Aplikace matematiky

Karel Čulík

A note on complexity of algorithmic nets without cycles

Aplikace matematiky, Vol. 16 (1971), No. 4, 297–301

Persistent URL: http://dml.cz/dmlcz/103359

A NOTE ON COMPLEXITY OF ALGORITHMIC NETS WITHOUT CYCLES

KAREL ČULÍK

(Received March 26, 1971)

This note is an appendix to [2] and presents, among others, its comparison with [5] and [1]. Further a generalized algorithm is sketched out by which the minimal number of addresses (or registers) required by computation of a special simple program is determined. All notations and terminology are used in accordance with [2].

A. By each simple program as many functions as there are output addresses are computed (see [3]), but it should be stressed that at two different output addresses one and the same function may be computed (and, moreover, this function is computed by the same *algorithmic method* [4] in both cases). There arises a dilemma for the user of such a program: either he does not allow to change his program i.e. all the output addresses must be preserved, or he allows to simplify it in order to avoid possible superfluous repetitions during the computation (obviously together with an information about all "*equivalent*" = "*the same results giving*" output addresses); for this case the necessary preprocessing has been described in [2] (i.e. for a program its algorithmic net is constructed, then the algorithm of unification leading to a simple net is used, and finally a course of it is chosen). In this way an *equivalency relation* in the set of all output addresses (or of all output vertices of the corresponding net) is determined. Only with respect to this equivalency relation we may speak about the *set of functions computed by a program* (or by a net), i.e. if the computed functions differ from each other.

It may be proved that two simple programs compute the same set of functions (with respect to the above mentioned equivalency relation) in each of possible computers (i.e. for each of possible choices of ***Obj*** and ***Fct***) if and only if they are structurally equivalent. To have this theorem is the main reason why the restriction (2.1) – which says that different input vertices must be labelled by different input addresses – has been accepted, and why never any use of special properties of functions from ***Fct*** (as commutativity, associativity, etc.) has been made.

Another reason for acceptance of (2.1), close to the mentioned one, is the require-

ment of being able to interprete *the number of input vertices* as *the number of input addresses* of the corresponding program, and also as *the number of variables* of the set of computed functions: we assume that each function has a fixed number of variables, i.e. has its *arity*, independently as to whether some of them are *singulary* or not (as it is very frequent with boolean functions in logic, e.g. in $a \vee \bar{a}$, and sometimes also in arithmetics, e.g. in $a - a$, a/a, the variable a is singulary etc.).

B. In [5] a special type of algorithmic net from [2] is investigated where the following restrictions are accepted: (i) $\mathrm{id}(x) = 2$ for each non-input vertex x; (ii) $\mathrm{od}(x) = 1$ for each non-output vertex x and (iii) there exists only one output vertex which is usually called the root. By (ii) it follows: (iv) there are no parallel paths and therefore such a net may be called an *algorithmic binary rooted tree.* The second labelling λ_E of [2] is replaced by new partial ordering of vertices in planar figures from the left to the right. According to (2.1) only special expressions correspond to the algorithmic binary rooted trees, namely those in which no variable occurs more than once; e.g. the expression $a/(b + c) - a * (b + c)$ is not considered but $a/(b + c) - d * (e + f)$ is, which is connected with the type of instructions allowed in [5]. In [2] is assumed, as a fact of greatest importance that the first expression just mentioned may be computed more efficiently than the second because of certain repetition of variables and even subexpressions. Moreover in [2] both expressions are considered absolutly independent of each other although the first arises from the second by certain identification of its variables and the same is true for their corresponding functions (see ***A.***).

In order to be able to compare [5] and [2] let $\boldsymbol{Adr} = \boldsymbol{Reg} \cup \boldsymbol{Sto}$ where $\boldsymbol{Reg} \cap \cap \boldsymbol{Sto} = \emptyset$ and the set ***Reg***, ***Sto*** contains the addresses of registers, of storage cells respectively. Thus two sorts of memory elements are distinguished.

In [5] the following four types of instructions (or assignement statements) are considered: 1. $s =: r$, 2. $r =: s$, 3. $f(r, s) =: r^*$, 4. $f(r, r^*) =: r^{**}$ where $s \in \boldsymbol{Sto}$ and $r, r^*, r^{**} \in \boldsymbol{Reg}$ and $f \in \boldsymbol{Fct}$ ("f" $\in \boldsymbol{Opr}$). A minimalization of number of registers (or of register addresses) required by a computation of an expression represents a search among the corresponding programs which do not contain instructions of type 2. If, however, instructions of type 2. are not admitted, then the importance of instructions of type 3. decreases considerably because they may be used only for leaves of a tree and never within the tree itself. Therefore if the instructions of type 3. are not allowed (or if there is another prescription for the minimal number or registers required for leaves, which is possible in general case of n – ary functions in ***Fct***) then the required minimal number does not differ very much of that with instructions of type 3.

The instructions of type 1. are mostly time-wasting and therefore the minimalization of their number is of greater importance than that of the number of registers itself. This is the point of view in [2] where the best possible case is assumed (if in the expression some variables may occur many times), namely *the minimal number*

of instructions of type 1. is assumed which is the number of different input addresses (and simultaneously the number of input vertices, as mentioned in A.).

On the other hand, in [5] the worst possible case is assumed namely the maximal number of instructions of type 1. being equal to the number of all occurrences of all input addresses in the program (and simultaneously to the number of all leaves of the tree). Here a much more decisive role will be played by the distinguishing of two sorts of instructions of type 3. and 4., namely: 3a) $f(r, s) =: r$, 3b) $f(r, s) =: r^*$, 4a) $f(r, r^*) =: r$, 4b) $f(r, r^*) =: r^{**}$, where it may, but need not, be $r = r^*$ and $r = r^{**}$.

If only types 3a) and 4a) are admitted which is unfortunately very often the case with real computers, then the point of view of [2] loses its importance because after the application of such an instruction the content of the first register address is always disturbed (and therefore it cannot be preserved in the same register for all the time when it is needed). In this case it seems to be desirable to pass to the next lower level of microcommands. There must always be stated before each application of a microcommand whether or not the content should be preserved, etc.

C. In [2] only instructions of type 4b) are admitted because (tacitly in accordance with [6]) the instructions of type 1. are considered as input instructions and therefore they are not confused with other operational types, i.e. it is assumed that the *actual program* is preceded by its *input program* of the type $(4 =: a,\ 5 =: b,\ 2 =: c)$ for the simple program $(b + c =: x,\ a/x =: y,\ a * x =: x,\ y - x =: x)$ of the expression mentioned above (in [6] such programs are called *standard*).

Two sorts of memory elements allow an other point of view (in [6] the corresponding programs are called *non-standard*) illustrated by the following example: $(5 =: b,\ 2 =: c,\ b + c =: b,\ 4 =: c,\ c/b =: a,\ c * b =: b,\ a - b =: a)$. Here 3 addresses suffice, but above at least 5 of them are required. Thus the input data are coming consecutively and not at once.

Therefore let us slightly modify the definitions of scopes in a course $(v_1, v_2, \ldots, v_n)$ of [2] as follows: the scope of an *input vertex* v_i is defined in the same way as of any *non-ouput vertex*, i.e. $\boldsymbol{Sc}(v_i) = (v_{i+1}, \ldots, v_j)$ where j is the maximal integer such that $(v_i, v_j) \in \varrho$ and $\langle V, \varrho \rangle$ is the corresponding net; further let $\boldsymbol{Sc}(v_i) = \emptyset$ for each output vertex v_i. This last change allows to have all results on one and the same address but, of course, consecutively in time (after each occurence of an output address x an ouput command "*print*(x)" should be added).

According to [2] this new definitions of scopes lead to a new concept of *scope width of a course* $P = (v_1, v_2, \ldots, v_n)$ which will be denoted by $scwi^*(P) = \max_{1 \leq i \leq n} scwi^*(v_i)$, and to a definition of scope width of a net N which will be denoted by $scwi^*(N) = \min_{P \in \boldsymbol{P}(N)} scwi^*(P)$ where $\boldsymbol{P}(N)$ is the set of all courses of N. This allows a formulation of the following conjecture: if T is an algorithmic binary rooted tree

then $scwi^*(T)$ is the minimal number of registers required for computation of the expression corresponding to the tree T.

The following assertions may be proved about algorithmic rooted trees, i.e. if it is allowed to have instructions $f(r_1, r_2, \ldots, r_n) =: r_0$ where $r_i \in \textbf{\textit{Reg}}$ for $i = 0, 1, \ldots, n$:

Lemma. *Let be given integers k_i such that $k_1 \geqq k_2 \geqq \ldots \geqq k_n$ where $n \geqq 1$. Then an integer m satisfies the inequalities* (1) *and is as small as possible, if and only if* (2) *where*

$$m \geqq k_1,\ m - 1 \geqq k_2,\ \ldots,\ m - (n-1) \geqq k_n\,, \tag{1}$$

and

$$m = \max_{1 \leqq q \leqq p} [k_{s_q} + s_q - 1] \tag{2}$$

where p is the number of different integers k_i and the integers $s_1, s_2, \ldots, s_p = n$ are determined by the following inequalities

$$k_1 = \ldots = k_{s_1} > k_{s_1+1} = \ldots = k_{s_2} > \ldots > k_{s_{p-1}+1} = \ldots = k_{s_p}\,. \tag{3}$$

Theorem. *Let T be an algorithmic rooted tree with the root v and let $v_1, v_2, \ldots, v_n$ ($n \geqq 1$) be all the vertices such that each edge terminating in the root v must start in one of them. If T_i is the algorithmic rooted subtree of T with the root v_i and if we put $k_i = scwi^*(T_i)$ for $i = 1, 2, \ldots, n$, then $scwi^*(T) = m$ where m is determined by* (2) *and* (3).

One easily sees that if $n = 2$ which is the case for binary rooted trees, then (3) gives the same value as required in [5].

Therefore the number $scwi^*(v)$ is defined for aech vertex of an algorithmic rooted tree T and it is easy to describe simple algorithm by which a course $(v_1, v_2, \ldots, v_N)$ of T is constructed such that $scwi^*(v_i) \leqq scwi^*(v_j)$ for all $i < j$ where $i, j = 1, 2, \ldots, N$.

In this way a special case of a modification of the problem 2 of [2] is solved.

D. In [1] the ideas of Z. Pawlak concerning his addressfree computers are fomalized. Here also only algorithmic binary rooted trees are considered. In [1] a completely different structure of memory than in [2] or [5] is assumed, i.e. a push-down store is required. However there is a conjecture that the minimal deep of the push-down store required for computation of the tree T 's $scwi^*(T)$ again.

References

[1] *Blikle, A.:* Addressless units for carrying out loop-free computations, pp. 48, Polish Academy of Sciences, Institute of Mathematics, July 1970.

[2] *Čulík, K.:* Combinatorial problems in theory of complexity of algorithmic nets without cycles for simple computers, Aplikace matematiky 16 (1971), 188—202.

[3] *Čulík, K.:* On semantics of programming languages, Automatentheorie und formale Sprachen ed. J. Dörr, G. Hotz, Bibliographisches Institut, Mannheim—Wien—Zürich, 1970, pp. 291—303.

[4] *Čulík, K.:* Structural similarity of programs and some concepts of algorithmic metod, preprint of a lecture presented at the GI — Conference in Munich, March 1971.

[5] *Sethi, R., Ullman, J. D.:* The generation of optimal code for arithmetic expressions, Jour. of ACM, vol. 17, No. 4, October 1970, pp. 715—728.

[6] *Čulík, K.:* Theory of programming languages and of algorithms, a textbook for Institute of Technology in Prague (in print), July 1970.

Souhrn

POZNÁMKA O SLOŽITOSTI ALGORITMICKÝCH SÍTÍ BEZ CYKLŮ

KAREL ČULÍK

Rozsahovou šířkou algoritmické sítě bez cyklů N se rozumí číslo $scwi^*(N) = \min_{P \in \boldsymbol{P}(N)} scwi^*(P)$, kde $scwi^*(P)$ je modifikovaná rozsahová šířka průběhu P sítě N a $\boldsymbol{P}(N)$ je množina všech průběhů sítě N. Je-li T algoritmický kořenový strom (tj. síť s jedním výstupem a bez rovnoběžných sledů) s kořenem v a jestliže $v_1, v_2, \ldots, v_n$ jsou všechny uzly stromu T, z nichž vedou hrany do v, a jestliže položíme $k_i = scwi^*(T_i)$ pro $i = 1, 2, \ldots, n$, potom je vyslovena domněnka, že $scwi^*(T) = m$, kde m je určeno podle (2) a (3).

Author's address: Prof. Dr. *Karel Čulík*, DrSc., Výzkumný ústav matematických strojů, Lužná ul., Praha 6-Vokovice.