Theoretical Computer Science 10 (1980) 171–185
© North-Holland Publishing Company

COMPUTATION ON GRAPH-LIKE EXPRESSIONS

John STAPLES
Department of Mathematics and Computer Science, Queensland Institute of Technology, Brisbane 4001, Australia

Communicated by Arto Salomaa
Received August 1978
Revised March 1979

Abstract. This paper begins a three-part study of efficient evaluations of expressions, which shows that for a wide class of expressions there are theoretical benefits in allowing graph-like expressions in evaluation algorithms. In this first part, a theory of graph-like expressions is developed as far as the basic theoretical result which underlies the remainder of the study. It asserts a modified commutativity property for computational steps in suitable systems of graph-like expressions.

1. Introduction

Graph-like expressions arise when one represents ordinary applicative expressions economically, by storing only one copy of (some) subexpressions which occur in several places in the expression. The practical advantages of such storage economy are clear, and well-known. In this paper and its sequels [11, 12] it is shown that sharing of subexpressions also has substantial theoretical advantages. In this paper a basic 'subcommutativity' theorem for graph-like expressions is established, which is known [9] to lead to a particularly simple optimality theory. That abstract theory is then applied in [11] to develop a theory of optimal evaluation of graph-like expressions, which gives in a uniform way optimal evaluation algorithms for a variety of systems of graph-like expressions. It is shown in [12] that under suitable conditions evaluation in systems of graph-like expressions is never less efficient than evaluations of corresponding classical applicative expressions.

There have been several approaches to the development of theories of computation on graph-like objects; see [2, 8] and the further references given there. Their motivations have been either the modelling of biological systems, or the development of a theory which generalizes various particular theories in computing science and theoretical biology.

The purpose of this paper is somewhat different. Our aim is to describe notions of graph-like expression, and computation on such expressions, which are specifically designed to facilitate the development of a simple theory of efficient evaluation of applicative expressions.

When are applicative expressions evaluated in practice? The classical example in computing science of an applicative expression evaluator is the LISP interpreter. It is now realized however that applicative expression languages have a much wider significance. For example, the most promising strategy for designing a highly parallel computer in which many processors can cooperatively execute a single program (when it is not assumed that the program instructions will have special forms determined by a special-purpose machine architecture) is to use a suitable applicative expression language as a machine-level or intermediate language; see for example [4] and the further references given there. Also there is a growing body of opinion that suitable applicative expression languages are natural high-level programming languages; see for example [1].

2. Definitions and first properties

2.1. Classical applicative expressions are finite symbols strings of the form

$$f(a_1, \ldots, a_n), \quad n \geq 0,$$

where $a_1, \ldots, a_n$ are again applicative expressions, and where f is a primitive symbol which we shall call a *function symbol* (see Fig. 1). In the case $n = 0$, f may also be called a *constant.*

The link with graphs is the observation that the parenthesis structure of such applicative expressions defines a finite, ordered tree with labelled nodes. That is, a constant c represents a single, terminal node labelled c; an expression $f(a_1, \ldots, a_n)$, $n \geq 1$, represents a tree whose root is labelled f and has n out-edges, say $e_1, \ldots, e_n$ respectively, ordered as indicated by the subscripts and such that e_i ends at the root of the tree represented by the expressions a_i, $i = 1, \ldots, n$.

Expressions with shared subexpressions can then be naturally thought of as collapsed such trees; that is, as directed, ordered, rooted graphs whose nodes are labelled by function symbols.

In practice, and in our theory, it is convenient to allow some terminal nodes of such graphs to be unlabelled; for consistency of notation, such nodes will be referred to as having the empty label. When taken together with a definition of homomorphism which allows empty-labelled nodes to have images with an arbitrary label, empty-labelled nodes will behave as parameters.

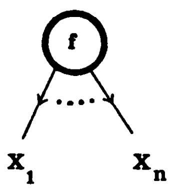

Fig. 1

Thus we define as follows *expression graphs* over a pair (F, C) of sets of primitive symbols; the elements of F are called function symbols, and the elements of the subset C of F are called constants.

Definition 1. An *expression graph* over (F, C) is a finite, rooted, directed, ordered graph, each of whose nonterminal nodes is labelled by an element of $F \setminus C$, and each of whose terminal nodes may be labelled by an element of C.

Note that we have not at this stage assigned 'arities' to elements of $F \setminus C$; that can be done in particular applications as required by restricting attention to a suitable subset of the expression graphs just defined. Our main results are for acyclic graphs only, but that restriction is not yet necessary.

2.2. It is convenient to have specific representations, as symbol strings, of the above expression graphs. For, such a representation allows a simpler description of the details of the transformations on expression graphs which are of interest. Also, it provides a link with the classical concept of a formal expression as a symbol string. Although it would be possible to formulate such a representation in terms of, for example, PASCAL 'record' data structures, the following simple symbol string representations of expression graphs are more suitable for our purposes.

We require for our representations a countable infinity of new formal symbols, which we shall call *addresses*. Intuitively these symbols model pointers; that is, names of locations. We will allow, for technical convenience (especially in the definitions of transformations on graph-like expressions) synonymous addresses; that is, we allow that different addresses may name the same location. We shall also require some formal punctuation symbols; parentheses (and), comma, and becomes :=.

Our representations will be constructed from *assignments*

$$s := v$$

where s is a finite, nonempty set of (synonymous) addresses and v is a *value*, defined as follows: the empty string is a value, called the empty value; each constant $c \in C$ is a value; and for all $f \in F \setminus C$, all $n > 0$ and all addresses $a_1, \ldots, a_n, f(a_1, \ldots, a_n)$ is a value. Note that we may also call v the value of the assignment, of s or of the elements of s; and s may be called the address set of the assignment.

Now an expression graph can be represented by a *term* $T = (r, P)$, where r is an address and P is a set of assignments which satisfies the following conditions:

(i) every address which occurs in the value of an assignment of P also occurs in the address set of some assignment of P,

(ii) address sets of distinct assignments of P are disjoint,

(iii) there is an assignment of P whose address set includes r,

(iv) every address in every assignment of P is the end of some path in P which starts at r; where a path in P is defined to be a sequence

$$a_1, m_1, a_2, m_2, \ldots, a_k, m_k, a_{k+1}, \quad k \geq 0,$$

where the a's are addresses of assignments of P, the m's are positive integers, and for all $i = 1, \ldots, k$ the value of a_i in T is

$$f(b_1, \ldots, b_m)$$

where $m \geq m_i$ and $a_{i+1} = b_{m_i}$. We may call a_1 the *start* of the path, a_{k+1} its *end* and k its *length*.

2.3. *Remarks*

(i) In the above notion of path the sequence is evidently determined by the smaller sequence $a_1, m_1, \ldots, m_k$; but the fuller notation given above is frequently convenient.

(ii) Address sets will often be singleton sets; we may loosely denote a singleton set by its unique element.

(iii) Two addresses of a term T which are in the same address set, and therefore are intuitively synonymous, may be called *equivalent*.

2.4. Terms represent expression graphs in the sense that a simple mapping of terms to expression graphs is defined as follows. To each assignment $s :=v$ of the given term T corresponds a node of the expression graph defined by T. If v is empty, the corresponding node of G is empty-labelled. If v has the form $f(a_1, \ldots, a_n)$, $n \geq 0$, then the corresponding node is labelled f and has n out-edges, say $e_1, \ldots, e_n$ with the ordering indicated by the subscripts, and where e_i ends at the node defined by the assignment $s' :=v'$ such that $a_i \in s'$.

It has already been implied that the addresses occurring in terms are devices for the concrete representation of expression graphs and do not have a place in the abstract, or geometrical, theory of expression graphs. They can be introduced to expression graphs as labels for the edges, but there is generally no advantage in doing so. On the contrary the theory of expression graphs is simplified by the omission of such arbitrary labels.

There are various ways in which an expression graph can be transformed to a term which represents it. Arbitrariness arises from arbitrary choices of addresses, and also because it must be arbitrarily decided whether two edges ending in the same nodes will be labelled by the same address or by different, equivalent addresses. It is not necessary to give here a general description of transformations of expression graphs to terms; specific transformations will be given as required.

Now that both expression graphs and terms are available, we shall use whichever formulation is more convenient for a particular purpose; but concepts defined for terms should be such as to induce a corresponding concept on expression graphs. That will generally be evident enough to require no explicit discussion.

Example. Consider the definition of path $a_1, m_1, \ldots, a_k, m_k, a_{k+1}$ in a term T. Such a path evidently induces a corresponding path

$$n_1, e_1, \ldots, n_k, e_k, n_{k+1}$$

in the corresponding expression graph; where n_i is the node defined by the assignment $s_i := v_i$ of T such that $a_i \in s_i$, and where e_i is the m_ith out-edge of m_i.

In this introductory paper we shall frequently prefer to give definitions and results for the more concrete notion of term; in the sequels [11, 12] it is often more convenient to give a more geometrical discussion in terms of expression graphs.

2.5. We can now consider computation on terms. When the definition of computational step, or *contraction*, on a term has been given it will be clear that it induces a corresponding notion on expression graphs.

Computation rules for applicative expressions allow subexpressions which match certain conditions to be replaced by other subexpressions. Similarly we shall take a rule ρ to be a suitable pair (R_ρ, C_ρ) of terms, with the intention that any copy of R_ρ which occurs in a term may be replaced by C_ρ. When implementing this intention, however, we have to take into account phenomena which have no counterparts in the evaluation of conventional applicative expressions. Consider for example the expression

$$\textbf{if } T \textbf{ then } E_1 \textbf{ else } E_2$$

which we shall write in the standard form

$$\textbf{ite}(T, E_1, E_2).$$

A conventional rule scheme for replacing occurrences of this expression as a subexpression in larger expressions would be of the form

$$\textbf{ite}(T, x_1, x_2) \Rightarrow x_1.$$

Such a rule scheme can be represented by a single rule (R_ρ, C_ρ) in our approach, where

$$R_\rho = \{r_\rho := \textbf{ite}(a_1, a_2, a_3), a_1 := T, a_2 := e, a_3 := e\}$$

and

$$C_\rho = \{a_2 := e\},$$

where we have confused the address sets with their unique elements, where r_ρ, a_2 are the roots of R_ρ, C_ρ respectively and where e denotes the empty value. The occurrence of a_2 in both R_ρ and C_ρ will have the effect that when the replacement of R_ρ by C_ρ is made the address a_2 will take on the same value as a_2 had in the original term.

Corresponding to the notion of rule for terms, a rule for expression graphs is a pair (R_ρ, C_ρ) of expression graphs (with R_ρ nontrivial) together with a function, A say, from (some) empty-labelled nodes of C_ρ to nodes of R_ρ. We shall place restrictions on the notion of term rule which will ensure that such a function A is induced by a term rule.

2.6. The following features of such a replacement have no counterpart in the evaluation of conventional applicative expressions:

(i) replacement of an occurrence of R_ρ in a term will not in general mean the removal of all assignments of R_ρ, since some of them may also form part of other subexpressions which should not be affected by this rule application,

(ii) in order to correctly simulate the application of the conventional rule to conventional expressions, the above rule (R_ρ, C_ρ) should be applicable to such terms as

$$I = \{r := \mathbf{ite}(a_1, a_2, a_3), \{a_1, a_2\} := T, a_3 := 0\}$$

and

$$I = \{r := \mathbf{ite}(a_1, a_1, a_3), a_1 := T, a_3 := 0\}$$

both of which represent the conventional expression

$$\mathbf{ite}(T, T, 0).$$

Thus we shall arrange that the rule (R_ρ, C_ρ) applies to homomorphic images of R_ρ.

2.7. At both of the points of Section 2.6 our approach to rules and their application differs from the biologically motivated approach of Rajlich [8]. The category-theoretic approach of [2, 3] generalizes, in a weak sense, both Rajlich's and the present definition; but at some cost in complexity and to an extent which is unnecessary for the work on efficient evaluation for which we are preparing. The generalization is a weak one because it does not preserve the rootedness of the graphs; it does not recognize 'garbage', as is essential for the study of optimality.

2.8. In order to give our precise definition of computational step we define a homomorphism $T \to U$ between terms to be a function h from addresses of T to addresses of U which satisfies the following conditions:

(i) h preserves equivalence; that is, for all $(s := v) \in T$ there is $(s' := v') \in U$ such that $h(s) \subseteq s'$,

(ii) h preserves values; that is, whenever $(s := f(a_1, \ldots, a_n)) \in T$, $(s' := v') \in U$ and $h(s) \subseteq s'$, v' has the form $f(b_1, \ldots, b_n)$, where $h(a_i)$ and b_i are equivalent in U, $i = 1, \ldots, n$.

Note that we have not required that the image of the root of T is the root of U, neither have we required that addresses which are empty-valued in T should have images in U which are empty-valued.

Clearly this definition induces a natural notion of homomorphism on expression graphs.

2.9. We also define an address a of a term C_ρ in a pair (R_ρ, C_ρ) to be *committed* if it is also an address of R_ρ. Now the conditions which we place on a pair (R_ρ, C_ρ) in order that it should be a rule are as follows:

(i) all committed addresses of C_ρ are empty-valued addresses of C_ρ,

(ii) all addresses of C_ρ which are equivalent to a committed address are committed,

(iii) committed addresses which are equivalent in C_ρ are equivalent in R_ρ,

(iv) the root of R_ρ is not empty-valued.

These conditions will ensure that our definition of application of a rule succeeds in correctly simulating rule application for conventional expressions. Conditions (i) to (iii) ensure that the notion of 'committed' carries over to the nodes of the expression graph defined by C_ρ, and that there is a unique node of the expression graph defined by R_ρ which corresponds to a given committed node of C_ρ.

2.10. We call a rule (R_ρ, C_ρ) *acyclic* if R_ρ and C_ρ are acyclic terms.

2.11. An *instance* of a rule (R_ρ, C_ρ) is a homomorphism $h: R_\rho \to T$. An application of a rule (R_ρ, C_ρ) to a term T, which for brevity we call a *contraction*, is a transformation of T to a term U which is defined as below by an instance $h: R_\rho \to T$.

(i) First take a suitable copy $i(C_\rho)$ of C_ρ; that means, for each committed address a of C_ρ, $i(C_\rho) = h(a)$ and for each uncommitted address a of C_ρ, $i(a)$ is distinct from all addresses of T. It is convenient in the rest of this definition to call an assignment of $i(C_\rho)$ uncommitted if the addresses of the corresponding assignment of C_ρ are uncommitted.

(ii) Write T' for the union of T with the set of uncommitted assignments of C_ρ.

(iii) Writing $s := v$, $s' := v'$ for the assignments of T' such that the images in T' of the roots of R_ρ, C_ρ are in s, s' respectively, write T'' for the set of assignments obtained from T' by replacing both these assignments by

$$s \cup s' := v'.$$

(iv) The root of T will become the root of the term U which results from the computational step. Create U by deleting from T'' all addresses which do not end any path in T'' which starts at the root of T (and delete assignments which then have empty address sets).

2.12. The above definition involves a rather arbitrary choice of an isomorphism i. Since this arbitrariness arises only as a result of a specific choice of addresses, it does not carry over to the theory of expression graphs.

2.13. Systems of applicative expressions which can be modelled straightforwardly by graph-like expressions include:

(i) classical weak combinatory logic (for which see e.g. [5]),

(ii) all of McCarthy's recursive definitions [6],

(iii) The programming languages of Vuillemin [13] and Pacini et al. [7].

The lambda calculus can also be modelled, though before the optimality theory of [11] can be applied it is necessary to make a suitable reformulation of the rules of the lambda calculus; see [10].

2.14. As an example we give here a graph-like approach to the simplest of all nontrivial systems of applicative expressions: weak combinatory logic. The correctness of this approach to weak combinatory logic follows from the work of [12].

In the conventional system there are two constants S and K, which are expressions, and for all expressions T_1 and T_2, (T_1T_2) is an expression. A more standard notation for (T_1T_2) would be $\alpha(T_1, T_2)$, where α is another function letter; the conventional notation is merely simpler.

The terms of the corresponding graph-like system are therefore constructed from three function symbols S, K and α, with the following restrictions:

(i) all terms are acyclic,

(ii) all addresses of terms have nonempty values,

(iii) for all values $f(a_1, \ldots, a_n)$ of a term of the system, either f is α and $n = 2$, or $f = S$ and $n = 0$, or $f = K$ and $n = 0$.

The conventional system has two rule schemes,

$$(((SX)Y)Z) \Rightarrow ((XZ)(YZ)) \quad \text{and} \quad ((KX)Y) \Rightarrow X,$$

where X, Y and Z denote arbitrary terms.

The counterparts of these schemes in the graph-like system are two rules which are denoted (R_S, C_S) and (R_K, C_K), are illustrated in Fig. 2 and are defined as follows. Note the terms appearing in the rules are not themselves terms of the system, since they include empty-valued addresses. Note also that in Fig. 2 the edges of the graphs have been labelled by addresses, so as to aid comparison with their corresponding terms; and unlabelled nodes have been omitted. In each case a spurious edge, having no start but ending at the root-node of the graph, has been introduced so as to allow the root address to be displayed;

$$R_K := \{r_K := \alpha(a_1, a_2), a_1 := \alpha(a_3, a_4), a_3 := K, a_2 := e, a_4 := e\},$$

$$C_K := \{a_4 := e\},$$

$$R_S := \{r_S := \alpha(a_1, a_2), a_1 := \alpha(a_3, a_4), a_3 := \alpha(a_5, a_6), a_5 := S,$$
$$a_2 := e, a_4 := e, a_6 := e\}$$

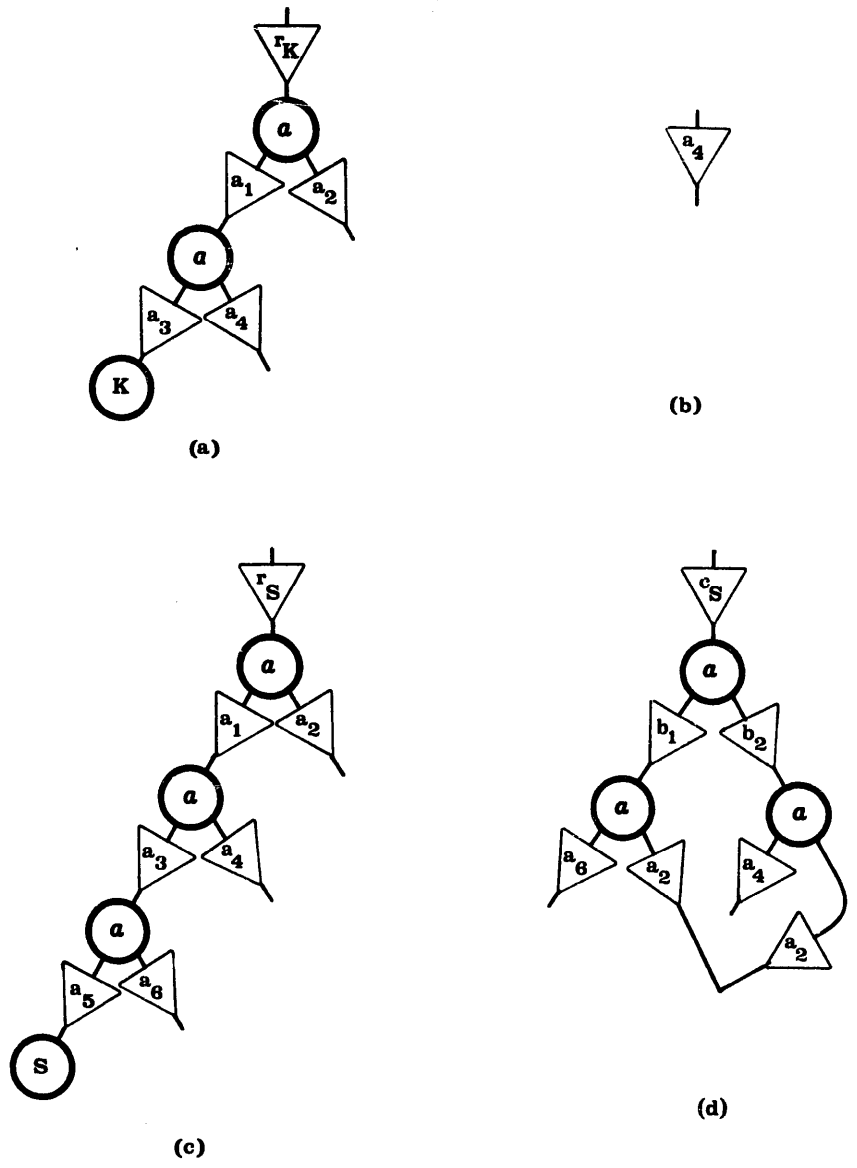

Fig. 2

and

$$C_S := \{c_S := \alpha(b_1, b_2), b_1 := \alpha(a_6, a_2), b_2 := \alpha(a_4, a_2), a_2 := e,$$
$$a_6 := e, a_4 := e\}.$$

3. The main result

3.1. The result stated in this section is fundamental to the study of graph-like expressions in general, and in particular to the optimality theory of [11] and [12] for which we are preparing. The result is, to begin with, a consistency theorem (a Church-Rosser theorem) which ensures for many systems of graph-like expressions that any two complete evaluations of the same expression give the same result. However it also has a particularly simple structure, not possible for nontrivial systems of conventional expressions, which is the key to the optimality theory.

3.2.

Definition 2. First we define two instances $h: R_\rho \to T$ and $k: R_\sigma \to T$ of rules ρ and σ to be *disjoint* if (writing r_ρ and r_σ for the roots of R_ρ and R_σ respectively);

(i) $h(r_\rho)$ is not equivalent to any address $k(p)$ such that p has nonempty value in R_σ,

(ii) symmetrically, $k(r_\sigma)$ is not equivalent to any address $h(p)$ such that p has nonempty value in R_ρ.

Note that ρ and σ are not required to be distinct rules.

3.3. The main result can be stated as follows (see Fig. 3).

$$\begin{tikzcd} T \arrow[r] \arrow[d] & U \arrow[dl, dashed] & \text{or} & T \arrow[r] \arrow[d] & U & \text{or} & T \arrow[r] \arrow[d] & U \arrow[d, dashed] \\ V & & & V \arrow[ur, dashed] & & & V \arrow[r, dashed] & W \end{tikzcd}$$

Fig. 3

Theorem 1. *If $h: R_\rho \to T$ and $k: R_\sigma \to T$ are disjoint instances of acyclic rules applied to an acyclic term T which define contractions $T \Rightarrow U$, $T \Rightarrow V$ respectively, then there are induced instances $h: R_\rho \to V$, or $k: R_\sigma \to U$, or both, such that*

(i) *h defines a contraction $V \Rightarrow U$, or*

(ii) *k defines a contraction $U \Rightarrow V$, or*

(iii) *there is W such that h and k define contractions $V \Rightarrow W$ and $U \Rightarrow W$ respectively.*

3.4. Remarks

(i) More precisely, the above result is true of expression graphs. A precise statement of the result for terms should take into account the isomorphisms mentioned in Section 2.12. For brevity, however, we are continuing to suppress discussion of those isomorphisms.

(ii) More precisely, the above result is true of isomorphism classes of terms; as previously mentioned, we usually ignore the difference between terms and their isomorphism classes.

(iii) The reason why we do not always get a simple commutative diagram, the last of the three alternatives, is that one of the contractions of T may discard that part of the term in which the site of the other contraction occurs.

(iv) The author has called systems, all of whose contractions have the property asserted in Section 3.3, *subcommutative* [9]. It is a characteristic feature of many systems of graph-like expressions and is shown in [9] to lead to a simple optimality theory.

(v) We said above that the category-theoretic approach to graph-like expressions provides 'in a weak sense' a generalization of our work. That is because in the category-theoretic approach the natural definition of computational step, or contraction, does not include part (iv) of Section 2.11, that is the 'garbage collection' phase in which parts of the expression which have become superfluous are discarded. While the omission of that phase is convenient, in as much as it simplifies our result (Theorem 1) to a commutativity result (the third of our three alternatives), it is essential for our optimality theory that the garbage collection phase be included; otherwise, all complete evaluations have the same length. That does not mean that our results depend on garbage collection taking place in practice after each computational step; our approach simply incorporates a convention that parts of an expression which are eligible for garbage collection should not be taken into account.

4. Proof of the main result

4.1. It is convenient to approach the proof of Theorem 1 through a commutativity lemma. As it is essentially a special case of the commutativity results of [2, 3], we shall only sketch the simple argument which suffices in this special case. For the discussion of the commutativity lemma we generalize the concept of term slightly, by omitting part (iii) of the definition of term given in Section 2, 2; we call the resulting objects *preterms*.

Thus a *preterm* P is a nonempty set of assignments such that

(i) every address in the value of every assignment of P occurs in the address set of some assignment of P,

(ii) address sets of distinct assignments of P are disjoint.

Note that there is no distinguished root in a preterm.

4.2. The definition of term homomorphism extends immediately to preterms.

4.3. We also generalize slightly the notion of contraction of a term, to a notion for preterms which we call *replacement*, by omitting part (iv) of the definition of contraction given in Section 2.11.

4.4. As in the case of contractions, we shall generally, for the sake of brevity, ignore the distinction between preterms and their isomorphism classes. Where it is necessary to refer to the copy $i(C_\rho)$ of C_ρ mentioned in Section 2.11, we shall argue as if $i(C_\rho) = C_\rho$.

4.5. Thus a replacement defined by a homomorphism $h: R_\rho \to P$ of the left side of a rule (R_ρ, C_ρ) into a preterm P will be loosely regarded as the successive application of two functions h_1 and h_2 as follows:

(i) $h_1(P)$ denotes the preterm obtained from P by the addition to P of the uncommitted assignments of C_ρ,

(ii) $h_2(P)$, where P includes assignments $s := v, s' := v'$ such that $h(r_\rho) \in s$ and $c_\rho \in s'$, denotes the preterm obtained from P by removing those two assignments and replacing them by $s \cup s' := v'$.

4.6. We shall also write $r(P)$, where r is some address of some assignment of the preterm P, for the term obtained from P by deletion of all addresses which are not at the end of some path in P which starts at r.

Thus a contraction of a term T by an instance $h: R_\rho \to T$ can be denoted:

$$(r, T) \Rightarrow (r, r(h_2(h_1(T)))).$$

It will be convenient to use the ordinary notation for composition of functions; for example,

$$h_2(h_1(P)) \text{ may be denoted } (h_2 \circ h_1)P.$$

4.7. In order to prove the main result we thus consider two contractions which we denote

$$(r, T) \Rightarrow (r, r((h_2 \circ h_1)T)) = (r, U) \quad \text{say,}$$

and

$$(r, T) \Rightarrow (r, r((k_2 \circ k_1)T)) = (r, V) \quad \text{say,}$$

where $h: R_\rho \to T$ and $k: R_\sigma \to T$ are instances of rules (R_ρ, C_ρ), (R_σ, C_σ) respectively. Our first step is to show that replacements commute.

Lemma 1. *If h, k are disjoint instances of acyclic rules, and if P is an acyclic preterm, then*

$$(h_2 \circ h_1 \circ k_2 \circ k_1)P = (k_2 \circ k_1 \circ h_2 \circ h_1)P.$$

Although this lemma is essentially a special case of the category-theoretic commutativity result of [2], the following more elementary proof takes advantage of the simplicity of our definitions.

Proof. Clearly $(h_1 \circ k_1)P = (k_1 \circ h_1)P$, since set union is associative ' ᵓd commutative.

Since k_1 simply adds some assignments, none of whose address sets includes $h(r_\rho)$ or c_ρ, it is also clear that

$$(k_1 \circ h_2 \circ h_1)P = (h_2 \circ k_1 \circ h_1)P. \tag{1}$$

Symmetrically,

$$(h_1 \circ k_2 \circ k_1)P = (k_2 \circ h_1 \circ k_1)P. \tag{2}$$

Next we observe that in order to prove Lemma 1 it is enough to show

$$(h_2 \circ k_2 \circ h_1 \circ k_1)P = (k_2 \circ h_2 \circ h_1 \circ k_1)P. \tag{3}$$

For then,

$$\begin{aligned}(h_2 \circ h_1 \circ k_2 \circ k_1)P &= (h_2 \circ k_2 \circ h_1 \circ k_1)P \quad \text{from (2)},\\ &= (k_2 \circ h_2 \circ h_1 \circ k_1)P \quad \text{from (3)}\\ &= (k_2 \circ h_2 \circ k_1 \circ h_1)P\\ &= (k_2 \circ k_1 \circ h_2 \circ h_1)P \quad \text{from (1)}.\end{aligned}$$

Hence we now prove (3). The situation is simple enough to allow the following combinatorial proof:

We write $s := v$, $s' := v'$, $t := w$, $t' := w'$ for the assignments of $(h_1 \circ k_1)P$ such that $h(r_\rho) \in s$, $c_\rho \in s'$, $k(r_\sigma) \in t$, $c_\sigma \in t'$. There are several cases to consider, depending on which of s, s', t, t' are distinct, but their number is reduced by the fact that $s \neq t$, since h, k are disjoint. In particular,

(i) since $s \neq t$, trivially s, s', t, t' cannot all be equal,

(ii) similarly the only two ways in which just three of s, s', t, t' can be equal are:

$$s = s' = t' \neq t \quad \text{and} \quad t = t' = s' \neq s,$$

but by symmetry only one of these cases, say the former, need be considered,

(iii) similarly, since $s \neq t$ there are a priori only seven ways in which any two of s, s', t, t' can be equal, of which two can be omitted by symmetry, leaving the following five cases:

$s = t' \neq t = s'$,

$s = s' \neq t' = t$,

$s = s'$, t, t' are distinct,

$s = t'$, s', t are distinct,

$s' = t'$, s, t are distinct;

The first of these cases does not occur since P and the rules are acyclic, so that four cases remain to be considered,

(iv) finally it is possible that all four of s, s', t, t' are distinct.

In each of the above six cases the argument is elementary.

4.8. Our next step in the proof of Theorem 4 is a lemma which describes the properties of the operation $P \Rightarrow r(P)$ which converts a preterm P with address r to a term with root r.

Lemma 2. *If P is an acyclic preterm, if r is an address of P and if $h: R_\rho \to P$ is an instance of an acyclic rule, then*

(i) *if $h(r_\rho)$ is not equivalent to an address of $r(P)$, then*

$$r(P) = (r \circ h_2 \circ h_1)P,$$

(ii) *if $h(r_\rho)$ is equivalent to some address of $r(P)$, then h defines an instance $R_\rho \to r(P)$, also loosely denoted h, and*

$$(r \circ h_2 \circ h_1)P = (r \circ h_2 \circ h_1 \circ r)P.$$

Proof. (i) We show that no address in $(h_2 \circ h_1)P$ which is an address of C_ρ or is equivalent to $h(r_\rho)$ can be the end of a path π in $(h_2 \circ h_1)P$ which starts at r. For if there were such a path, and if π is one of minimal length, then π is not a path in P. Thus write π' for the maximal initial subpath of π which is a path in P. Then π' is a proper subpath of π, but on the other hand the end of π' must be equivalent to $h(r_\rho)$, since only those addresses of P have their values altered in the transformation $P \Rightarrow (h_2 \circ h_1)P$. That contradicts the minimality of π.

(ii) Since R_ρ and C_ρ are terms and $h(r_\rho)$ is equivalent to an address, p say, of $r(P)$, then for all addresses $q \neq r$ of R_ρ, $h(q)$ is in $r(P)$, so h induces an instance $h': R \to r(P)$ as follows.

$$h'(r) = p, \qquad h'(q) = h(q) \quad \text{for } q \neq r.$$

We loosely denote h' by h also.

It is thus enough to show that every path π of $(h_2 \circ h_1)P$ which starts at r is a path of $(h_2 \circ h_1 \circ r)P$. Argument by induction on the length of π is straightforward and is omitted.

4.9. It is now easy to prove Theorem 1 in the following form, where we continue to use the notation introduced above:

One of the following three cases holds:

(i) *$h(r_\rho)$ is not equivalent to an address of $k(T)$: in which case*

$$(r \circ k_2 \circ k_1 \circ r \circ h_2 \circ h_1)T = (r \circ k_2 \circ k_1)T,$$

(ii) *$k(r_\sigma)$ is not equivalent to an address of $h(T)$: in which case*

$$(r \circ h_2 \circ h_1 \circ r \circ k_2 \circ k_1)T = (r \circ h_2 \circ h_1)T$$

(iii) *$h(r_\rho)$, $k(r_\sigma)$ are equivalent to addresses of $k(T)$, $h(T)$ respectively: in which case*

$$(r \circ h_2 \circ h_1 \circ r \circ k_2 \circ k_1)T = (r \circ k_2 \circ k_1 \circ r \circ h_2 \circ h_1)T.$$

Proofs. (i)

$$\begin{aligned}(r \circ k_2 \circ k_1)T &= (r \circ r \circ k_2 \circ k_1)T \\ &= (r \circ h_2 \circ h_1 \circ r \circ k_2 \circ k_1)T \quad \text{from Lemma 2 (i),}\end{aligned}$$

(ii) is symmetrical,

(iii)

$$\begin{aligned}(r \circ h_2 \circ h_1 \circ r \circ k_2 \circ k_1)T &= (r \circ h_2 \circ h_1 \circ k_2 \circ k_2 \circ k_1)T \quad \text{from Lemma 9 (ii)} \\ &= (r \circ k_2 \circ k_1 \circ h_2 \circ h_1)T \quad \text{from Lemma 1} \\ &= (r \circ k_2 \circ k_1 \circ r \circ h_2 \circ h_1)T \quad \text{from Lemma 9 (ii).}\end{aligned}$$

References

[1] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, 1977 Turing lecture, *Comm. A.C.M.* **21** (1978) 613–641.

[2] H. Ehrig, H.-J. Kreowski, A. Maggiolo-Schettini, B.K. Rosen and J. Winkowski, Deriving structures from structures, I.B.M. Research Report RC 7046 No. 30203 (1978).

[3] H. Ehrig and B.K. Rosen, Commutativity of independent transformations on complex objects, I.B.M. Research Report RC6251 No. 26882 (1976).

[4] D.P. Friedman and D.S. Wise, Aspects of applicative programming for parallel processing, *I.E.E.E. Trans. Computers* **27** (1978) 289–296.

[5] J.R. Hindley, B. Lercher and J.P. Seldin, *Introduction to Combinatory Logic* (Cambridge University Press, London, 1972).

[6] J. McCarthy, Basis for a mathematical theory of computation, in: P. Braffort and D. Hirschberg, Eds, *Computer Programming and Formal Systems* (North-Holland, Amsterdam, 1963) 33–70.

[7] G. Pacini, C. Montangero and F. Turini, Graph representation and computation rules for typeless recursive languages, in: J. Loeckx, Ed., *Automata, Languages and Programming*, Lecture Notes in Computer Science **14** (Springer, Berlin, 1974) 157–169.

[8] V. Rajlich, Dynamics of discrete systems and pattern reproduction, *J. Comput. System Sci.* **11** (1975) 186–202.

[9] J. Staples, A class of replacement systems with simple optimality theory, *Bull. Austral. Math. Soc.* **17** (1977) 335–350.

[10] J. Staples, A graph-like lambda calculus for which leftmost-outermost evaluation is optimal, in: *Proc. 1978 Graph Grammars Workshop*, Lecture Notes in Computer Science **73** (Springer, Berlin, 1979).

[11] J. Staples, Optimal evaluations of graph-like expressions, *Theoret. Comput. Sci.* **10** (1980) to appear.

[12] J. Staples, Speeding up subtree replacement systems, *Theoret. Comput. Sci.* **11** (1980) to appear.

[13] J. Vuillemin, Correct and optimal implementations of recursion in a simple programming language, in: *Proc. 5th Annual A.C.M. Symposium on the Theory of Computing*, Austin, TX (1973).