数理解析研究所講究録
第 547 巻 1985 年 268-286

The BC-chain Method for Representing Combinators in Linear Space

野下浩平
Kohei Noshita
Department of Computer Science
Denkitusin University
Chofu, Tokyo 182

疋田輝雄
Teruo Hikita
Department of Mathematics
Tokyo Metropolitan University
Setagaya, Tokyo 158

Abstract

Turner's combinator implementation (1979) of functional programs requires the memory space of size $\Omega(n^2)$ in the worst case for translating given lambda expressions of length n to combinator graphs. In this paper a new idea named the BC-chain method for transferring actual arguments to variables is presented. We show that the BC-chain method requires only O(n) space for the translation. The basic idea is to group together into a single entity a sequence of combinators B, B', C and C', for a variable, which appear consecutively along a path in the combinator graph. We formulate two reduction algorithms in the new representation. The first algorithm naively simulates the original normal order reduction, while the second algorithm simulates it in constant time per unit operation of the original reduction. Another reduction method is also suggested, and a technique for practical implementation is briefly mentioned.

Key words and phrases

BC-chains, functional programming, lambda expressions, normal order reduction, space complexity, Turner combinators

1. Introduction

Turner [9] proposed in 1979 an elegant method for implementing functional programming languages by means of combinators. In this paper we investigate how efficiently we can implement them, particularly in terms of the space complexity.

Turner's combinator representation [9] requires the memory space of size $\Omega(n^2)$ in the worst case for translating given source lambda expressions of length n to combinator graphs [5], which may result in the runtime inefficiency, although the average size is at most $O(n^{3/2})$ under a certain definition of the average case [3]. A balancing transformation of a source expression enables the O(n) space translation, but this pretransformation requires $\Omega(n^2)$ space [2]. By extending the set of combinators, the translation can be done in O(n log n) space, while the reduction algorithm remains essentially unchanged from the original normal order reduction [7]. Kennaway and Sleep [6] has also shown the same O(n log n) result with the idea of "counter director strings."

All the works cited above can be regarded to assume Turner's combinators as a basis for representing combinator graphs of object programs. Now we also start our study on the same basis. (See [1] and [4] for different types of combinators.) We propose a new method for representing combinator graphs, which requires only O(n) space. We call this the BC-chain method. Our representation of combinator graphs is roughly described as follows. Each Turner combinator can be regarded to be responsible for exactly one variable, as it plays the role of a "director" at a node for passing actual arguments to the variable in a graph. If combinators B, B', C or C' appear

consecutively along a path in the graph and they are all responsible for a particular variable, we can efficiently encode those combinators into a single entity in a new combinator graph. The key idea to this encoding is to dissect a tree structure of combinators into chains of combinators only at S or S' nodes.

With the BC-chain method we can also expect an improvement of efficiency at the reduction stage. In this paper we present two reduction algorithms for our new representation. The first algorithm simulates the normal order reduction in a naive and faithful manner. The second algorithm is more efficient in the sense that it simulates the normal order reduction in constant time per unit operation of the original reduction. We also suggest another type of reduction algorithm for BC-chains.

In the next section we present a new representation, and we also sketch a translation method of functional programs to this representation. In Section 3, we prove that the complexity of this representation is linear in space. In Section 4, a naive reduction algorithm is given, and it is improved in Section 5, which simulates the original normal order reduction in constant time per unit operation. Finally, another reduction algorithm is suggested, and some implementation technique is also mentioned. For the sake of easier understanding, we present our BC-chain method in an informal way to some extent.

2. Combinator graphs with BC-chains

First we illustrate our new representation by a simple example. Let us consider an expression with a single variable x:

a(xb)(cx).

The ordinary abstraction ([8,9]) of this expression with the variable x by the combinators S, B, C and I is

S(Ba(CIb))(BcI).

This expression is represented in a graph (binary tree), as shown in Fig. 1 (a). Here note that for brevity we exclude the extentionality rule (η-rule) in the abstraction, since it can be easily incorporated into our method.

If we attach a combinator of either S, B or C at an interior node of the source graph, each combinator may be regarded to be a director for transferring an actual argument toward a corresponding variable. This is depicted in Fig. 1 (b), which is obviously equivalent to (a) in a suitable sense.

In this graph (b), let us focus on the combinators B and C (but not S) which appear consecutively along a path from the root to a leaf. We group this sequence of combinators into a single new node, which we will name a BC-chain. Thus we obtain a string

S([BC]a(Ib))([B]cI),

The graph representation is shown in Fig 1 (c). Here brackets "[" and "]" are used as a notational device to show a sequence of constituent combinators of a BC-chain. For convenience we shall freely omit the brackets if a BC-chain contains only one combinator, i.e., either [B] or [C].

In general, we construct a BC-chain from an expression in the original Turner's representation by the following rules:

Bab $\longrightarrow$ [B]ab

Cab $\longrightarrow$ [C]ab

Ba([w]bc) $\longrightarrow$ [Bw]a(bc)

$$C([w]ab)c \longrightarrow [Cw]abc$$

where [w] denotes a BC-chain, and a, b, c denote subexpressions. See Fig. 2 for the graph representation of the last two rules. We define the length of a BC-chain [w] to be the number of combinators in w. We naturally assume that, among possibly many BC-chains thus constructed from a sequence of combinators, a BC-chain having the maximum length is to be chosen for our purpose.

We give a bit more realistic example. The following is a definition of the factorial function f which receives an integer n and returns the value of the n-th factorial.

f n = cond (eq n 1) 1 (mult (f (minus n 1)) n).

The expression obtained by abstracting with the variable n is

f = S ([CBCB] (cond (eq I 1)) 1)

(S (([BBCB] mult) (f (minus I 1))) I).

The graph representation of this expression is shown in Fig. 3.

If a source expression contains more than one variable, the representation with BC-chains is slightly more complicated. In the original Turner's method, all the combinators for a variable which is first abstracted are distinguished from those for any other variable. In our present method, for the time being, each BC-chain will be assumed to be explicitly subscripted with an index in order to indicate the variable for which the combinators in the chain are responsible. This subscript will turn out to be unnecessary in the reduction algorithm in Section 5.

Fig. 4 shows the process of the translation (abstraction) of an expression

$$x_1 x_2 a \ ,$$

which is first abstracted with x_2 and then with x_1. The translation yields

$$[CC]_1[CB]_2(II)a \ ,$$

while the result of the ordinary translation is

$$C'C(C'BII)a.$$

From now on, we denote variables in a source expression by x_1, x_2, ..., x_k, and without loss of generality we fix the order of the variables for the abstraction as follows: first with x_k, then with x_{k-1}, ..., and finally with x_1.

As a larger example consider the following expression

$$x_3(x_1x_2x_2)x_1 \ ,$$

where x_1, x_2 and x_3 are variables. By abstracting this expression with x_3, x_2 and x_1 in this order, Turner's translation yields the following expression:

$$S'(C'C)(B'(B'C)I(C'S(C'BII)I))I \ .$$

This expression is also depicted in Fig. 5 (a). From this expression we can construct BC-chains in the same way as discussed above. See Fig. 5 (b) for the result of this BC-chaining. This is equivalently rewritten as

$$S_1([CB]_2[CC]_3)([BCC]_1I(S_2(B_2II)I))I \ ,$$

which is also depicted in Fig. 5 (c).

These examples suggest a naive method for translating a source program to a combinator graph with BC-chains. For the binary tree of a given functional program, the abstraction with a variable proceeds by attaching combinators to interior nodes, and then, as the second step, combinators B, B', C and C' are grouped to form a BC-chain. The detail of the translation (abstraction) method is left to the reader.

3. Space complexity of combinator graphs with BC-chains

Let n be the size of a source expression. In this section we prove that the order of the size of a combinator graph with BC-chains is linear in n.

First we show that each BC-chain can be represented in the space of the constant size, regardless of the length of a BC-chain. For this purpose, we encode a BC-chain in the following way. Label each node of the original expression with integers 1 through n in the inorder. (See Fig. 6 for the result of labelling the example in Fig. 5). Then each BC-chain can be encoded into a pair of integers [i,j] (with a subscript), where i is the label of the "starting node" of the chain and j is that of the "ending node". By applying this method to Fig. 5 (c), we obtain Fig. 7.

It is easy to see that from this representation with encoded BC-chains we can recover the original sequence of constituent combinators B and C. For example, from the encoded BC-chain [2,3] in Fig. 7, we can obtain that the head combinator of the BC-chain is B, since $2 < 3$, and that the tail part of the BC-chain is [6,3], since the right son node of the node (2) is (6), and so on.

The reader may wonder if pointers referring downward to the next combinator I or S can replace BC-chains in this context, but unfortunately they do not work when the reduction is taken into account.

Now we are ready to state the main result of this paper.

Theorem The size of the BC-chain representation is of linear order.

Proof: Fix a variable x in a source expression, and let r be the number

of occurrences of this variable in the expression. The increase of the size in the new representation with respect to this variable is at most

$r-1$ occurrences of the combinator S at interior nodes, and

$2r-1$ BC-chains,

so that in total at most $3r-2$.

Let $r_1, \ldots, r_k$ be the numbers of occurrences of all variables $x_1, \ldots, x_k$ in the source expression, respectively. Then obviously we have

$$r_1 + r_2 + \ldots + r_k \leq n.$$

Hence the total increase in the new representation is at most 3n.

4. A naive reduction algorithm for BC-chains

In this section we present a reduction algorithm for combinator expressions with BC-chains, which naively simulates the normal order reduction. The normal order (or leftmost-outermost order) reduction algorithm always reduces expressions (graphs) at the leftmost-outermost reducible part of the expression.

Our reduction rules for combinators are exactly the same as the original ones, except for BC-chains. In case of expressions with a single variable, the reduction rules for BC-chains are as follows.

$$[Bw]a(bc)d \longrightarrow a([w]bcd)$$

$$[Cw](ab)cd \longrightarrow [w]abdc$$

Here, w denotes the tail part of a BC-chain, and d is an actual argument in this reduction. The first rule is shown graphically in Fig. 8.

In case of expressions with more than one variable, the reduction rules for BC-chains for the i-th variable are as follows.

$$[Bw]_i ua(vbc)d \longrightarrow ua(v^*bcd)$$

$$[Cw]_i u(vab)cd \longrightarrow u(v^*abd)c$$

Here, u is of the form

$$p_{j_1}(p_{j_2}(\ldots(p_{j_{m-1}} p_{j_m})\ldots))$$

where each p_{j_h} is either a combinator or a BC-chain, $i < j_h \leq k$ (k is the number of variables), and the subscripts of these elements are sorted in the increasing order, i.e., $j_1 < j_2 < \ldots < j_m$. Also v has a similar form to u, and the only difference from u is that the range of the subscripts is between 1 and k. And finally v* is the result of inserting $[w]_i$ into v at the position where the order of subscripts is preserved. See Fig. 9 for a graphical representation of the reduction rule for $[Bw]_i$.

It is easily verified that these rules, together with the conventional reduction rules for combinators, constitute our desired reduction scheme which naively simulates the normal order reduction. However, the simulation is not always realtime, because of the insertion operation of the BC-chain $[w]_i$ into v.

For example, we illustrate in Fig. 10 how the reduction proceeds for the combinator graph in Fig. 5 (c). Subtrees a, b and c are actual arguments for variables x_1, x_2 and x_3, respectively. Each reduction step is always performed for a combinator or a BC-chain at the leftmost position of the graph. In the figure, small circles stand for newly created nodes during the reduction.

5. An efficient reduction algorithm for BC-chains

The simulation of the normal order reduction in the preceding section is conceptually simple but not always efficient in time as

mentioned above. In this section we present a more efficient reduction algorithm which runs in constant time per unit operation of the normal order reduction.

First we slightly change the structure of combinator graphs. In general, a combinator graph can be regarded as consisting of "basic building blocks" of the [standard] form in Fig. 11 (a), in which each of subtrees a and b consists of these basic blocks. We modify the form of the basic block to [modified] in Fig. 11 (b), where the symbol ϕ denotes the special "nil" node. This modified form will prove to be useful for achieving the desired efficiency of the reduction algorithm.

Another change to the previous algorithm is to eliminate all the subscripts of BC-chains and combinators. The reason for this is that we can immediately tell where to transfer the tails of BC-chains, as seen below.

Now we describe our second reduction algorithm. Initially, every basic block in a combinator graph to be reduced is transformed to the [modified] form, except the outermost basic block. This pretransformation can be done in time proportional to the number of nodes in the graph.

The reduction rule for a BC-chain [Bw] is graphically shown in Fig. 12 (which should be compared with Fig. 9), where w is nonnull. We explain the algorithm by using Fig. 12, since it is more comprehensible than formally describing the reduction. On reducing with a BC-chain [Bw] at the leftmost position of the graph having an argument c_i for the i-th variable, we tranfer the argument and the tail [w] of the BC-chain to the subtree t in Fig. 12. More precisely, we create two nodes p and q in this subtree. On reducing with [Bw], we transfer the argument c_i to the rightson of p and the tail [w] to the leftson

of q.

In case of the reduction with a single combinator B or S, instead of a BC-chain, we transfer an argument c_i to the top of the argument list. In this case there must be a combinator or a BC-chain for the i-th argument at the leftmost position of t. Therefore we simply move it to the top of the list of combinators or BC-chains in a similar way to the case of [w]. The reduction rules for [Cw], C and S are similarly defined in the subtree t'.

When all the actual arguments are transferred into t or t', the leftmost parts of t and t' must be empty (ø). Now the subtrees t and t' are transformed back to the [standard] form, except the inside of subtrees a and b. In this backward transformation we must rearrange the sequence of BC-chains (or single combinators) in the reverse order. (In practice, this transformation should be postponed until the reduction of a subtree starts.) Then the reduction proceeds to this new t' in the standard form.

The process of transferring and accumulating an actual argument and the tail of a BC-chain in a subtree can be performed in constant time, thanks to our restructuring of the subtree. Note that no operation of inserting the tail part of a BC-chain is required. The transformation of subtrees to the standard form can be done in time proportional to the number of variables in question, since the reversing of a list can be done in linear time with respect to the length of the list. Hence, in total, the simulation runs in constant time per unit operation of the normal order reduction.

6. Concluding remarks

In this paper we have shown that the BC-chain method achieves the linear space representation for combinator graphs with virtually no change to the original reduction scheme. This result has solved a complexity question which has been studied by several authors [2,3,5,6,7]. We believe that our method is worthwhile to investigate from the viewpoint of actual implementation as well.

In practical implementation it is expected that the lengths of BC-chains are reasonably small. In such cases the following technique seems promising both in space and time. A BC-chain is represented in a memory word as a bit string of B and C, e.g., '1' for B and '0' for C. When the length of a BC-chain is too large, we divide the BC-chain to several subchains in order to fit them in the word length, with little change to the reduction algorithm.

Besides our two algorithms, we can formulate still another reduction algorithm for BC-chains. The key idea of this algorithm is to reduce with all the constituent combinators of a BC-chain. Thus no transfer of the tail of a BC-chain is required. This is contrasted with our previous algorithms, which always reduce only with the head combinator of a BC-chain. This reduction differs from the normal order reduction in the sense of computation process, but it can be shown to be normalising, that is, every graph having a normal form is reduced to that form as the final result of this reduction. In practice this strategy seems to be more efficient.

References

[1] S. K. Abdali: An abstraction algorithm for combinatory logic, J. Symbolic Logic, 41 (1976), 222-224.

[2] F. W. Burton: A linear space representation of functional programs to Turner combinators, Inform. Process. Lett., 14 (1982), 201-204.

[3] T. Hikita: On the average size of Turner's translation to combinator programs, J. Inform. Process., 7 (1984), to appear.

[4] R. J. M. Hughes: Super-Combinators, Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming, 1982, 1-10.

[5] J. R. Kennaway: The complexity of a translation of λ-calculus to combinators, School of Computing Studies and Accountancy, Univ. of East Anglia, Norwich, 1982.

[6] J. R. Kennaway and M. R. Sleep: Efficiency of counting director strings, typescript, 1983.

[7] K. Noshita: Translation of Turner combinators in O(n log n) space, Inform. Process. Lett., to appear.

[8] D. A. Turner: Another algorithm for bracket abstraction, J. Symbolic Logic , 44 (1979), 267-270.

[9] D. A. Turner: A new implementation technique for applicative languages, Softw. Pract. Exper., 9 (1979), 31-49.

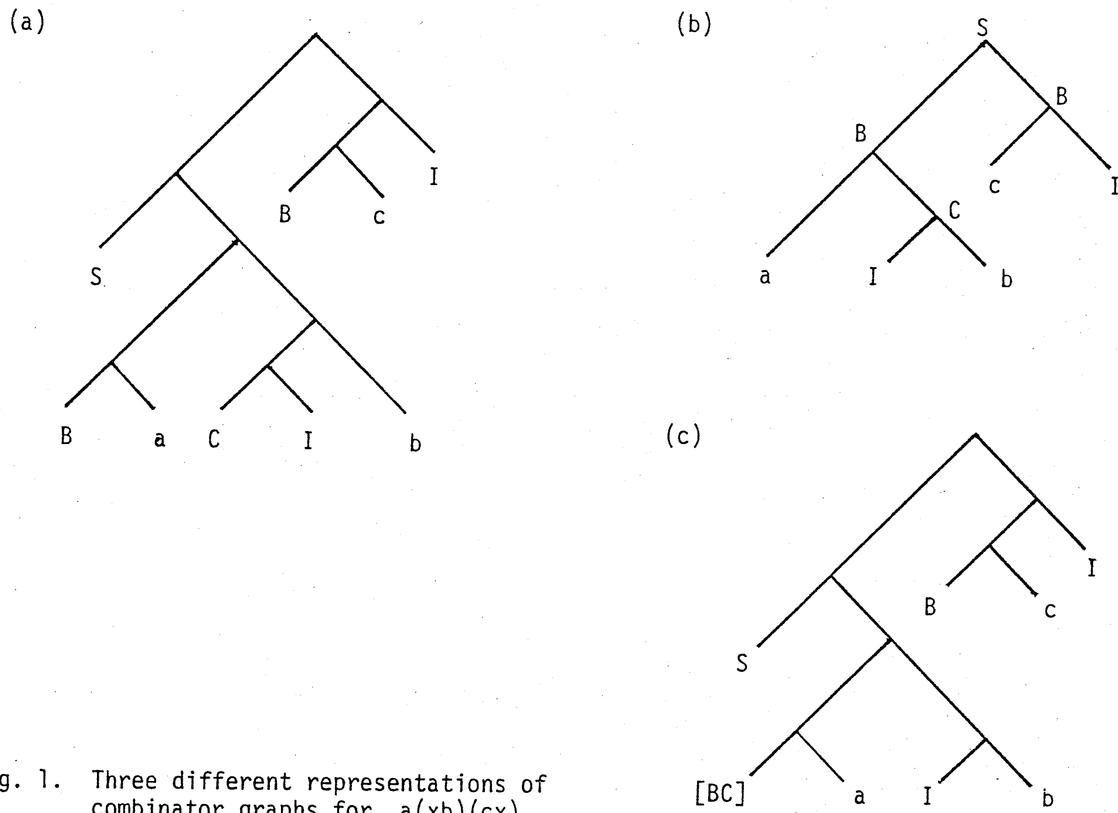

Fig. 1. Three different representations of combinator graphs for a(xb)(cx)

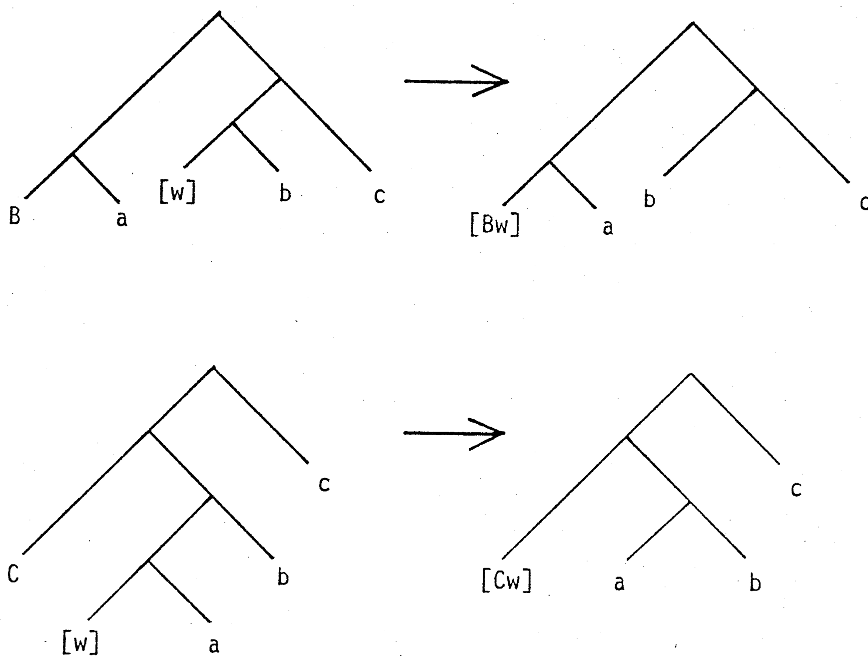

Fig. 2. Constructing BC-chains

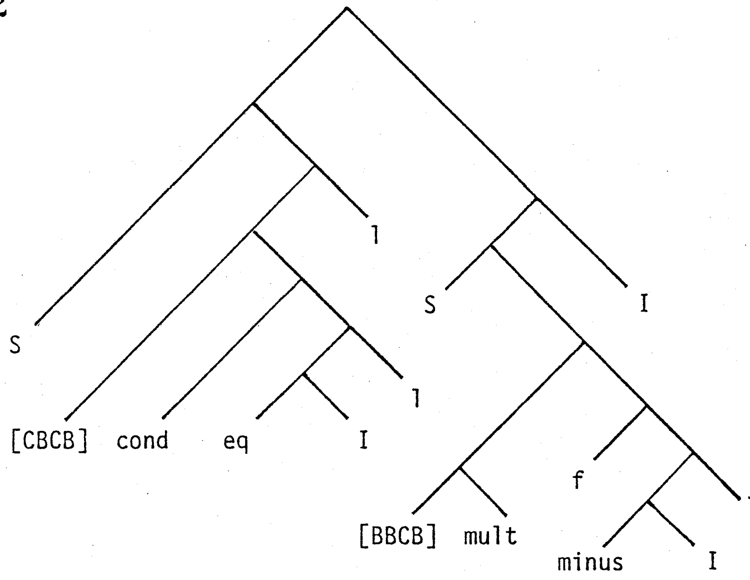

Fig. 3. Combinator graph with BC-chain for f

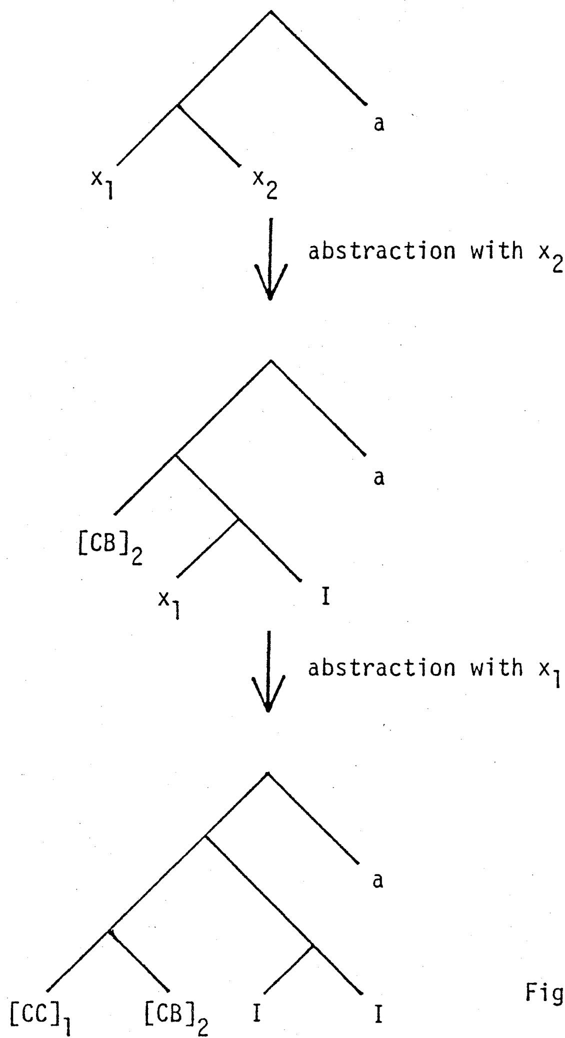

Fig. 4. BC-chains with more than one variable

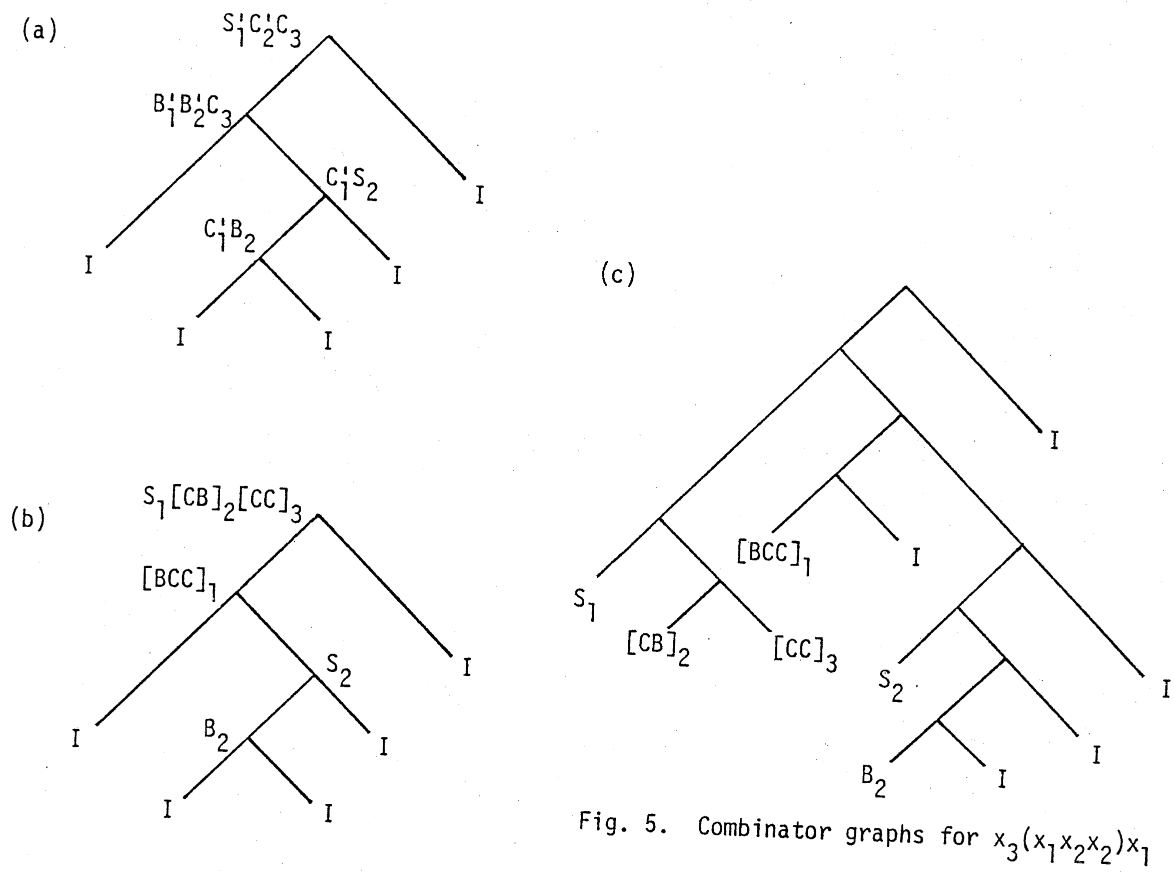

Fig. 5. Combinator graphs for $x_3(x_1x_2x_2)x_1$

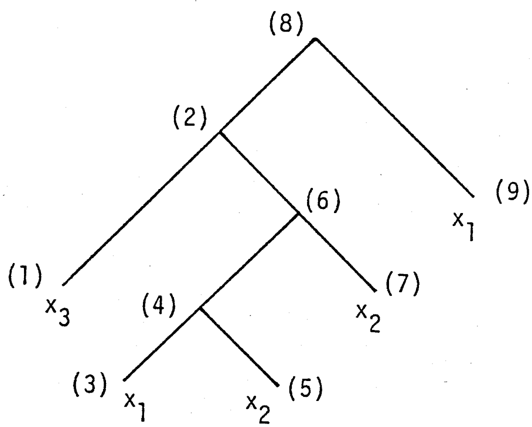

Fig. 6. Labelling the nodes in the inorder

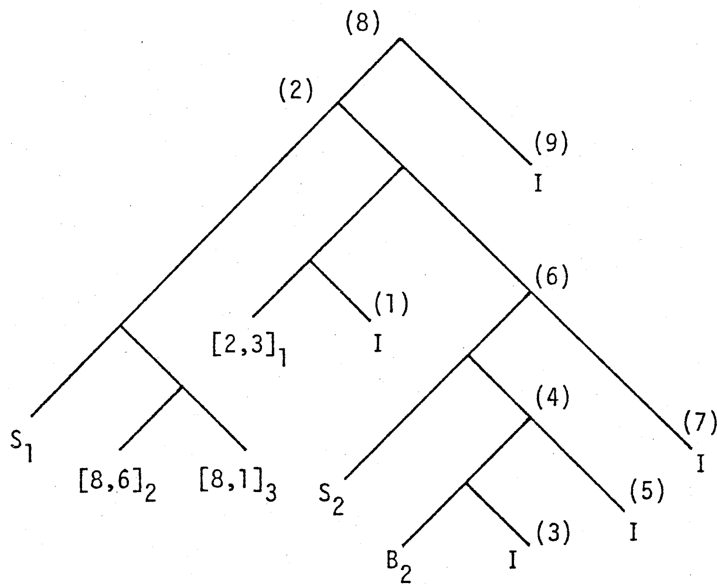

Fig. 7. Encoded BC-chains with pairs of Integers

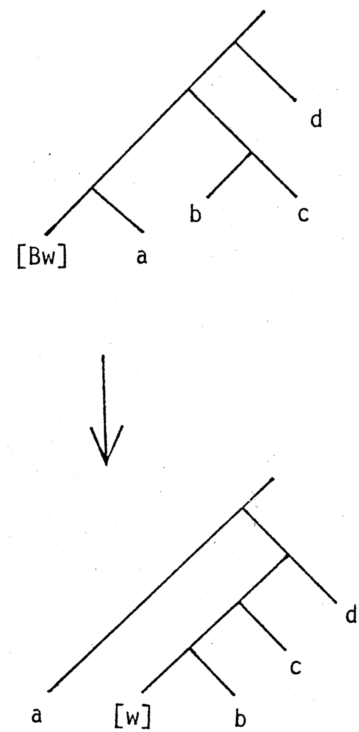

Fig. 8. Reduction rule for a BC-chain [Bw]

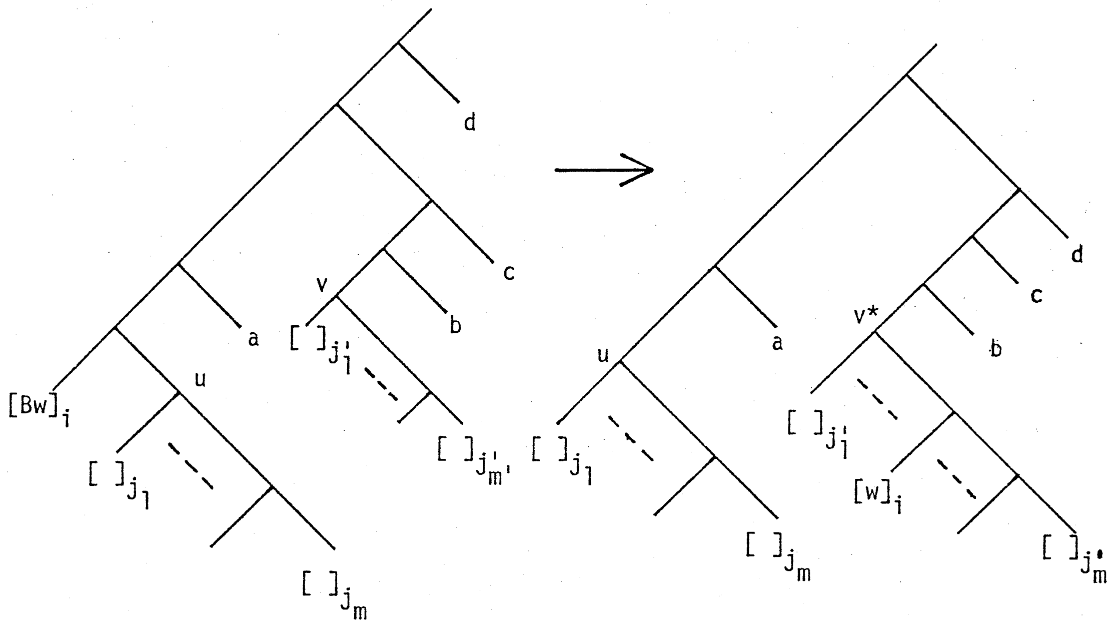

Fig. 9. Reduction rule for $[Bw]_i$ with more than one variable

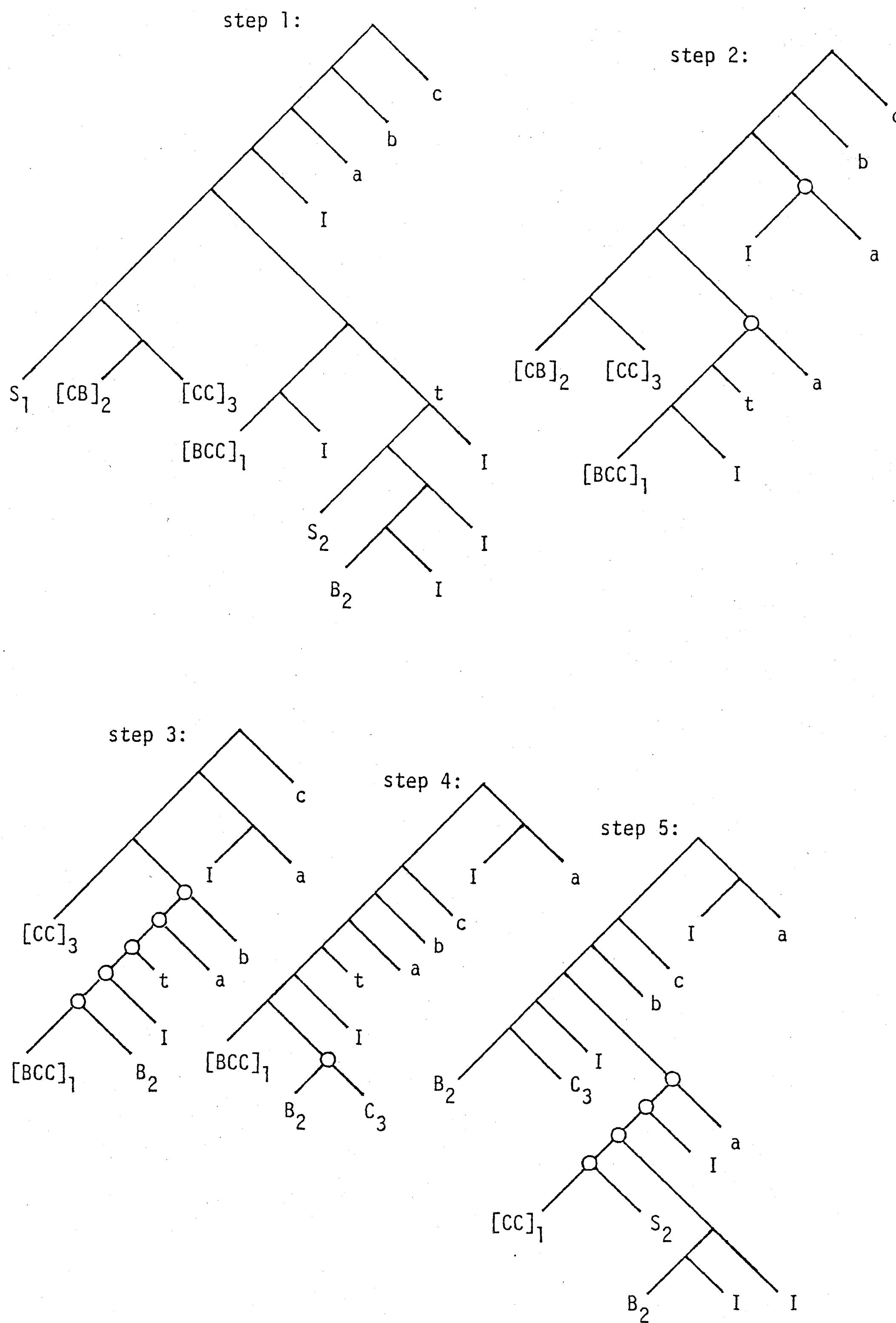

Fig. 10. Example of the reduction for BC-chains

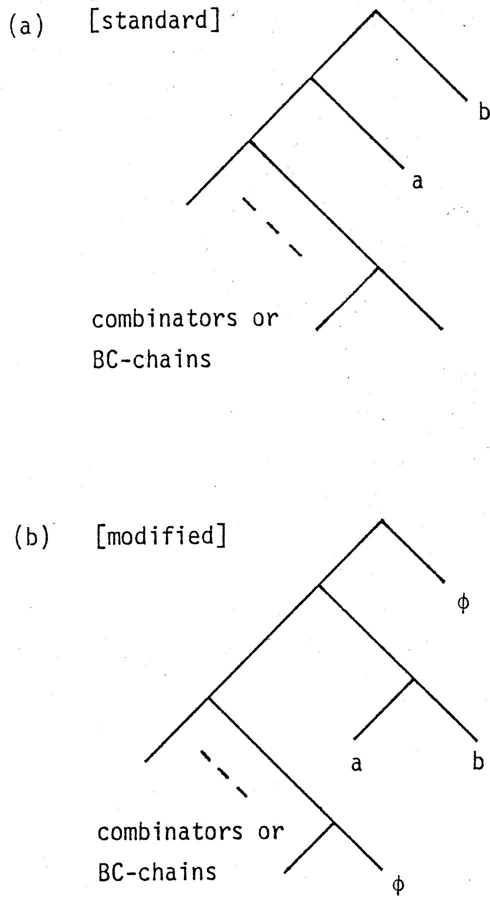

Fig. 11. Basic blocks of a combinator graph with BC-chains

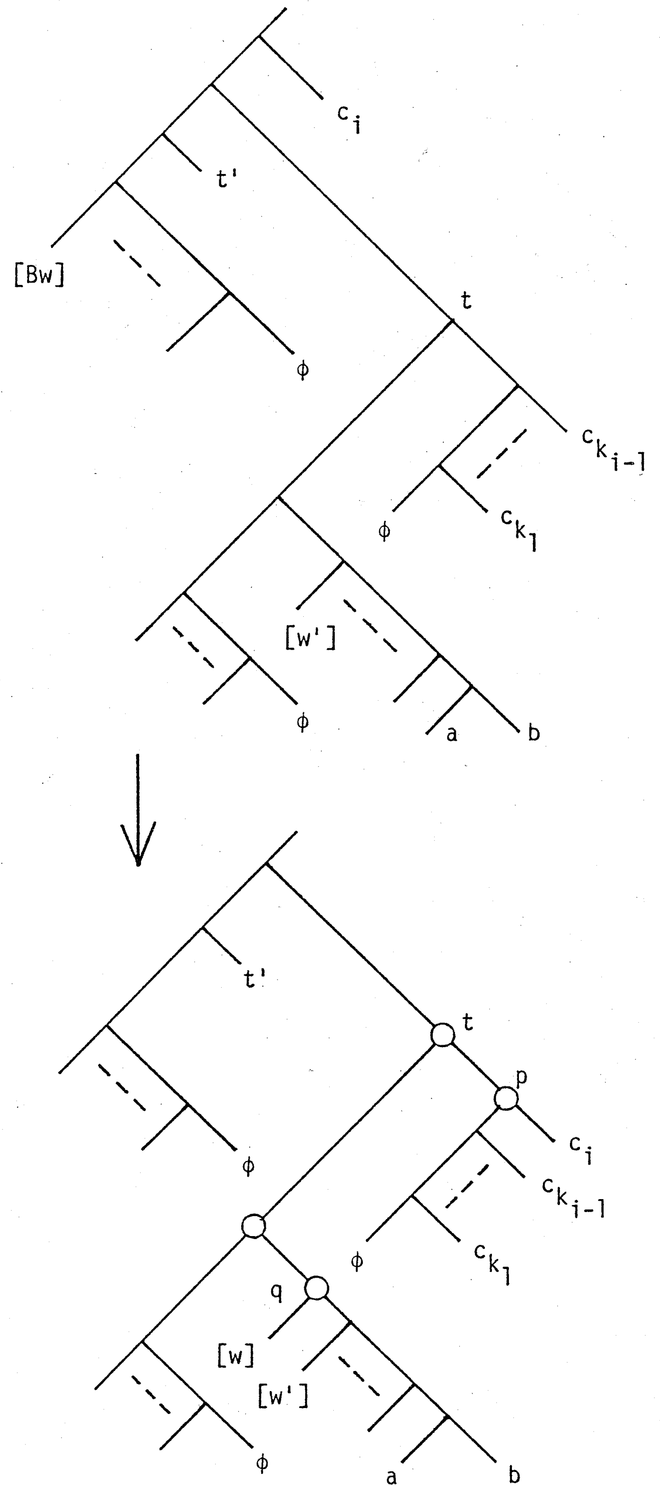

Fig. 12. Reduction rule for the modified form