数理解析研究所講究録
556巻 1985年 319-326

THE COMPLEXITY OF SUBSTITUTIVE PROGRAMS

Takeo Yaku, Dept. Math. Sci., Tokai Univ.,
Akeo Adachi, Sci. Inst., IBM Japan,
and Kokichi Futatsugi, Comput. Sci. Div., Electrotech. Lab.

ABSTRACT

A control strucure called "substitution" is introduced, which generates a balanced tree of the substitution module. The function description ability of substitutive programs is investigated in terms of the maximum depth of the nest for substitution statements.

We obtain an interesting equivalence between looping (chain generating) plus (loop) nesting, and substituting (balanced tree generating) without nesting. The result provides a new and intuitive characterization of the Kalmar's elementary functions.

1. INTRODUCTION.

Control structures of a program is one of the essential issues of the program theory. Because both theoretical and software engineering properties of programs depend on them.

Sequential control structures (see e.g., [6]) such as "for do", "do while", "loop exit" and "loop exit(i)", and recursive structures such as linear recursion and binary recursion [8] are well known examples. They have been introduced in order to increase the ability of design and analysis of programs from software engineering requirements.

Meanwhile, the 2-dimensional graphical representation of programs has been employed since Goldstein-Neumann flowcharts. In 1970's several structured flowcharts were proposed, corresponding to development of structured programming methods. Most of them

represent program modular structures by nesting structures of charts, in which only the vertical locations of cells have meaning, but the horizonal location does not any have meaning.

In 1978, we proposed a hierarchical flowchrt language called "Hichart" [9]. In a Hichart flowchart, the sequential order of statement execution is represented by vertical order of cells, but the hierarchical level of statement corresponds to horizonal distance from the root cell of the flowchart. In these considerations, sequential control structures are considered to control vertical repetition of the repetition (iteration) scope.

We introduced another class of control structures, called "substitution", that control the horizonal repetition of the repetition (substitution) scope in Hichart type flowcharts. The substitution is implemented by the "subst" statement. A program with substituion statements is called a "subst" program or, informally, a "substitutive" program. A substitutive program has several superior software engineering characteristics such as "visual", "easy to estimate" the complexities, "easy to understand", etc.

"Substitution" is an extension of the iteration, because a subst program is executed as an in-order traversal of a balanced tree of the substitution scope, as the limited "recursion" is, but an iteration program is executed as only a sequence of the iteration scope. However, the function description ability of the substitution programs is just limited to the primitive recursive functions. Therefore, "substitution" is considered to lie between iteration and recursion.

2. SUBSTITUTIVE PROGRAMS.

We first define a "subst program", which is a generalization of a static recall program [10].

Definition 1. Let X and S be fixed mutually disjoint countable sets of symbols. An element of X or S is called a control variable or a simple variable, respectively. Let Var denote the all variables, that is, Var = X U S . A subst program is a sequence of statements over Var defined recursively as follows:

<atomic statement>::= u := u+1 | u := u-1 | u := 0 | u := v
<subst statement>::= **subst** <subst control> **do begin**
<statement list> **end**
<subst control> ::= x-k | x
<statement> ::= <atomic statement> | <subst statement> | **resubst**
<statement list>::= <statement> | <statement>;<statement list>
| null
<subst program>::= **begin** <statement list> **end.**,

where u and v are in Var, x is in X, null denotes the empty sequence of a statement, and k is an nonnegative integer.

The computation of a subst program is defined below. Let s_i (1 < i =< N) be either a null or a list of statements other than **resubst** (If s_i is null, then an associated semicolon ; is eliminated). A subst statement of the form

subst x-k **do begin** s_1;**resubst**;s_2;**resubst**; ... ; **resubst**;s_N
end (2.1).

is equal to one of the followings:

(i) null (x-k = 0),

(ii) null (x-k > 0 and N <= 1, i.e., with no **resubst** statement), or

(iii) s_1; **subst** x-k-1 **do begin** Q **end** ;
s_2; **subst** x-k-1 **do begin** Q **end** ;
...
s_{N-1}; **subst** x-k-1 **do begin** Q **end** ;
s_N, (x-k > 0 and N > 1), (2.2),

where Q = s_1; **resubst**; s_2; **resubst**; ... ; **resubst**; s_N. We say (2.1) is expanded to (2.1).

A loop statement, a loop program and the computation of a loop program are defined as in [7]. Proposition 1 and Example 1 below show a simulation mechanism of a loop statement by a subst statement. Accordingly, a subst statement is regarded as an extension of a loop statement.

Proposition 1. A loop statement "loop n **do begin** P **end**" is represented by a subst statement "**subst** n **do begin resubst**;P **end**" which is expanded to :

subst n-1 **do begin resubst**; P **end**; P
= ... = P ; P ; ... ; P (n time repetition).

3. THE COMPLEXITY.

This section deals with correspondence relations beween classes of subst and loop programs with respect to the function description ability and to the maximum depth of nesting of loop or subst statements, under a relation "mutual bounding" defined later.

Notation. Analogously to a loop(i) program [2], a subst(i) program denotes a subst program having the maximum depth i of the nesting for a **subst end** pairs. $\mathbf{S}_i$ denotes the class of functions computed by subst(i) programs. $\mathbf{L}_i$ denotes the class of functions computed by loop(i) programs [1]. And

$$\mathbf{L} = \mathrm{U}_i\ \mathbf{L}_i$$
$$\mathbf{S} = \mathrm{U}_i\ \mathbf{S}_i .$$

The function tree(k,n) denotes the number of nodes in a k-ary balanced tree with the depth n, which is defined by :

$$\mathrm{tree}(k, n) = \Sigma_i\ k^i$$

For a program P, f(P) denotes the function computed by P.

Definition 2 . A class C of functions is bounded by a class **D** of functions, denoted by $C <=_b D$, if for any function f in **C** there exists a function g in **D** such that f(n) <= g(n) for any n. If $C <=_b D$ and $D <=_b C$, we say C and D are mutually bounded, denoted by $C =_b D$.

The following theorem provides an interesting equivalence between "the looping plus the loop nesting" and "the balanced tree generating". It also gives a new and intuitive characterization of the Kalmar's elementary functions.

Theorem 1. The class S_1 and the Kalmar's elementary functions are mutually bounded, that is, $S_1 =_b L_2$.

Proposition 2. $L_i <=_b S_i <=_b L_{2i}$.

Theorem 2. $S = L$.

Theorem 3. (The main theorem) $S_i =_b L_{i+1}$.

Theorem 3 showed that replacing sequence generating operators by tree generating operators is corresponding to augmenting the maximimum depth of nesting by one, but not two, of loop statements, that is, sequence generating operators. Theorem 3 is illustrated by the followng figure.

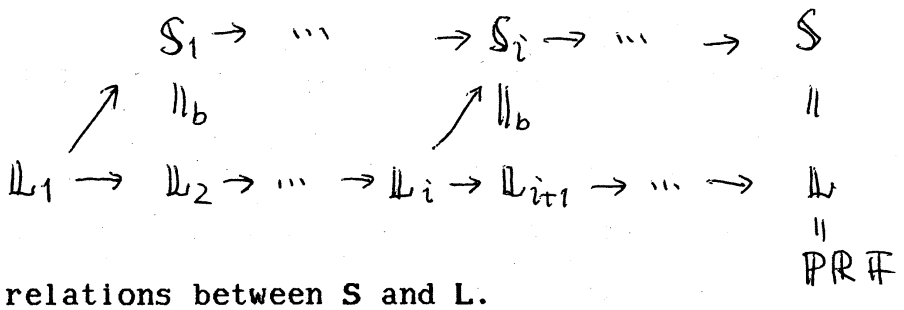


Figure 4. The relations between S and L.

REFERENCES.

1. A. Adachi, T. Kasai and E. Moriya, A theoretical study on the time analysis of programs, Lecture Notes in Computer Science 74 (1979), 201-209.
2. W. S. Brainerd and L. H. Landweber, "Theory of Computation", John Wiley and Sons, 1974.
3. A. Grzegorczyk, Some classes of recursive functions, Rozprawy Mathematyczne 4 (1953), 1-45 (Instytut Mathematyczny Polskey, Akademe Nauk, Warsaw, Poland).
4. L. Kalmar, Egyszery pelda eldonthetlen aritmetikai problemara, Mathematikai es fizika lapok 50 (1943), 1-23 (Hungarian with German abstract).
5. T. Kasai and A. Adachi, A characterization of simple loop programs, J. Comput. Systems Sci. 20 (1980), 1-17.
6. H. F. Ledgard and M. Marcotty, A genealogy of control structure, Commun. ACM 11 (1975), 629-639.
7. A. Meyer and D. M. Ritchie, The complexity of loop programs, Proc. 22nd ACM National Meeting (1967), 465-469.
8. H. R. Strong, Translating recursive equations into flowcharts, J. Comput. System Sci. 25(1971), 254-285.
9. T. Yaku and K. Futatsugi, Tree structured flowcharts, Inst. Electron. Commun. Engin. Japan, Report AL-78-47 (1978),61-66 (Japanese with English abstract).
10. T. Yaku, K. Futatsugi and A. Adachi, A control structure between iteration and recursion, Proc. Fac. Sci. Tokai Univ. 18 (1983), 119-129.
11. Y. Futamura, T. Kawai, H. Horikoshi and M. Tsutsumi, Development of computer programs by PAD (Problem Analysis Diagram), Proc. the fifth International Software Engineering Conference (1981), 325 - 332.

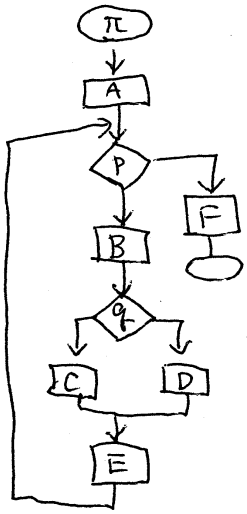
APPENDIX I. HICHART FLOWCHARTS

We introduce here an outline of the Hichart flowchart language [9,1978]. Hichart was designed to illustrate the structured program. Hichart is based on trees and hierarchically cyclic graphs for program modular structures.

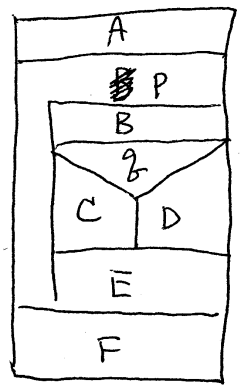
Hichart is the first language that illustrates the programs by hierarchical trees. Symbols and structures of Hichart have been employed by several authors in succeeding structured flowchart languages (e.g., PAD [11] and FESDD). Thus, several ten thousands of programmers presently use Hichart symbols and Hichart structures in several flowchart languages in Japan.

A Hichart flowchart is a labeled flowgraph written by the following convensions.

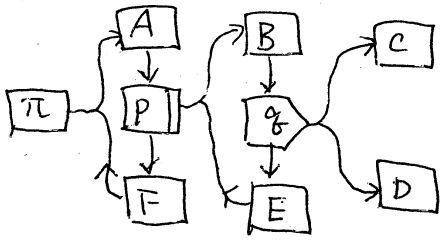
Examples. Examples of flowcharts by several flowchart languages.



Goldstein
-Neumann



NS chart



Hichart