Sad.

In addition to regular distribution of electrical power, The Elektrovojvodina to its consumers provides service in balancing single-tariff and double-tariff counters, single-phase and three-phase meters.

To the industrial consumers of the electrical energy in Vojvodina, it also maintains the electrical power facilities and plants.

To the household consumers the Elektrovojvodina can give services in high quality maintenance of main receptacle power lines at very competitive prices.

The teams of experts in the company for more than three decades also successfully perform specialized works in design, development and maintenance of information systems for requirements of the Elektrovojvodina and interested partners. The computer experts of Elektrovojvodina actually initiated the acceptance of sponsorship for the LIRA '92 Conference.

Telephone: +3821 621-222
Telefax: +3821 23-470
Telex: YU elevoj 14188

# Sojaprotein®

"Sojaprotein" is a significant and very modern soybean processor worldwide, regarding its production capacity, advanced technology, wide range and quality of the products. It is in possibility to provide necessary soy products for food industry, mass consumption and animal feeding. Annual processing capacity is 200.000 t of soybeans. Factory was finished in 1983 and disposes with big store house capacities for soybean storing, as well as oil and protein processing complexes.

The up-to-date equipment installed in the plant is constantly renewed and expanded, following the latest developments of the world's technology in this field.

By soybean processing a numerous fullfat and defatted products are obtained, which have a wide range of application in the industrial food production: meat industry, bakery, pasta industry, dietetics and pharmaceutics, catering industry and households. Soja-Vita products are used in family consumption.

By using soy products in animal feeding, better results are obtained in the production of meat, milk and eggs.

Factory is located in the plain part of country, in agricultural region, thus providing the possibility to obtain the bigger necessity for raw material from it's surroundings. The rest part of row material is from import.