Transactions of the Institute for Informatics and Automation Problems of NAS RA.
Collected reports of participants of JAF-23.

A POLYNOMIAL PROGRAMMING LANGUAGE

Anatoly P. Beltiukov

A programming language (similar to Pascal) is proposed in which only polynomial time computable functions can be programmed. More precisely: any program in the proposed language, computing a function of words in some finite alphabet (with word values), can be executed by the standard interpreted Turing machine in the time bounded by some polynomial of the length of the input word. It is convenient to use this language for programming realizations of formulas in weak arithmetic theories, that can prove only formulas with polynomially computable realizations.

§1. INTRODUCTION

The main purpose of this work is to build a language that is convenient for defining effective (polynomial time computable) realizations of constructively understood formulas. This language can be used for example to ease proving results that are similar to [1-3].

We construct a language that is similar to Pascal. It is allowed to use natural numbers and functions in the proposed language. Arguments and values of the functions may be not only numbers but functions also. All arguments must be precisely specified (including arguments of functional arguments and so on). Recursion is not allowed. Values ranges must be specified for all variables. Nevertheless range frames may be computed while entering corresponding routines. It means that range frames may be defined by expressions. Basic operations in expressions are addition, subtraction, multiplication, integral division. Defined functions may also be used in expressions. Variable arrays are allowed (with computed indices ranges). Arrays may also be arguments and values of functions. In this case array index range may depend on previous arguments.

The simplest language statement is an assignment operator. The function value is defined by the expression at the end of the function definition body. Conditional statement may also be used. Conditions are comparisons of numeric expressions („greater"

or „equal"). Loops are allowed to be only „for"-type without changing the loop variable within the loop body.

Words of a finite alphabet are simulated by arrays. A word processing function has two arguments: the word length and the vector of the word letters numeric codes. The function value is an array with the result length in the first element. Note that the result length is estimated by a polynomial of the argument length. Exceding data cells in the output array should be ignored.

For programming formulas realizations it is convenient to add records data types to the language. Of course, the language may be upgraded in many other convenient ways.

§2. DEFINITION OF POLYNOMIAL PROGRAMMING LANGUAGE (PPL)

First of all we describe the context free syntax of the language to be built. Below we define the notion $< type >$ that means „expression which value is a type values":

$< type >::=$

$natural | < vartype > |$

$function(< name >:< type >) < type >$

Here „natural" denotes the set of all natural numbers without any limitations: 0, 1, 2,... to infinity. Of course, it is not convenient to have a variable of this type because we do not know how many storage space it will occupy (in any case it is not good from computational complexity point of view). Therefore we have special data type class for variable values. It is denoted $< vartype >$ above. Of course, it would be very hard to estimate computational complexity if we had variables with functional (algorithmic) types. Therefore function types in the definition above are separated from the $< vartype >$. The notion $< vartype >$ („type of variables" or type of stored data) is defined as follows:

$< vartype >::=$

$0.. < term > |$

$array[0.. < term >]of < vartype >$

Here $< term >$ is an expression that is defined below. The values of this expression should be natural numbers. To „compute" the types defined here we should substitute this numbers instead of the terms. The type $0..n$ means the set of all natural numbers from 0 to n. The type $array[0..n]t$ means the set of all finite functions that map the

set $0..n$ into the set t. Any expression has its own type. The types of the terms in the previous rule are „narural" or have a form $0..a$. Here is the definition of the $< term >$ notion:

$< term >::=< name > | < digits > |$
$(< term >< op >< term >)|$
$< term > [< term >]|$
$< term > (< term >)|$
$(if < term >< rel >< term >$
$then < term >$
$else < term >)|$
$program(< name >:< type >) < type >;$
$< definitions >$
$begin < sequence >< term > end$
$< op >::= +|subt| * |div$
$< rel >::=< | > | =$

Here $+$ and $*$ are usual addition and multiplication of natural numbers correspondingly. The value of the term $(a \ \textbf{subt} \ b)$ is $a - b$ if a is greater than b and it is 0 otherwise. The value of the term $(a \ \textbf{div} \ b)$ is the integral part of a/b. If b is 0, then a program computing $(a \ \textbf{div} \ b)$ ends abnormally. The value of $a[i]$ is the i -th element of the array a. If the array a has no i -th element, the program ends abnormally. Computing the value $f(a)$ is executing the body of the algorithm f with a as the value of the argument of the algorithm. The body of an algorthm

program $(a : t)$ u; d begin s e end

is the part s e, where s is a sequence and e is a term. To compute this part we should firstly execute the sequence s and then we have to compute the expression e obtaining the value of the computed function. The argument of this algorithm is the name a. The definitions d of the algorithm can be described according to the following rule:

$< definitions >::= |$
$var < name >:< vartype >;$
$< definitions >$

The first part of the algorithm body („sequence") is defined by the following rules:

< *sequence* >::= |

< *statement* >; < *sequence* >

< *statement* >::= |

< *variable* >:=< *term* > |

if < *term* >< *rel* >< *term* >

then < *statement* >

else < *statement* > |

for < *name* >: 0.. < *term* >

do < *statement* > |

begin < *sequence* >< *statement* > *end*

< *variable* >::=< *name* > |

< *variable* > [< *term* >]

The notion < *variable* > corresponds to expression that defines a piece of the computer storage that can accept some value. In the simpliest version of the language that we are describing now it corresponds to a name or to an array element. Conditional expressions

if c then a else b

are executed in the usual way. Executing a loop

for i : 0..*n do s*

is executing n and then executing s for $i = 0, 1, ..., n$ subsequently.

§3. CONTEXT CONSTRAINTS

Name definition is its occurence in a construction of one of the following forms:

var < *name* >:< *type* >;

function(< *name* >:< *type* >)

program(< *name* >:< *type* >)

for < *name* >: 0.. < *term* > *do*

In the first case the name is defined in the body of the defined algorithm. In the second case the name is defined in the type of the function (for example: ***function***(*n*:***natural***)0..*n*). In the third case the name is defined in the type of the defined algorithm, in its variable definitions, and in its body. In the last case the name is defined in the body of the loop.

Any terms that do not use undefined names are called „PPL-programs". Note that PPL-programs cannot contain recursive function due to the accepted syntax description.

All the used names should be defined with appropriate types.

Types of array indices (i in $a[i]$), types of boundaries (n in $0..n$), types of arithmetical operations ($+$, ***subt***, $*$, ***div***) and relations ($<, >, =$) should be natural or of the form $0..a$.

Types of right parts of assignments (e in $v := e$) should correspond to the types of the variables in the left parts (v in $v := e$). Note, that when an array index (or a variable value) is out of the allowed range, the program stops abnormally.

The type of a function argument must correspond to the function type.

The type of the returned value must correspond to the type of the computed function.

Loop value names (i in ***for*** $i : 0..n$ ***do*** s) are not variables and therefore they are not allowed to be assigned in the body of the loop (s). Program parameters, a in

program$(a : t)u; d$ *begins eend*

are also not variables and may not be assigned in s.

For convinience we require functions to have no side effect: variables that are defined outside the function definition may mot be assigned in the function body.

§4. A PROGRAM EXAMPLE

Here we show a short example of wellknown sort program (a version of „bubblesort"):

```
program(n : natural)
  function(f : function (k : 0..n) natural)
    function (k : 0..n) natural;
  begin
  program (f : function (k : 0..n) natural)
    function (k : 0..n) natural;
    var b : 0..n;
    var a : array[0..n] of 0..n;
    begin for i : 0..n do a[i] := i;
      for i : 0..(n subt 1) do
       for j : 0..(n subt (i + 1)) do
        if f(a[j]) > f(a[(j + 1)])
          then begin b := a[j];
                     a[j] := a[(j + 1)];
```

```
                a[(j + 1)] := b
              end
          else;
      program (k : 0..n) natural;
      begin f(a[k]) end
    end
  end
```

This program processes a pair: a number n and a natural function f, that is defined on the numbers $0,...,n$. Note that in the defined language we use a function with a functional value to express a function with two ore more arguments ($f(x,y) = f(x)(y)$, i.e. $f(x) = g$, $f(x,y) = g(y)$). The result of the program is a function

program $(k : 0..n)$ *natural*; *begin* $f(a[k])$ *end*

that has precisely all values of f but in nondecreasing order. To compute this function the program generates a substitutional array a using common „bubblesort"-like routine:

```
for i : 0..(n subt 1) do
 for j : 0..(n subt (i + 1)) do
  if f(a[j]) > f(a[(j + 1)])
   then begin b := a[j];
              a[j] := a[(j + 1)];
              a[(j + 1)] := b
        end
   else;
```

§5. WORD FUNCTION COMPUTATION DEFINITION

Let S be a finite alphabet that consists of $d + 1$ letters. Let us consider a function f that maps all (nonempty) words in S into the set 0,1. To compute f we will use a program of the following form:

```
program (n : natural)
   function (w : array[0..n] of 0..d) 0..1;
begin
...
end
```

Here the pair (n, w) describes an input word, n is its length, and w is its content. The final value of the computed function is 0 or 1. For example, the following program recognizes palindromes in the alphabet 0,1:

```
program (n : natural)
  function (w : array[0..n] of 0..1) 0..1;
begin
  program (w : array[0..n] of 0..1) 0..1;
  var r : 0..1;
  begin
    r := 1;
      for i : 0..(n div 2) do
        if w[i] < w[(n subt i)] then r := 0;
          else if w[i] > w[(n subt i)] then r := 0; else;
    r
  end
end
```

§6. THE MAIN RESULTS

Note that we cannot compute an exponential function by a PPL-program. For example, the construction

```
var x : 1..b;
...
x := 1;
for i : 0..n do
  x := 2 * x
```

fails when the value of x becomes greater than the value of b. We cannot define „unbounded" variable to use a similar construction for exponentiation. More precisely this fact can be expressed in the following theorem:

Theorem 1. All word functions computable by PPL-programs are polynomial time computable.

Proof: The proof idea is to exclude all intermediate functions and estimate numbers of steps in the loops. To excude function call it is convenient to add new syntax construction

extending the notion < *term* > by the following „block":

< *term* >::= (< *definitions* > *begin* < *sequence* > < *term* > *end*)

with obvious sense. We can use these blocks to replace function calls of the form:

program (< *name* >: < *type* >) < *type* >;

< *definitions* >

begin < *sequence* > < *term* > *end*

(< *term* >)

because the last < *term* > can be substituted instead of the < *name* > occurencas in the program body. The process of these substitutions is finite because all functions have finite types and this operation decreases these types. After this process being completed the program will consist of nested loops with polynomial bounds that gives desirable computing time estimation.

Theorem 2. Any polynomial time computable word function can be programmed in the proposed language.

Proof: The proof idea is the following. First of all, estimate the space and the time that are required to compute the given function by polynomials of the input array length. After that one can easily simulate the Turing machine that computes the function by a PPL-program with a single work array (that simulates the machine tape), two variables (simulating the head position and the inner machine state correspondingly) and a single „for"-loop (to perform the computation process) using directly the definition of Turing machine computation.

§7 CONCLUSION

We can add some new data types to the language: structures (records, direct products), unions (variant records, direct sums).

The proposed language is supposed to use for studying deductive systems of constructive weak arithmetics. This language fits for constructing „polynomial time computable realizations" for systems of weak arithmetics with various induction schemes, such as in [1-3].

REFERENCES

1. A. P. Beltiukov, Intuitionistic formal theories with realizability in subrecursive

classes // Annals of Pure and Applied Logic, 89, 1997, p. 3-15.

2. A. P. Beltiukov, A strong induction scheme that leads to polynomially computable realizations // Theoretical Computer Science, 322 (2004) 17-39

3. A. P. Beltiukov, A Weak Constructive Second-Order Arithmetic with Extraction of Algorithms Computable in Polynomial Time // Journal of Mathematical Sciences. Publisher: Consultants Bureau, An Imprint of Springer Verlag New York LLC. ISSN: 1072-3374 (Paper), 1573-8795 (Online). DOI: 10.1007/s10958-005-0351-4. Issue: Volume 130, Number 2. Date: October 2005 Pages: 4571 - 4573

12 October 2005

Udmurt University, Izhevsk, Russia
E-mail: belt@uni.udm.ru