# A KERNEL LANGUAGE FOR PROGRAMMED REWRITING OF (HYPER)GRAPHS [1]

Andrea Maggiolo-Schettini
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56 125 Pisa, Italy

Józef Winkowski
Instytut Podstaw Informatyki PAN
ul. Ordona 21, 01 237 Warszawa, Poland

**Abstract:** The paper presents a formalism for rewriting (hyper)graphs in a controlled manner. This formalism is essentially a simple programming language with productions, that is rewriting rules, playing the role of basic instructions. Programs in this language are built from productions by means of rather standard constructors, including a parallel composition. They may contain parameters to point to specific elements of graphs to which they are supposed to be applied. Programs are intended to describe how to transform a graph and a valuation of parameters in this graph in order to reach a resulting graph and a resulting valuation of parameters.

**Key words:**
(hyper)graph, rewriting rule, production, parametrized production, rewriting step, program of rewriting, structural operational semantics, denotational semantics, resulting relation.

## 1 Introduction

Graphs are powerful and flexible representations of complex object structures. Evolutions of such structures can be represented conveniently by graph transformations. When these transformations are local then often they can be performed by rewriting graphs according to formal rules called productions, where a rule says that a certain given pattern can be replaced by another pattern, if it occurs in a graph.

Also some models of computing can be formulated in a natural manner in terms of rewriting of appropriate data structures represented as graphs. Take, for instance, the representation of actor systems as in [JaRo 91] or that of logic programs in [CMREL 91].

A theory of graph rewriting systems has been developed which describes how to

[1] This work has been supported by the Italian National Council for Research (CNR-GNIM), by the Polish Academy of Sciences (IPI PAN), and by COMPUGRAPH Basic Research Esprit Working Group n. 7183. It has been published in the journal Acta Informatica, vol 33 (1996), pp.523-546.

rewrite graphs of very general types, including hypergraphs, coloured hypergraphs, relational structures, etc. (cf. [EPS 73], [CER 79], [EKMRW 82], [ENR 83], [ENRR 87], [EKR 91]). This theory in its pure form does not assume anything about where and in what order to apply productions. In this situation at each stage of rewriting an independent search of an applicable rule and of a place of application must be done, which in general is a task of high complexity. On the other hand, in some problems the structure of data and the algorithm to solve the problem allow to organize rewriting in an efficient manner.

Observations of this type inspired various attempts of enriching graph rewriting systems by equipping them with control mechanisms similar to those of programming languages. Such mechanisms can be found, for example, in [B 79] and [Na 79]. Recently some works in this area resulted in specialized programming and specification languages (cf. [S 91] and [ZS 92], for example).

The present paper is another attempt of creating a formalism for organizing processes of rewriting graphs. Our formalism is in the framework of the algebraic approach proposed in [CER 79], [ENRR 87], and [EKR 91]. It is essentially a kernel of a simple programming language with productions playing the role of basic instructions. Programs are built in this formalism from productions by means of rather standard constructors which define the order and modalities of rewriting steps. Among the program constructors which have not been considered in the context of programmed graph rewriting there is a parallel composition which declares the possibility of executing programs in parallel. The parallelism is understood here as an arbitrary interleaving of atomic (i.e. indivisible) actions of component programs, where atomic actions are either single instructions or larger parts of programs, if they are specified as atomic with the aid of a special constructor. Another important feature of our approach is that productions and programs may contain parameters to point to particular elements of graphs to which they are supposed to be applied. When applied to pairs consisting of a graph and a valuation of parameters in this graph they transform such pairs one into another as long as it follows from their meaning. The mechanism of accessing graphs through valuations of parameters allows to enforce components of a program to operate on the same data and to realize shared variables whose values represent some parts of data.

For simplicity we assume only a global environment and we do not consider problems of typing.

The presented formalism is endowed with a structural operational semantics in the style of [Plo 81] and with a denotational semantics which is consistent with the operational one. These semantics define the possible executions of programs. Consequently, they determine the corresponding relations between data and results of performed executions.

The formalism we define may be useful whenever a problem can naturally be reduced to graph rewriting and the process of rewriting is too complex to be represented as a result of a free application of a system of productions. We shall illustrate it on example of a concurrent execution of a program in a simple concurrent logic language (called FCP after [Sh 89]).

**1.1. Example** (after [Sh 89]). Consider the logic program:

$$
\begin{array}{l}
sum(Y,S) \leftarrow sum'(Y,0,S) \\
sum'([],P,S) \leftarrow P = S \\
sum'([X|Y],P,S) \leftarrow plus(X,P,Q), sum'(Y,Q,S) \\
plus(0,0,X) \leftarrow X = 0 \\
plus(0,1,X) \leftarrow X = 1 \\
\ldots
\end{array}
$$

When applied to the goal $sum([1,2],S)$ this program computes the sum of elements of the list $Y = [1,2]$ and assigns the result to $S$.

If such a program is regarded as a concurrent logic program in FCP then the atomic formula of the goal and those which are obtained by applying the clauses of this program to the goal can be viewed as processes which communicate via their variables (in [Sh 89] such variables are called logical ones). Each process of this type keeps trying to match its formula (that is the formula it corresponds to) with a clause head by a suitable substitution of terms for variables, and, if successful, it creates processes corresponding to the atomic formulas of the right hand side of the clause. This procedure may imply an instantiation of variables the process shares with other existing processes. Due to this, the processes awaiting for such an instantiation may advance.

For our program and goal we obtain the following computation:

$$
\begin{array}{l}
sum([1,2],S) \\
sum'([1,2],0,S) \\
plus(1,0,P), sum'([2],P,S) \\
plus(1,0,P), plus(2,P,Q), sum'([],Q,S) \\
P = 1, plus(2,P,Q), sum'([],Q,S) \\
P = 1, plus(2,P,Q), Q = S \\
P = 1, plus(2,P,S) \\
plus(2,1,S) \\
S = 3.
\end{array}
$$

In this computation $S, P, Q$ are variables. In order to find the required sum and assign it to $S$, process $sum([1,2],S)$ matches its formula with the head of the first clause and creates process $sum'([1,2],0,S)$. This process matches its formula with the third clause and creates two parallel processes $plus(1,0,P)$ and $sum'([2],P,S)$ which contain a new variable $P$. Now $plus(1,0,P)$ instantiates $P$ to 1 and $sum'([2],P,S)$ evolves into $plus(2,P,Q)$ and $sum'([],Q,S)$. Processes $plus(1,0,P)$ and $plus(2,P,Q)$ synchronize in the sense that $plus(2,P,Q)$ waits for instantiation of $P$ to 1 in order to instantiate $Q$. As, simultaneously, $S$ is instantiated to $Q$ due to the second clause, we obtain finally the required result $S = 3$.

In our approach each state of a computation of this type is represented by a hypergraph called a jungle as in [CMREL 91] and [CRP 91] (see section 2 for the concepts). For instance, the state $plus(1,0,P), sum'([2],P,S)$ is represented as shown in figure 1.1. In this representation hyperedges correspond to concrete occurrences of predicate and function symbols and nodes represent terms (more precisely, nodes are roots of subjungles which represent terms). For instance, the node $\sigma$ represents the term $2|[]$, that is the list [2].

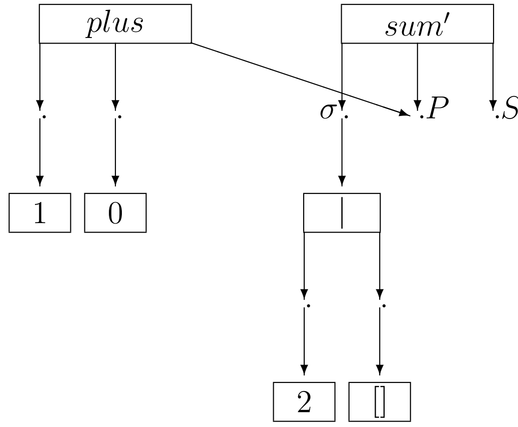



Figure 1.1

Processes which take part in a computation are present in it as subjungles which represent the respective atomic formulas. They are realized by calling, with suitable values of parameters, and executing, possibly many times, programs which specify how a process of a given class performs its step. Such a realization leads usually to parallel processes and then it appears as an interleaving of actions of the existing processes that is synchronized solely by instantiations of variables shared by processes.

For an illustration of this way of representing and realizing processes, let us consider process $plus(1, 0, P)$. Each step of this process can be realized by calling a program $PLUS(\xi, \eta, \zeta)$ with $\xi$ representing 1 and $\eta$ representing 0, and with $\zeta = P$. This program can be defined as:

$$\begin{array}{lrl} \Sigma x\ \Sigma y & (\textbf{set} & x, y\ \textbf{such that}\ \xi\ \textit{represents}\ x \text{ and } \eta\ \textit{represents}\ y; \\ & \textbf{if} & \text{not } (x = \textit{none} \text{ or } y = \textit{none}) \\ & \textbf{then} & \textit{replace plus}(\xi, \eta, \zeta)\ \textit{by}\ \zeta = x + y \\ & \textbf{else} & PLUS(\xi, \eta, \zeta)) \end{array}$$

where *replace* $plus(\xi, \eta, \zeta)$ *by* $\zeta = x + y$ denotes a reduction of the jungle which represents $plus(\xi, \eta, \zeta)$ (see the leftmost jungle in figure 2.4) to a single edge without target nodes, with a single source node $\zeta$, and with the colour equal to the sum of $x$ and $y$ (see the rightmost jungle in figure 2.4), and where $\Sigma x$ and $\Sigma y$ make $x$ and $y$ local to the program. Each call of $PLUS(\xi, \eta, \zeta)$ is an attempt of instantiating variables of terms represented by $\xi$, $\eta$, $\zeta$. If successful, it terminates process $plus(1, 0, P)$ with $P = x + y$. Otherwise it causes a subsequent call of $PLUS(\xi, \eta, \zeta)$ which may be viewed as a subsequent step of process $plus(1, 0, P)$ to be realized (possibly in parallel with other processes) after completing the current step.

Similarly, each step of process $sum'([1, 2], 0, S)$ can be realized by calling a program $SUM'(\xi, \eta, \zeta)$ with $\xi$ representing $[1, 2]$, $\eta$ representing 0, and with $\zeta = S$. This program can be defined as:

$\Sigma X\ \Sigma Y$ (**set** $X, Y$ **such that** $\xi$ *represents* $[X|Y]$;
**if** not ($X = none$ or $Y = none$)
**then** $\Sigma\varrho$ (*replace* $sum'(\xi, \eta, \zeta)$ *by* $plus(X, \eta, \varrho)$ *and* $sum'(Y, \varrho, \zeta)$;
($PLUS(X, \eta, \varrho) \parallel SUM'(Y, \varrho, \zeta)$))
**else** **if** $\xi$ *represents* []
**then** *replace* $sum'(\xi, \eta, \zeta)$ *by* $\zeta = \eta$
**else** $SUM'(\xi, \eta, \zeta)$)

where $(PLUS(X, \eta, \varrho) \parallel SUM'(Y, \varrho, \zeta))$ denotes the parallel interleaving execution of programs $PLUS(X, \eta, \varrho)$ and $SUM'(Y, \varrho, \zeta)$, and

$$replace\ sum'(\xi, \eta, \zeta)\ by\ plus(X, \eta, \varrho)\ and\ sum'(Y, \varrho, \zeta),$$

$$replace\ sum'(\xi, \eta, \zeta)\ by\ \zeta = \eta$$

are the operations of replacing the leftmost jungle by the rigthtmost one in figures 2.2 and 2.3, respectively. The call of $SUM'(\xi, \eta, \zeta)$ if $\xi$ represents the empty list [] may be viewed as a subsequent step of the realized process $sum'([1, 2], 0, S)$ whereas $PLUS(X, \eta, \varrho)$ and $SUM'(Y, \varrho, \zeta)$ start two new processes $plus(1, 0, P)$ and $sum'([2], P, S)$.

The present paper extends and improves previous work in [MW 83], [MW 91], [MW 92], and [MW 94]. It is organized as follows. In section 2 we recall and modify for our purposes the basic notions related to rewriting graphs. In section 3 we define programs of rewriting graphs. In sections 4 and 5 we present a structural operational semantics of these programs and a denotational semantics, respectively. In section 6 we describe input-output relations of programs.

# 2 Graphs, productions, and derivations

Let $\Lambda$ be a fixed many-sorted first-order language with equality which has sorts *nodes*, *edges*, *colours* with a common constant *none*, operation symbols
$1_source, ..., m_source, 1_target, ..., n_target : edges \to nodes$,
$edgecolour : edges \to colours$,
$\omega_1 : colours \times ... \times colours \to colours$, $\omega_2 : colours \times ... \times colours \to colours$, etc.,
and infinite, mutually disjoint sets *nodevariables*, *edgevariables*, *colourvariables* of node-, edge-, and colour variables, respectively.

Let $\Omega$ denote the signature consisting of the sorts and operation symbols of $\Lambda$ and $\Omega_0$ the part of $\Omega$ consisting of the sort *colours* and operation symbols $\omega_1$, $\omega_2$, ... .

The constant *none* denotes, in each context, a representative of the lack of a suitable object.

By an $\Omega$-*graph* (or a *graph*) we mean an $\Omega$- algebra $G$ such that $i_source_G(x) = none_G$ implies $j_source_G(x) = none_G$ for $j \geq i$ and $i_target_G(x) = none_G$ implies $j_target_G(x) = none_G$ for $j \geq i$.

Such a graph consists of *nodes* (elements of the set $nodes_G$) and *edges* (elements of the set $edges_G$), each edge $x$ with a *colour* $edgecolour_G(x)$ (an element of an $\Omega_0$-algebra on the set $colours_G$), with a sequence $source_G(x)$ of *source nodes* (the sequence of subsequent different from $none_G$ elements of the sequence $1_source_G(x), 2_source_G(x), ..., m_source_G(x)$), and with a sequence $target_G(x)$ of *target nodes* (the sequence of subsequent different from $none_G$ elements of the sequence $1_target_G(x), 2_target_G(x), ..., n_target_G(x)$). A sequence of the form $x_1e_1x_2...x_ke_kx_{k+1}$, where each $e_i$ is an edge with a source equal to $x_i \neq none_G$ and a target equal to $x_{i+1} \neq none_G$, and where $x_{k+1} = x_1$, is called a *cycle* and we say that $G$ is *acyclic* if such a sequence does not exist.

An example of a graph is shown in figure 1.1. Dots represent nodes, boxes represent edges, inscriptions in boxes represent colours, directed lines from dots to boxes (resp.: from boxes to dots) indicate source nodes (resp.: target nodes) of the respective edges (the order of source and target nodes corresponds to that from left to right). In this example colours are predicate symbols and they are represented by strings which should be regarded as constants, that is operations of the arity 0. Thus the corresponding $\Omega_0$-algebra reduces to a set of predicate symbols with a set constants, each constant representing a concrete predicate symbol. In general, it may be convenient to have some nontrivial operations on colours and thus a nontrivial $\Omega_0$-algebra of colours. We call such an algebra the $\Omega_0$-*reduct* of $G$ and write it as $\Omega_0_reduct(G)$. A typical example is $\Omega_0_terms$, the algebra of $\Omega_0$-terms with variables from the set $colourvariables$.

Of course, instead of one sort *colours* one might consider more such sorts and the respective multisorted algebra.

By an $\Omega$-*homomorphism* (or a *homomorphism*) from an $\Omega$-graph $G$ to an $\Omega$-graph $G'$ we mean any homomorphism $h : G \to G'$, written also as $G \xrightarrow{h} G'$, from the $\Omega$-algebra $G$ to the $\Omega$-algebra $G'$, and by $h_{nodes}$, $h_{edges}$, $h_{colours}$, we denote the components of $h$ corresponding to the sorts *nodes*, *edges*, *colours*, respectively. Such a homomorphism is called an *isomorphism* if its components are bijective and have inverses which constitute an $\Omega$-homomorphism.

For $\Omega$-homomorphisms $h : G \to G'$ and $h' : G' \to G''$ there exists a unique $\Omega$-homomorphism $hh' : G \to G''$ defined by $(hh')_{nodes}(x) = h'_{nodes}(h_{nodes}(x))$, $(hh')_{edges}(x) = h'_{edges}(h_{edges}(x))$, $(hh')_{colours}(x) = h'_{colours}(h_{colours}(x))$. We call it the *composition* of $h$ and $h'$.

Under some conditions two given $\Omega$-graphs $L$ and $D$ with two given $\Omega$-homomorphisms $l : K \to L$ and $d : K \to D$ from a given $\Omega$-graph $K$ can be combined into an $\Omega$-graph $G$ by gluing the image of $K$ under $l$ to the image of $K$ under $d$. This can be described formally as follows.

**2.1. Proposition.** For each pair $(L \xleftarrow{l} K \xrightarrow{d} D)$ of homomorphisms of $\Omega$-graphs such that $\Omega_0_reduct(K) = \Omega_0_reduct(L) = \Omega_0_terms$ and $l_{colours} : \Omega_0_reduct(K) \to \Omega_0_reduct(L)$ is the identity there exists a pair $(L \xrightarrow{g} G \xleftarrow{b} D)$ of homomorphisms of $\Omega$-graphs such that $b_s(x) = x$ for each sort $s$ and each $x \in s_D - d_s(s_K)$ and the following conditions are satisfied:

(1) $lg = db$,

(2) for each pair $(L \xrightarrow{g'} G' \xleftarrow{b'} D)$ of homomorphisms of $\Omega$-graphs such that $lg' = db'$

there exists a unique $\Omega$-homomorphism $h : G \to G'$ such that $gh = g'$ and $bh = b'$. We call such a pair $(L \xrightarrow{g} G \xleftarrow{b} D)$ and the $\Omega$-graph $G$ respectively a *natural pushout* and a *natural pushout object* of $(L \xleftarrow{l} K \xrightarrow{d} D)$ . $\Box$

Proof: Choose a homomorphism $i : L \to L'$ such that:

- $\Omega_0_reduct(L') = \Omega_0_reduct(D)$,
- $i_s(l_s(x)) = d_s(x)$ for each sort $s$ and each $x$ in $K$,
- $i_s(x) = i_s(y)$ implies $x = y$ or $x, y \in l_s(s_K)$ for each sort $s$ and all $x, y \in s_L$,
- $i_s(x)$ is not in $s_D$ for each $x$ not in $l_s(s_K)$.

Define:

$$nodes_G = nodes_D \cup (nodes_{L'} - i_{nodes}(l_{nodes}(nodes_K)))$$
$$edges_G = edges_D \cup (edges_{L'} - i_{edges}(l_{edges}(edges_K))),$$
$$source_G(x) = source_D(x) \text{ and } target_G(x) = target_D(x) \text{ and}$$
$$edgecolour_G(x) = edgecolour_D(x) \text{ for } x \in edges_D,$$
$$source_G(x) = source_{L'}(x) \text{ and } target_G(x) = target_{L'}(x) \text{ and}$$
$$edgecolour_G(x) = edgecolour_{L'}(x) \text{ for } x \in edges_{L'} - edges_D,$$
$$g_s(x) = i_s(x) \text{ for each sort } s \text{ and each } x \in s_L$$

$$b_s(x) = \begin{cases} x & \text{for each } x \in s_D - d_s(s_K) \\ d_s(y) & \text{for each } x = l_s(y) \text{ with } y \in s_K \end{cases}$$

$$h_s(x) = \begin{cases} b'_s(x) & \text{for each } x \in s_D \\ g'_s(i_s^{-1}(x)) & \text{for each } x \in s_{L'} - s_D \end{cases}$$

A straightforward verification shows that in this manner we obtain $G$, $g$, $b$, $h$ as required. $\Box$

To the just described operation of combining graphs an operation of reducing graphs corresponds.

Namely, under some conditions a subgraph $D$ of a given $\Omega$-graph $G$ and a homomorphism $d : K \to D$ from a given $\Omega$-graph $K$ to $D$ can be found with the property that by combining a given $\Omega$-graph $L$ with $D$ such that the image of $K$ in $L$ under a given homomorphism $l : K \to L$ is glued to the image of $K$ under $d$ one obtains $G$. This can be described formally as follows.

**2.2. Proposition.** For each pair $(K \xrightarrow{l} L \xrightarrow{g} G)$ of homomorphisms of $\Omega$-graphs such that $\Omega_0_reduct(K)$ and $\Omega_0_reduct(L)$ are equal to $\Omega_0_terms$ with $l_{colours} : \Omega_0_reduct(K) \to \Omega_0_reduct(L)$ being the identity, edges from $edges_G - g_{edges}(edges_L)$ have all their source

and target nodes in $(nodes_G - g_{nodes}(nodes_L)) \cup g_{nodes}(l_{nodes}(nodes_K))$, and $g_s(x) = g_s(y)$ implies $x = y$ or $x, y \in l_s(s_K)$ for each sort $s$ and all $x, y \in s_L$, there exists a pair $(K \xrightarrow{d} D \xrightarrow{b} G)$ such that $(L \xrightarrow{g} G \xleftarrow{b} D)$ is a natural pushout of $(L \xleftarrow{l} K \xrightarrow{d} D)$. We call such a pair $(K \xrightarrow{d} D \xrightarrow{b} G)$ and the $\Omega$-graph $D$ respectively a *natural pushout complement* and a *natural pushout complement object* of $(K \xrightarrow{l} L \xrightarrow{g} G)$. □

Proof: Define

$$nodes_D = (nodes_G - g_{nodes}(nodes_L)) \cup g_{nodes}(l_{nodes}(nodes_K)),$$

$$edges_D = (edges_G - g_{edges}(edges_L)) \cup g_{edges}(l_{edges}(edges_K)),$$

$$\Omega_0_reduct(D) = \Omega_0_reduct(G),$$

$$source_D(x) = source_G(x) \text{ and } target_D(x) = target_G(x) \text{ and}$$

$$edgecolour_D(x) = edgecolour_G(x) \text{ for } x \in edges_D,$$

$$d_s(x) = g_s(l_s(x)) \text{ for each sort } s \text{ and each } x \in s_K,$$

$$b_s(x) = \begin{cases} x & \text{for each } x \in s_D - d_s(s_K) \\ d_s(y) & \text{for each } x = l_s(y) \text{ with } y \in s_K \end{cases}$$

A straightforward verification shows that in this manner we obtain $D$, $d$, $b$ as required. □

Natural pushouts and pushout complements as described in 2.1 and 2.2 are pushouts and pushout complements in the sense of category theory. More precisely, they are pushouts and pushout complements in the category of $\Omega$-graphs and their homomorphisms. We denote this category by $\Omega$*_graphs*.

The category $\Omega$*_graphs* may be too large for some purposes. In paticular, it is too large as a universe for term-rewriting systems and logic programming with terms and atomic formulas represented by special graphs, called *jungles* (cf. [CMREL 91], [CRP 91], and [CR 93].

A jungle is an acyclic (hyper)graph in which each node may be a source of at most one edge, and each edge is coloured by a predicate or function symbol and it has respectively no or one source node and as many target nodes as specified by the arity of the respective symbol. An edge with a function symbol and its source node represent a term with this function symbol and with arguments represented by target nodes. An edge with a predicate symbol represents an atomic formula with this predicate symbol and with arguments represented by target nodes. In the sequel we shall assume that the considered predicate and function symbols are (constants denoting) colours, and that the considered jungles are $\Omega$-graphs, called $\Omega$- *jungles*. The $\Omega$-jungles and homomorphisms between $\Omega$-jungles constitute a subcategory $\Omega$*_jungles* of the category $\Omega$*_graphs*.

Though $\Omega$-jungles are $\Omega$-graphs, and thus each pair $\pi = (L \xleftarrow{l} K \xrightarrow{d} D)$ of homomorphisms of $\Omega$-jungles as in 2.1 has a natural pushout $(L \xrightarrow{g} G \xleftarrow{b} D)$ in $\Omega$*_graphs*, such a pushout does not need to be a pushout in $\Omega$*_jungles* since the pushout object $G$ does not need to be a jungle.

There may be three reasons of such a situation. Firstly, $G$ may contain a cycle and then $\pi$ has no pushout $(L \stackrel{g'}{\rightarrow} G' \stackrel{b'}{\leftarrow} D)$ in $\Omega$*_jungles* since otherwise there would be a homomorphism from $G$ to $G'$ and hence $G'$ would contain a cycle. Secondly, $G$ may contain nodes which are sources of two edges and such nodes cannot be made sources of single edges by identifying edges with the same sources and function symbols and the corresponding target nodes of such edges. Acoording to [CR 93], this corresponds to the lack of a unifier of terms represented by the same node. Thirdly, $G$ may contain nodes which are sources of two edges and such nodes can be made sources of single edges by identifying edges with the same sources and function symbols and the corresponding target nodes of such edges. This corresponds to the existence of a unifier for each pair of terms consisting of terms represented by the same node, and it allows to reduce $G$ to a jungle $G'$ such that there is a surjective homomorphism $f : G \rightarrow G'$ and $(L \stackrel{gf}{\rightarrow} G' \stackrel{bf}{\leftarrow} D)$ is a pushout in $\Omega$*_jungles*. Moreover, the reduction of $G$ by identification of some nodes and edges can be made such that the resulting nodes and edges coincide with those from $D$, that is $f_s(b_s(x)) = f_s(x) = x$ for each sort $s$ and each $x$ from $D$ of this sort. This means that the components of $bf$ are inclusions and thus the obtained pushout may be regarded as natural, in a sense similar to that in 2.1.

In the case of pushout complements in $\Omega$*_jungles* the situation is much simpler. For each pair $\varrho = (K \stackrel{l}{\rightarrow} L \stackrel{g}{\rightarrow} G)$ of homomorphisms of $\Omega$-jungles such that a natural pushout complement $\sigma = (K \stackrel{d}{\rightarrow} D \stackrel{b}{\rightarrow} G)$ exists in $\Omega$*_graphs* from the construction of $D$ we obtain that $D$ is an $\Omega$-jungle. Consequently, $\sigma$ can be characterized as in 2.2 with $\Omega$-jungles instead of $\Omega$-graphs and thus regarded as a natural pushout complement in $\Omega$*_jungles*.

In general, for some purposes we may need some other subcategories of the category of $\Omega$-graphs. So, in the sequel we shall relate our formalism to an arbitrary but fixed subcategory $C$ of $\Omega$-graphs in which morphisms between objects are homomorphisms of $\Omega$-graphs and pushouts and pushout complements are defined by conditions similar to those in 2.1 and 2.2, respectively.

Productions representing rewriting rules for $\Omega$-graphs can be defined as follows.

By a *production* we mean any $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, where $L$, $K$, $R$ are $\Omega$-graphs in the subcategory $C$ with finite sets of nodes and edges and the $\Omega_0$-reducts coinciding with $\Omega_0$*_terms*, the $\Omega_0$-algebra of terms, and $l : K \rightarrow L$, $r : K \rightarrow R$ are homomorphisms with $l_{colours}$ and $r_{colours}$ being identities.

We call $K$ the *gluing graph* of $p$, and we call $L$ and $R$ the *left side* and the *right side* of $p$, respectively.

In order to be able to enforce some elements of a production to be instantiated by given objects (nodes, or edges, or colours of a graph), we associate with such elements variables which may denote the respective objects. The variables thus associated are called *parameters*, and the respective production is called a *parametrized production.*

Formally, by a parametrized production we mean any $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R, m, n)$, where $pr_p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ is a production and $m$ and $n$ are triples $m = (m_{nodes}, m_{edges}, m_{colours})$ and $n = (n_{nodes}, n_{edges}, n_{colours})$ of partial mappings

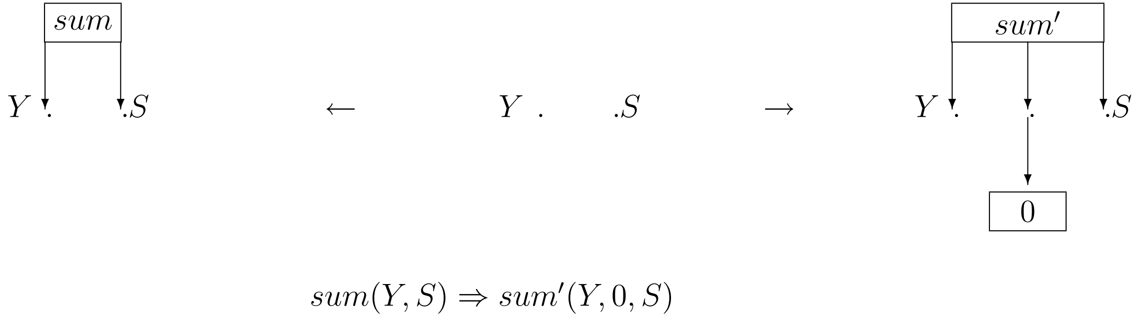
$$m_{nodes} : nodes_L \supseteq\!\!\rightarrow nodevariables$$
$$m_{edges} : edges_L \supseteq\!\!\rightarrow edgevariables$$

$m_{colours} : colours_L \supseteq\!\!\rightarrow colourvariables$
$n_{nodes} : nodes_R \supseteq\!\!\rightarrow nodevariables$
$n_{edges} : edges_R \supseteq\!\!\rightarrow edgevariables$
$n_{colours} : colours_R \supseteq\!\!\rightarrow colourvariables$

such that $m_{colours}$, $n_{colours}$ are respectively an inclusion of a subset of colour variables occurring in $\Omega_0$-terms assigned to edges of $L$ and an inclusion of a subset of colour variables occurring in $\Omega_0$-terms assigned to edges of $R$.

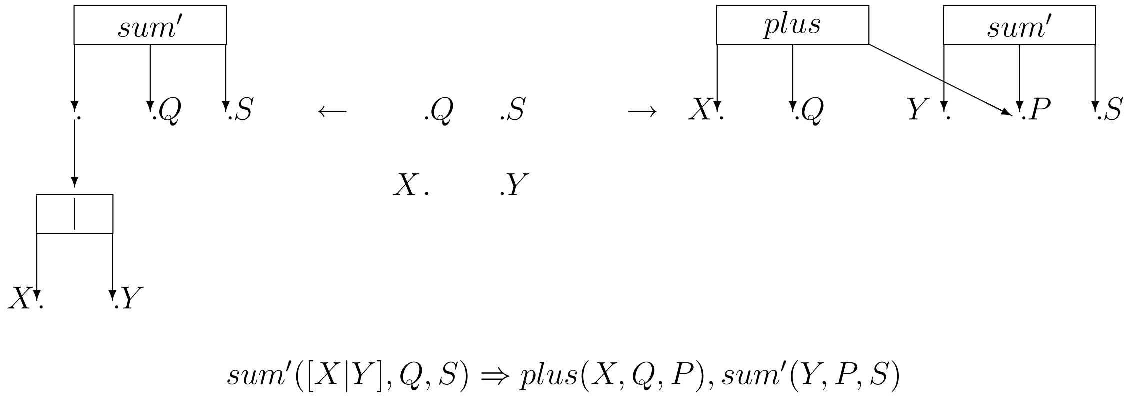
Values of mappings $m_{nodes}$, $n_{nodes}$, $m_{edges}$, $n_{edges}$, $m_{colours}$, $n_{colours}$ are called *node-*, *edge-*, and *colour parameters* of $p$, respectively.

**2.5. Example.** In the case of the logic program in 1.1 atomic formulas corresponding to processes can be rewritten in the subcategory of $\Omega$-jungles according to the parametrized productions in figures 2.1 - 2.4, where inscriptions in boxes play the role of colours and labels at dots play the role of parameters. In particular, the inscription $a+b$ in figure 2.4 may be regarded as an $\Omega_0$-term for $\Omega_0$ containing the operation symbol $+$. □




$$sum(Y, S) \Rightarrow sum'(Y, 0, S)$$

Figure 2.1




$$sum'([X|Y], Q, S) \Rightarrow plus(X, Q, P), sum'(Y, P, S)$$

Figure 2.2

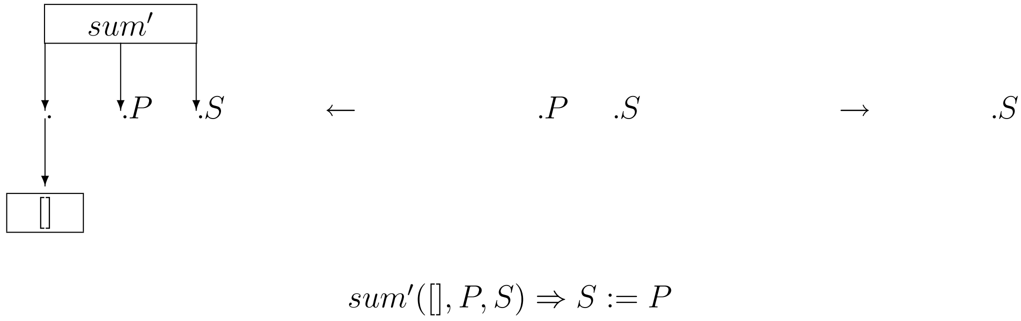



Figure 2.3

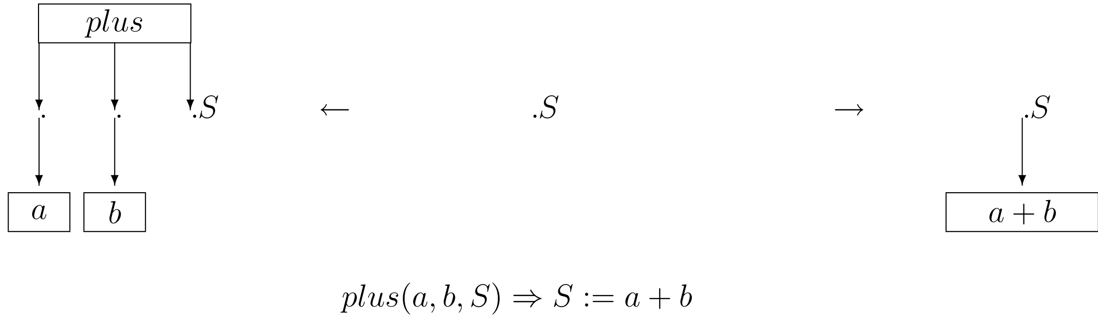



Figure 2.4

Applications of usual productions to $\Omega$-graphs from the subcategory $C$ can be defined following the standard algebraic approach. An application of a parametrized production $p$ can be defined as an application of the usual production $pr_p$ in which elements with associated parameters are instantiated in a specific manner.

Let $A$ be a fixed $\Omega_0$-algebra, possibly equipped with some relations whose symbols belong to the language $\Lambda$.

By a *rewriting step* (or a *direct derivation*) over A via a parametrized production $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R, m, n)$, we mean a pair $\sigma = (p, i)$ which consists of $p$ and of a diagram $i$ as in figure 2.5 in the subcategory $C$ of the category $\Omega_graphs$ such that

(1) $(K \stackrel{d}{\rightarrow} D \stackrel{b}{\rightarrow} G)$ is a natural pushout complement of $(K \stackrel{l}{\rightarrow} L \stackrel{g}{\rightarrow} G)$,

(2) $(D \stackrel{c}{\rightarrow} H \stackrel{h}{\leftarrow} R)$ is a natural pushout of $(D \stackrel{d}{\leftarrow} K \stackrel{r}{\rightarrow} R)$,

(3) $\Omega_0$-reducts of $G$, $D$, $H$ coincide with $A$,

(4) $b_{colours}$ and $c_{colours}$ are identities,

(5) for each sort $s$, we have:

$m_s(x) = m_s(y)$ implies $g_s(x) = g_s(y)$ whenever $m_s(x)$ and $m_s(y)$ are defined,

$n_s(x) = n_s(y)$ implies $h_s(x) = h_s(y)$ whenever $n_s(x)$ and $n_s(y)$ are defined.

We say that $\sigma$ *rewrites* $G$ into $H$ and write it as $G \overset{\sigma}{\Rightarrow} H$.

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle h} \\ G & \xleftarrow{\ b\ } & D & \xrightarrow{\ c\ } & H \end{array}$$

Figure 2.5

A rewriting step should be regarded as an action which transforms a graph $G$ and a partial valuation $v$ of variables in $G$ into a graph $H$ and a valuation $w$ of variables in $H$.

The pair $(G, v)$ plays here the role of data. It consists of the fixed $\Omega_0$-algebra $A$ of colours and of a proper graph which may be transformed. The valuation $v$ may be interpreted as an assignment of nodes, edges, and colours from $G$ to node-, edge-, and colour variables, respectively. Both the domain of such a valuation and the values it assigns to variables may be transformed.

The transformation of $(G, v)$ into $(H, w)$ is realized by finding the respective natural pushout complement $(K \xrightarrow{d} D \xrightarrow{b} G)$ of $(K \xrightarrow{l} L \xrightarrow{g} G)$ and the respective natural pushout $(D \xrightarrow{c} H \xleftarrow{h} R)$ of $(D \xleftarrow{d} K \xrightarrow{r} R)$ such that

$v(u) = g_s(x)$ if $u = m_s(x)$

and then defining:

$w(u) = h_s(x)$ with $x \in n_s^{-1}(u)$ for $u$ such that $n_s^{-1}(u) \neq \emptyset$,

$w(u) = \mathbf{none}_H$ for $u$ such that $m_s^{-1}(u) \neq \emptyset$ and $n_s^{-1}(u) = \emptyset$,

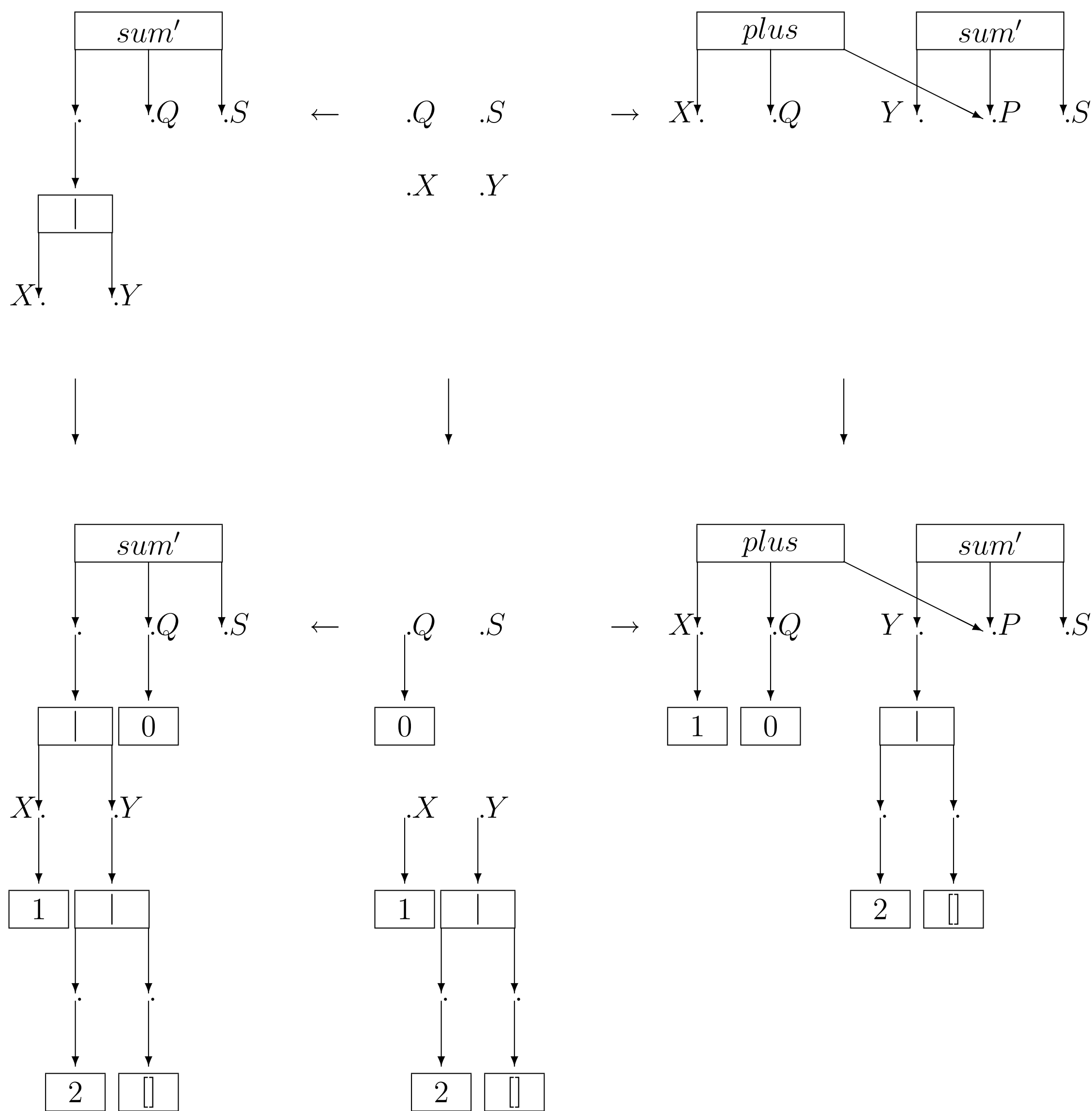
$w(u) = v(u)$ for $u$ such that $m_s^{-1}(u) = \emptyset$ and $n_s^{-1}(u) = \emptyset$.

Formally, we say that the parametrized production $p$ (and the respective rewriting step $(p, i)$) *transforms* $(G, v)$ into $(H, w)$ if such a realization is possible, that is if there exist $(K \xrightarrow{d} D \xrightarrow{b} G)$ and $(D \xrightarrow{c} H \xleftarrow{h} R)$ as described.

The conditions in (5) ensure that this definition is correct. The values of $v$ for parameters associated with elements of the left side of $p$ determine the values of the homomorphism $g$ for the represented elements. For parameters which are associated only with elements of the left side of $p$ the respective values of $w$ become $\mathbf{none}_H$, i.e., undefined. For parameters associated with elements of the right side of $p$ the respective values of $w$ are determined as the values of the homomorphism $h$ for the represented elements.

The values of of the parameters that are not involved in the considered application of $p$ remain unchanged. The naturality of the considered pushout complement and pushout is assumed in order to ensure that the part of $G$ which does not need to be transformed by the production remains unchanged. We want to ensure these properties because they are essential for reasoning about sequences of rewriting steps.

**2.6. Example.** The replacement of $sum'([1|2], 0, S)$ by $plus(1, 0, P)$ and $sum'([2], P, S)$ can be regarded as a rewriting step in the category of $\Omega$-jungles via the parametrized production in figure 2.2. This step is shown in figure 2.6. The applied production transforms the jungle representing $sum'([1|2], 0, S)$ and the indicated valuation of $X, Y, P, Q, S$ in this jungle into a jungle representing $plus(1, 0, P)$ and $sum'([2], P, S)$ and the indicated valuation of $X, Y, P, Q, S$ in this jungle. □




$sum'([1|2], 0, S) \Rightarrow plus(1, 0, P), sum'([2], P, S)$

Figure 2.6

The concept of a direct derivation can be easily generalized.

Given a set $\Pi$ of parametrized productions, by a *derivation* over $A$ via productions from $\Pi$ we mean a finite sequence $\sigma = (G_0 \overset{\sigma_1}{\Rightarrow} G_1 \overset{\sigma_2}{\Rightarrow} ... \overset{\sigma_i}{\Rightarrow} G_i)$ of rewriting steps over $A$ via productions from $\Pi$. Given such a sequence $\sigma$, we say that it *rewrites* $G_0$ into $G_i$ and write $G_0 \overset{\sigma}{\Rightarrow}^\star G_i$. By $Der_{C,A,\Pi}$ we denote the set of derivations of this kind with $G_0, G_1, ..., G_i$ having finite sets of nodes and edges.

Given a set $X \subseteq Der_{C,A,\Pi}$ of derivations, by the *relation of derivability* via derivations from $X$ we mean the following relation $rel(X)$ between graphs:

$$(G, H) \in rel(X) \text{ iff } G \overset{\sigma}{\Rightarrow}^\star H \text{ for some } \sigma \in X.$$

# 3 Programs

We are interested in rewriting graphs according to some programs.

Intuitively, a program $p$ we have in mind is a description, possibly with some parameters, of an algorithm of rewriting graphs by applying productions. In particular, it describes how a given graph $G$ and a given partial valuation $v$ of variables in this graph, which is defined for parameters of $p$, are transformed into subsequent graphs and valuations of variables until reaching a final result.

In order to facilitate a sort of busy waiting of processes as mentioned in section 1, we admit a recursion such that programs may call themselves without executing any real action (an unguarded recursion). Theoretically it leads to infinite idle loops, but, in practice, such loops do not happen, due to a sort of fairness which is usually ensured.

Programs are defined presupposing a set $programidentifiers$ of program identifiers, each identifier with an arity which specifies a number of node-, edge-, and colour parameters. They are given by *program expressions* $p$, $q$, $r$,..., which are of the following kinds:

(1) A constant **nil**. This program expression represents doing nothing.

(2) A parametrized production $p$. This program expression represents a possible rewriting step $\sigma = (p, i)$ with $i$ as in figure 2.5 which transforms a graph $G$ and a valuation $v$ of parameters of $p$ in $G$ into a graph $H$ and a valuation $w$ of parameters of $p$ in $H$ in the sense defined in section 2.

(3) An expression of the form **set** $x, y, ...$ **such that** $f$, where $x, y, ...$ are variables and $f$ is a formula in the language $\Lambda$. This program expression represents a search for a revaluation of $x, y, ...$ in $G$ such that $f$ is satisfied for the resulting valuation $v'$ (if no values of $x, y, ...$ can be found such that $f$ is satisfied then $x, y, ...$ are set to $none_G$).

(4) The result $p\gamma$ of an injective sort-preserving substitution $\gamma$ of new node- and edge variables for node- and edge parameters, respectively, and $\Omega_0$-terms for colour parameters, in a program expression $p$. This program expression represents an activity which transforms a graph $G$ and a valuation $v$ of parameters of $p\gamma$ in the way in which the activity represented by $p$ transforms $G$ and the valuation $\gamma v$, i.e. the composition of $\gamma$ and $v$ defined by $(\gamma v)(x) = v(\gamma(x))$.

(5) A conditional **if** $f$ **then** $p$ **else** $q$, where $f$ is a formula in the language $\Lambda$ and $p$, $q$ are program expressions. This program expression represents the choice and execution of $p$ or $q$ depending on the satisfaction of $f$ for the given graph $G$ and the given valuation $v$ of free variables of $f$ and parameters of $p$ and $q$.

(6) A sequential composition $p; q$ of program expressions $p$ and $q$. This program expression represents an execution of $p$ followed by an execution of $q$.

(7) A parallel composition $p \parallel q$ of program expressions $p$ and $q$. This program expression represents a parallel execution of $p$ and $q$ which can be viewed as an arbitrary interleaving of actions of $p$ and $q$.

(8) An indeterministic sum $p+q$ of program expressions $p$ and $q$. This program expression represents an indeterministic choice and execution of $p$ or $q$.

(9) The result $\Sigma x\ p$ of binding a parameter $x$ in a program expression $p$. This program expression represents the activity of substituting a suitable variable which does not belong to the domain of the current valuation $v$ of variables in the respective graph $G$ for each unbound occurrence of $x$ in $p$, and of assigning the value $none_G$ to this variable.

(10) An atomic program expression **atom** $p$, where $p$ is a program expression. This program expression represents the activity of successfully executing $p$ as one indivisible step.

(11) A defined program expression

$$\varphi_k(y_{k1}, y_{k2}, ...) \textbf{ where } (\varphi_1(y_{11}, y_{12}, ...) = \psi_1, ..., \varphi_n(y_{n1}, y_{n2}, ...) = \psi_n),$$

where $\varphi_1, ..., \varphi_n$ are program identifiers and $y_{11}, y_{12}, ..., y_{n1}, y_{n2}, ...$ are parameters as specified by the respective arities and each $\psi_i$ is a program expression which may contain expressions of the form $\varphi_j\gamma$, where $\gamma$ stands for a substitution, and is such that all parameters of $\psi_i$ occur among $y_{i1}, y_{i2}, ...$. This program expression represents an activity whose execution for $y_{k1}, y_{k2}, ...$ is defined by $\psi_k$. As in $\psi_k$ there may occur expressions of the form $\varphi_i\gamma$, one has to define the respective activities by the equations $\varphi_1(y_{11}, y_{12}, ...) = \psi_1, ..., \varphi_n(y_{n1}, y_{n2}, ...) = \psi_n$, and consider an occurrence of $\varphi_j\gamma$ in $\psi_i$ as a call of the respective $\psi_j$. In particular, each program expression of the form

$\varphi$ **where** ($\varphi =$ **if** $f$ **then** $p; \varphi$ **else nil**)

is equivalent to the standard iteration construct **while** $f$ **do** $p$.

Thus we have the following syntax of program expressions:

$p ::= \ \mathbf{nil}$
$\quad \mid < \textit{parametrized production} >$
$\quad \mid \mathbf{set}\ x, y, ...\ \mathbf{such\ that}\ f$
$\quad \mid p\gamma$
$\quad \mid \mathbf{if}\ f\ \mathbf{then}\ p\ \mathbf{else}\ q$
$\quad \mid p; q$
$\quad \mid p \parallel q$
$\quad \mid p + q$
$\quad \mid \Sigma x\ p$
$\quad \mid \mathbf{atom}\ p$
$\quad \mid \varphi_k(y_{k1}, y_{k2}, ...)\ \mathbf{where}\ (\varphi_1 = \psi_1, ..., \varphi_n = \psi_n)$

To each program expression $p$ a set $FP(p)$ of node-, edge-, and colour variables, called *free parameters* of $p$, corresponds which can be defined as follows:

(1) $FP(\mathbf{nil}) = \emptyset$.

(2) If $p$ is a parametrized production then $FP(p)$ is the set of parameters of $p$.

(3) $FP(\mathbf{set}\ x, y, ...\ \mathbf{such\ that}\ f)$ is the union of the set $\{x, y, ...\}$ and the set of free variables of $f$.

(4) $FP(p\ \gamma) = \gamma(FP(p))$.

(5) $FP(\mathbf{if}\ f\ \mathbf{then}\ p\ \mathbf{else}\ q)$ is the union of the set $FP(p) \cup FP(q)$ and the set of free variables of $f$.

(6) $FP(p; q) = FP(p \parallel q) = FP(p + q) = FP(p) \cup FP(q)$.

(7) $FP(\Sigma x\ p) = FP(p) - \{x\}$.

(8) $FP(\mathbf{atom}\ p) = FP(p)$.

(9) $FP(\varphi_k(y_{k1}, y_{k2}, ...)\ \mathbf{where}\ (\varphi_1 = \psi_1, ..., \varphi_n = \psi_n)) = \{y_{k1}, y_{k2}, ...\}$.

**3.1. Example.** A program of computing the sum of elements of a list of integers as in 1.1 can be defined as follows:

$$
\begin{array}{lllll}
SUM(\xi,\zeta) & \textbf{where} & & & \\
(SUM(\xi,\zeta) & = & & & \textit{replace sum}(\xi,\zeta)\ \textit{by sum}'(\xi,0,\zeta); \\
 & & & & SUM'(\xi,0,\zeta), \\
SUM'(\xi,\eta,\zeta) & = & \Sigma X\ \Sigma Y & (\textbf{set} & X,Y\ \textbf{such that}\ \xi\ \textit{represents}\ [X|Y]; \\
 & & & \textbf{if} & \text{not } (X = \textit{none} \text{ or } Y = \textit{none}) \\
 & & & \textbf{then} & \Sigma\varrho\ (\textit{replace sum}'(\xi,\eta,\zeta) \\
 & & & & \textit{by plus}(X,\eta,\varrho)\ \textit{and sum}'(Y,\varrho,\zeta); \\
 & & & & (PLUS(X,\eta,\varrho) \parallel SUM'(Y,\varrho,\zeta))) \\
 & & & \textbf{else} & \textbf{if}\ \xi\ \textit{represents}\ [] \\
 & & & & \textbf{then}\ \textit{replace sum}'(\xi,\eta,\zeta)\ \textit{by}\ \zeta = \eta \\
 & & & & \textbf{else}\ SUM'(\xi,\eta,\zeta)), \\
PLUS(\xi,\eta,\zeta) & = & \Sigma x\ \Sigma y & (\textbf{set} & x,y\ \textbf{such that}\ \xi\ \textit{represents}\ x \\
 & & & & \text{and } \eta\ \textit{represents}\ y; \\
 & & & \textbf{if} & \text{not } (x = \textit{none} \text{ or } y = \textit{none}) \\
 & & & \textbf{then} & \textit{replace plus}(\xi,\eta,\zeta)\ \textit{by}\ \zeta = x + y \\
 & & & \textbf{else} & PLUS(\xi,\eta,\zeta)))
\end{array}
$$

The instructions of replacement occurring in this program are parametrized productions which can be obtained from those in figures 2.1 - 2.4 by suitable substitutions of variables for variables. Formulas occurring in the program are abbreviations of formulas of the first order language $\Lambda$. For instance, the formula $\xi$ *represents* $[X|Y]$ is an abbreviation of a formula which states the existence of an edge with the colour $|$, the source node $\xi$, and the target nodes $X$ and $Y$.

# 4 Operational semantics

The way in which pairs consisting of graphs and valuations of variables in these graphs are transformed by executing programs can be described in the form of a labelled transition system which consists of a universe $conf$ of configurations and a transition relation $\rightarrow$. When considered together with suitable fairness assumptions, such a system allows to define all practically possible program computations.

The universe $conf$ consists of *configurations* of the form $c = (p, G, v)$, where $p$ is a program expression, $G$ is a graph, and $v$ is a partial valuation of variables in $G$ such that the defined values of node-, edge-, and colour variables are respectively nodes, edges, and colours of $G$. The program expression $p$ represents a control whereas the graph $G$ and the valuation $v$ represent data. For technical convenience we assume that each of these items may be undefined and then represented by the special symbol ?. For the same reason we abstract from inessential syntactic details of program expressions and we consider as identical program expressions as $p \parallel q$ and $q \parallel p$, $\mathbf{nil} \parallel p$ and $p$, $\mathbf{nil}; p$ and $p$, etc.

Configurations containing undefined items are said to be *unproper*, whereas all the other configurations are said to be *proper*. Among proper configurations we distinguish *terminal* ones which are defined as follows:

(1) $(\mathbf{nil}, G, v)$ is terminal,

(2) $(p\gamma, G, v)$ is terminal whenever $(p, G, \gamma v)$ is terminal,

(3) (**if** $f$ **then** $p$ **else** $q, G, v$) is terminal whenever $(p, G, v)$ is terminal and $f$ is satisfied for $G$ and $v$, or $(q, G, v)$ is terminal and $f$ is not satisfied for $G$ and $v$,

(4) $(p; q, G, v)$ is terminal whenever $(p, G, v)$ is terminal and $(q, G, v)$ is terminal,

(5) $(p \parallel q, G, v)$ is terminal whenever $(p, G, v)$ is terminal and $(q, G, v)$ is terminal,

(6) $(p + q, G, v)$ is terminal whenever $(p, G, v)$ is terminal and $(q, G, v)$ is terminal,

(7) (**atom** $p, G, v$) is terminal whenever $(p, G, v)$ is terminal.

The transition relation $\rightarrow$ consists of triples of the form $c \xrightarrow{\alpha} c'$, called *transitions*, where $c, c'$ are configurations such that $c$ is proper, and $\alpha$ is either an invisible action $\tau$ which does not change data (that is such that $G = G'$ and $v = v'$ for $c = (p, G, v)$ and $c' = (p', G', v')$), or an action $\Sigma x$ of binding a parameter $x$ in a program, or an action of applying a production or executing a program of the form **set** $x, y, ...$**such that** $f$ or an atomic program. Denoting by *actions* the set of possible actions we can define the transition relation as the smallest $\rightarrow \subseteq conf \times actions \times conf$ satisfying the following conditions:

(1) $(p, G, v) \xrightarrow{p} (\mathbf{nil}, H, w)$ for each parametrized production $p$ and $H, w$ such that $p$ transforms $(G, v)$ into $(H, w)$ in the sense defined in section 2.

(2) $(p, G, v) \xrightarrow{p} (?, G, v)$ for each parametrized production $p$ and $G, v$ such that there are no $H, w$ as in (1).

(3) $(p, G, v) \xrightarrow{p} (\mathbf{nil}, H, w)$ for each program expression $p$ of the form

**set** $x, y, ...$**such that** $f$

and $G, H, v, w$ such that $H = G$ and $w$ is obtained from $v$ by modifying $v(x), v(y), ...$ such that the formula $f$ is satisfied for the new valuation $w$, or by replacing $v(x), v(y), ...$ by $none_G$ if such a modification is impossible.

(4) If $(p, G, \gamma v) \xrightarrow{\alpha} (p', G', \gamma v')$ for an injective sort-preserving substitution of variables for variables then $(p\gamma, G, v) \xrightarrow{\alpha\gamma} (p'\gamma, G', v')$, where $\tau\gamma = \tau$, $\Sigma x\gamma = \Sigma\gamma(x)$, and for other actions $\alpha\gamma$ defined as for the respective programs.

(5) If $(p, G, v) \xrightarrow{\alpha} (p', G', v')$ and the formula $f$ is satisfied for the valuation $v$ then (**if** $f$ **then** $p$ **else** $q, G, v$) $\xrightarrow{\alpha} (p', G', v')$.

(6) If $(q, G, v) \xrightarrow{\alpha} (q', G', v')$ and the formula $f$ is not satisfied for the valuation $v$ then (**if** $f$ **then** $p$ **else** $q, G, v$) $\xrightarrow{\alpha} (q', G', v')$.

(7) If $(p, G, v) \xrightarrow{\alpha} (p', G', v')$ then $(p; q, G, v) \xrightarrow{\alpha} (p'; q, G', v')$.

(8) If $(p, G, v)$ is terminal and $(q, G, v) \xrightarrow{\alpha} (q', G', v')$ then $(p; q, G, v) \xrightarrow{\alpha} (q, G', v')$.

(9) If $(p, G, v) \xrightarrow{\alpha} (p', G', v')$ then $(p \parallel q, G, v) \xrightarrow{\alpha} (p' \parallel q, G', v')$.

(10) If $(q, G, v) \xrightarrow{\alpha} (q', G', v')$ then $(p \parallel q, G, v) \xrightarrow{\alpha} (p \parallel q', G', v')$.

(11) If $(p, G, v) \xrightarrow{\alpha} (p', G', v')$ then $(p + q, G, v) \xrightarrow{\alpha} (p', G', v')$.

(12) If $(q, G, v) \xrightarrow{\alpha} (q', G', v')$ then $(p + q, G, v) \xrightarrow{\alpha} (q', G', v')$

(13) $(\Sigma xp, G, v) \xrightarrow{\Sigma x} (p', G, v')$, where $p'$ is obtained from $p$ by substituting a suitable variable which is not in the domain of $v$ for each free occurrence of $x$ in $p$, and $v'$ is the extension of $v$ by this new variable, and by assigning the value $none_G$ to this variable.

(14) $(\mathbf{atom}\, p, G, v) \xrightarrow{\mathbf{atom}\, p} (\mathbf{nil}, G', v')$ for each program $p$ and $G, G', v, v'$ such that there exists a finite sequence $u = t_1 t_2 ... t_m$ of transitions $t_i = (c_i \xrightarrow{\alpha_i} c'_i)$ such that $c_1 = (p, G, v)$, $c'_i = c_{i+1}$ whenever $t_i$ is followed by $t_{i+1}$, the configuration $c'_m$ is terminal, and $c'_m = (p', G', v')$.

(15) $(\mathbf{atom}\, p, G, v) \xrightarrow{\mathbf{atom}\, p} (?, G', v')$ if the configuration $(p, G, v)$ is unproper and $G' = G$ and $v' = v$, or if there exists a finite sequence $u = t_1 t_2 ... t_m$ of transitions $t_i = (c_i \xrightarrow{\alpha_i} c'_i)$ such that $c_1 = (p, G, v)$, $c'_i = c_{i+1}$ whenever $t_i$ is followed by $t_{i+1}$, and there is no transition $t_{m+1} = (c_{m+1} \xrightarrow{\alpha_i} c'_{m+1})$ with $c_{m+1} = c'_m$, and if the configuration $c'_m$ is unproper and $c'_m = (?, G', v')$.

(16) $(\mathbf{atom}\, p, G, v) \xrightarrow{\mathbf{atom}\, p} (?, ?, ?)$ if $(p, G, v) = (?, ?, ?)$ or there exists an infinite sequence $u = t_1 t_2 ...$ of transitions $t_i = (c_i \xrightarrow{\alpha_i} c'_i)$ such that $c_1 = (p, G, v)$ and $c'_i = c_{i+1}$ whenever $t_i$ is followed by $t_{i+1}$.

(17)

$$(\varphi_k(y_{k1}, y_{k2}, ...)\ \mathbf{where}\ (\varphi_1 = \psi_1, ..., \varphi_n = \psi_n), G, v) \xrightarrow{\tau} (\psi'_k, G, v)$$

where $\psi'_k$ is obtained by substituting in $\psi_k$ the program expression

$$\varphi_i(y_{i1}, y_{i2}, ...)\ \mathbf{where}\ (\varphi_1 = \psi_1, ..., \varphi_n = \psi_n)\gamma$$

for each occurrence of $\varphi_i(y_{i1}, y_{i2}, ...)\gamma$ in $\psi_k$, where $\gamma$ is a substitution.

The existence of the smallest relation satisfying (1) - (17) follows from the facts that the family of relations satisfying (1) - (17) is closed with respect to intersections of its nonempty subfamilies and that it is nonempty since the product $conf \times actions \times conf$ satisfies (1) - (17).

The transition relation allows to define computations and resulting relations of programs.

A *computation* is defined as a sequence $u = t_1 t_2 ...$, finite or not, of transitions $t_i = (c_i \xrightarrow{\alpha_i} c'_i)$ such that $c'_i = c_{i+1}$ whenever $t_i$ is followed by $t_{i+1}$. Such a computation is written in the form

$$u = (c_1 \xrightarrow{\alpha_1} c_2 \xrightarrow{\alpha_2} ...).$$

and $c_1$ is written as $\partial_0(u)$ and called the *initial* configuration of $u$. If $u$ is finite and its last transition is $t_m = (c_m \xrightarrow{\alpha_m} c'_m)$ then $c'_m$ is written as $\partial_1(u)$ and called the *final* configuration of $u$. If this final configuration is terminal then we say that $u$ *terminates*. If such a

final configuration is not terminal and there is no transition from this configutation to any other configuration then we say that $u$ *aborts*. A computation which terminates or aborts or is infinite is said to be *complete*. Each configuration $c$ is also regarded to be a computation, and we define $\partial_0(c)$ and $\partial_1(c)$ as $c$.

The concatenation $uv$ of two computations $u$ and $v$ such that $\partial_1(u) = \partial_0(v)$ is a computation (for $u$ being a configuration we define $uv = v$ and for $v$ being a configuration we define $uv = u$). Given a computation $u$, a *segment* of $u$ is a computation $v$ such that $u = xvy$ for some $x$ and $y$. Given a computation $u$, a *prefix* of $u$ is a computation $v$ such that $u = vy$ for some $y$. Such a prefix $v$ may be *proper* (if $v \neq u$) or *unproper* (if $v = u$).

Note that from the definition of the transition relation it follows that no proper prefix of a computation is a terminating or aborting computation. It is also obvious that each infinite computation is determined uniquely by its finite prefixes.

Given a program $p$, by a *computation of* $p$ we mean each computation $u$ such that $\partial_0(u) = (p, G, v)$ for some $G$ and $v$. The pair $(G, v)$ is called the *data* of $u$. If $u$ terminates and $\partial_1(u) = (p', G', v')$ then the pair $(G', v')$ is called the *result* of $u$.

The *resulting relation* of a program $p$, written as $res(p)$, is defined as the relation which holds between the data and the results of terminating computations of $p$: $(G, v)\ res(p)\ (G', v')$ iff there exists a terminating computation $u$ of $p$ such that $\partial_0(u) = (p, G, v)$ and $\partial_1(u) = (p', G', v')$.

As far as infinite computations are concerned, it is usually ensured in implementation that only such complete computations can be realized which enjoy a fairness property. In the sequel we shall assume that each actual complete computation $u$ is fair in the sense that there is no transition which is permanently possible starting from a configuration $c$ of $u$ and does not occur among the transitions which follow $c$ in $u$.

# 5 Denotational semantics

The possible (not necessarily fair) computations of a program $p$ constitute a set $[|p|]$ which is prefix-closed in the sense that together with each computation $u$ it contains every prefix of $u$. This set is determined uniquely by the subset of its finite computations, written as $[p]$. The subset $[p]$ is prefix-closed and it contains all terminating computations of $p$. In particular, it contains all information which is needed in order to determine the resulting relation of $p$. Thus $[p]$ may be regarded as a *meaning* of $p$. Consequently, The correpondence $p \mapsto [p]$ between programs and sets of their finite computations may be regarded as a denotational semantics of the considered language of programs.

Formally, the denotational semantics $p \mapsto [p]$ is a correspondence between programs and prefix-closed sets of finite computations, called *behaviours*. In this section we show that this correspondence is compositional in the sense that the behaviour of a compound program can be obtained by combining the behaviours of the respective component programs with the aid of suitable operations on behaviours.

Operations on behaviours correspond to the considered constructors of programs. The respective definitions are as follows.

The *result of applying a substitution* $\gamma$ of variables for program parameters to a behaviour $B$, written as $B\gamma$, is defined as the set of computations of the form $u\gamma$, where $u \in B$ and $u\gamma$ is obtained from $u$ by replacing each program $p$ which occurs in a configuration

of $u$ by the result $p\gamma$ of applying $\gamma$ to $p$.

The *result of a conditional choice* between behaviours $B$ and $B'$ depending on a formula $f$, written as $if\ f\ then\ B\ else\ B'$, is defined as the set of computations $u$ such that either $f$ is satisfied for the initial configuration of $u$ and $u \in B$, or $f$ is not satisfied for the initial configuration of $u$ and $u \in B'$.

The *sequential composition* of behaviours $B$ and $B'$, written as $B\hat{;}B'$, is defined as the set of prefixes (including unproper ones) of computations of the form $u$ or $u'v$, where

$$u = ((p_1; q_1, G_1, v_1) \stackrel{\alpha_1}{\to} ... \stackrel{\alpha_m}{\to} (p_{m+1}; q_1, G_{m+1}, v_{m+1}))$$

for a computation

$$((p_1, G_1, v_1) \stackrel{\alpha_1}{\to} ... \stackrel{\alpha_m}{\to} (p_{m+1}, G_{m+1}, v_{m+1})) \in B$$

and

$$u' = ((p_1; q_1, G_1, v_1) \stackrel{\alpha_1}{\to} ... \stackrel{\alpha_m}{\to} (p_{m+1}; q_1, G_{m+1}, v_{m+1}))$$

for a terminating computation

$$((p_1, G_1, v_1) \stackrel{\alpha_1}{\to} ... \stackrel{\alpha_m}{\to} (p_{m+1}, G_{m+1}, v_{m+1})) \in B$$

and

$$v = ((p_{m+1}; q_1, G_{m+1}, v_{m+1}) \stackrel{\alpha_{m+1}}{\to} (q_2, G_{m+2}, v_{m+2}) \stackrel{\alpha_{m+2}}{\to} ... \stackrel{\alpha_n}{\to} (q_{n+1}, G_{n+1}, v_{n+1}))$$

for a computation

$$((q_1, G_{m+1}, v_{m+1}) \stackrel{\alpha_{m+1}}{\to} (q_2, G_{m+2}, v_{m+2}) \stackrel{\alpha_{m+2}}{\to} ... \stackrel{\alpha_n}{\to} (q_{n+1}, G_{n+1}, v_{n+1})) \in B'.$$

The *interleaving* of behaviours $B$ and $B'$, written as $B\hat{\|}B'$, is defined as the set of prefixes of computations which consist of alternating segments of the form

$$(r_i, G_i, v_i) \stackrel{\alpha_i}{\to} ... \stackrel{\alpha_j}{\to} (r_{j+1}, G_{j+1}, v_{j+1}),$$

where either $r_i = p_i \parallel q_i, ..., r_{j+1} = p_{j+1} \parallel q_{j+1}$ with $q_i = ... = q_{j+1}$ and

$$(p_i, G_i, v_i) \stackrel{\alpha_i}{\to} ... \stackrel{\alpha_j}{\to} (p_{j+1}, G_{j+1}, v_{j+1})$$

is a segment of a computation from $B$, or $r_i = p_i \parallel q_i, ..., r_{j+1} = p_{j+1} \parallel q_{j+1}$ with $p_i = ... = p_{j+1}$ and

$$(q_i, G_i, v_i) \stackrel{\alpha_i}{\to} ... \stackrel{\alpha_j}{\to} (q_{j+1}, G_{j+1}, v_{j+1})$$

is a segment of a computation from $B'$.

The *indeterministic sum* of behaviours $B$ and $B'$, written as $B\hat{+}B'$, is defined as the set of prefixes of computations of the form

$$u = ((p, G_1, v_1) \stackrel{\alpha_1}{\to} (p_2, G_2, v_2) \stackrel{\alpha_2}{\to} ... \stackrel{\alpha_m}{\to} (p_{m+1}, G_{m+1}, v_{m+1})),$$

where either $p = p_1 + q_1$ with

$$(p_1, G_1, v_1) \stackrel{\alpha_1}{\to} (p_2, G_2, v_2) \stackrel{\alpha_2}{\to} ... \stackrel{\alpha_m}{\to} (p_{m+1}, G_{m+1}, v_{m+1})$$

being a computation from one of the sets $B$, $B'$ and with $(q_1, G_1, v_1)$ being the initial configuration of a computation from the other of the sets $B$, $B'$, or $u$ is a computation from one of the sets $B$, $B'$ and the other of the sets $B$, $B'$ is empty.

The *result of binding a parameter* $x$ in a behaviour B, written as $\Sigma x B$, is defined as the set of prefixes of computations of the form

$$(\Sigma x p, G_1, v) \stackrel{\Sigma x}{\rightarrow} (p'_1, G_1, v_1) \stackrel{\alpha'_1}{\rightarrow} ... \stackrel{\alpha'_m}{\rightarrow} (p'_{m+1}, G_{m+1}, v_{m+1}),$$

where

$$(p_1, G_1, v_1) \stackrel{\alpha_1}{\rightarrow} ... \stackrel{\alpha_m}{\rightarrow} (p_{m+1}, G_{m+1}, v_{m+1})$$

is a computation from $B$, $p'_1, ..., p'_{m+1}$ and $\alpha'_1, ..., \alpha'_m$ are obtained from $p_1, ..., p_{m+1}$ and $\alpha_1, ..., \alpha_m$ by substituting a variable $x'$ which is not in the domain of $v$ for each free occurrence of $x$, $v'_1$ is obtained from $v$ by extending the domain of $v$ by $x'$ and by assigning the value $none_{G_1}$ to $x'$, and $v'_1, ..., v'_{m+1}$ are obtained from $v_1, ..., v_{m+1}$ by replacing each pair $(x, v_i(x))$ by $(x', v_i(x))$.

The *result of atomizing* a behaviour $B$, written as *atom* $B$, is defined as the set of prefixes of computations of the form

$$(\mathbf{atom}\, p, G, v) \stackrel{\mathbf{atom}\, p}{\rightarrow} (p', G', v')$$

such that there exists a finite complete computation $u \in B$ with $\partial_0(u) = (p, G, v)$ and $\partial_1(u) = (p', G', v')$, or $(p', G', v') = (?, ?, ?)$ and there is an infinite computation $u \in B$ with $\partial_0(u) = (p, G, v)$.

All these operations, called *basic* ones, allow to define more operations. This is due to the chain completeness of the product orders induced by the inclusion order of behaviours and due to the fact that the basic operations preserve the least upper bounds of countable chains.

Namely, taking into account the well known results about fixed-points we can formulate the following definition.

The *least solution of a system of fixed-point equations*

$$B_1 = F_1(B_1, ..., B_m)$$
$$\vdots$$
$$B_m = F_m(B_1, ..., B_m)$$

where $F_1(B_1, ..., B_m)$,...,$F_m(B_1, ..., B_m)$ are expressions built from constants denoting concrete behaviours and from symbols $B_1$,...,$B_m$ of behaviours with the aid of basic operations on behaviours, is defined by the formula

$$B_i = \bigcup(B_i^k : k = 0, 1, ...),$$

where

$$B_1^0 = ... = B_m^0 = \emptyset$$

and

$$B_i^{k+1} = F_i(B_1^k, ..., B_m^k).$$

It is known that $B_i$ thus obtained constitute indeed a solution of the considered system of equations and that they are the least behaviours enjoying this property with respect to the inclusion order.

With the basic operations on behaviours and the possibility of solving systems of fixed-point equations we are able to define the meanings of programs in a compositional way.

**5.1. Theorem.** The meaning of each program can be defined as follows by induction on the syntactic construction of programs, starting from single actions:

(1) $[\mathbf{nil}]$ is the set of proper configurations of the form $(\mathbf{nil}, G, v)$.

(2) If $p$ is a parametrized production then $[p]$ is the set consisting of prefixes of computations of the form

$$(p, G, v) \xrightarrow{p} (\mathbf{nil}, H, w),$$

where $G, H, v, w$ are such that $p$ transforms $(G, v)$ into $(H, w)$ in the sense defined in section 2, and of prefixes of computations of the form

$$(p, G, v) \xrightarrow{p} (?, G, v),$$

where $p, G, v$ are such that the respective $H, w$ do not exist.

(3) If $p$ is a program of the form **set** $x, y, ...$**such that** $f$ then $[p]$ is the set of prefixes of computations of the form

$$(p, G, v) \xrightarrow{p} (\mathbf{nil}, G, w),$$

where $w$ is obtained from $v$ by modifying $v(x), v(y), ...$ such that the formula $f$ is satisfied for the new valuation $w$, or by replacing $v(x), v(y), ...$ by $none_G$, if such a modification is impossible.

(4) $[p\ \gamma] = [p]\gamma$.

(5) $[\mathbf{if}\ f\ \mathbf{then}\ p\ \mathbf{else}\ q] = if\ f\ then\ [p]\ else\ [q]$.

(6) $[p; q] = [p]\hat{;}[q]$.

(7) $[p \parallel q] = [p]\hat{\parallel}[q]$.

(8) $[p + q] = [p]\hat{+}[q]$.

(9) $[\Sigma x\ p] = \Sigma x[p]$.

(10) $[\mathbf{atom}\ p] = atom\ [p]$.

(11) If $p$ is a program of the form

$$\varphi_k(y_{k1}, y_{k2}, ...) \textbf{ where } (\varphi_1(y_{11}, y_{12}, ...) = \psi_1, ..., \varphi_n(y_{n1}, y_{n2}, ...) = \psi_n),$$

then $[p]$ is the $k$-th component of the least solution of the following system of fixed-point equations

$$B_j = \tau_j \hat{;} [\psi'_j],$$

where $j \in \{1, ..., n\}$, $\tau_j$ is the set of prefixes of computations of the form

$$(\varphi_j(y_{j1}, y_{j2}, ...) \textbf{ where } (\varphi_1 = \psi_1, ..., \varphi_n = \psi_n), G, v) \xrightarrow{\tau} (\textbf{nil}, G, v),$$

and $[\psi'_j]$ is the expression obtained from $\psi'_j$ by replacing each program of the form $\varphi_i\gamma$ by $B_i\gamma$, each explicitly defined program $q$ by $[q]$, and each symbol of operation on programs by the symbol of the corresponding operation on behaviours. □

Proof outline: The characterizations of the meanings of **nil**, parametrized productions, and program expressions of the form **set** $x, y, ...$**such that** $f$, follow directly from the definition of the transition relation (it suffices to take into account points (1) - (3) of this definition and the minimality of the transition relation).

The formulas in (4) - (10) follow directly from the definition of the transition relation and from the definitions of operations on behaviours.

For (11) it suffices to show that the behaviours of the programs $\varphi_j$ **where** $(\varphi_1 = \psi_1, ..., \varphi_n = \psi_n)$ satisfy the fixed-point equations in (11) and that they are the least behaviours with this property.

Let $B'_j$ denote the behaviour of the program $\varphi_j$ **where** $(\varphi_1 = \psi_1, ..., \varphi_n = \psi_n)$.

Suppose that $u \in B'_j$ and $u \neq \partial_0(u)$. Then $u = tu'$, where $t$ is a transition from $\tau_j$ and $u'$ is a computation of $\psi'_j$. Hence $u \in \tau_j \hat{;} [\psi'_j]$. Thus $B'_j \subseteq \tau_j \hat{;} [\psi'_j]$. Conversely, each computation of the form $tu'$, where $t$ is a transition from $\tau_j$ and $u'$ is a computation of $\psi'_j$, belongs to $B'_j$. Thus $\tau_j \hat{;} [\psi'_j] \subseteq B'_j$. Consequently, $B'_j = \tau_j \hat{;} [\psi'_j]$, that is $B'_j$ satisfies the respective fixed-point equation.

Let $B_1, ..., B_n$ constitute the least solution of the set of fixed-point equations in (11). It remains to prove that $B_1 = B'_1, ..., B_n = B'_n$.

We know that each $B_j$ is approximated by $B^0_j, B^1_j, ...$, where $B^0_j = \emptyset$ and

$$B^{k+1}_j = \tau_j \hat{;} [\psi'_j(B^k_1, ..., B^k_n)].$$

Thus it suffices to prove that each $u \in B'_j$ belongs to some $B^k_j$.

Suppose that $u \in B'_j$. Then $u$ can be represented in the form $t_1u_1...t_ku_k$, where each $t_i$ belongs to some $\tau_{j(i)}$, where $j(1) = j$, and $u_1, ..., u_k$ consist of transitions which do not belong to any $\tau_m$. This implies that $t_ku_k$ corresponds to a computation in $B^1_{j(k)}$, $t_{k-1}u_{k-1}t_ku_k$ corresponds to a computation in $B^2_{j(k-1)}$,..., $u$ corresponds to a computation in $B^k_{j(1)} = B^k_j$, as required. □

# 6 Resulting relations of programs

The possibility of using computations of programs to define the respective resulting relations suggests that an input-output semantics could be defined directly in terms of

resulting relations. Unfortunately, such a definition is impossible because of the lack of compositionality of the correspondence between programs and their resulting relations. This is caused by the presence of the parallel composition since the resulting relation of a program obtained by composing given programs depends not only on the resulting relations of these programs, but also on the resulting relations of smaller program components. For example, for a program $p \parallel q$, where $p = p_1; p_2$, $q = q_1; q_2$, and $p_1$, $p_2$, $q_1$, $q_2$ are parametrized productions the resulting relation is

$$res(p \parallel q) = res(p_1; p_2; q_1; q_2) \cup res(p_1; q_1; p_2; q_2) \cup res(p_1; q_1; q_2; p_2)$$

$$\cup res(q_1; p_1; p_2; q_2) \cup res(q_1; p_1; q_2; p_2) \cup res(q_1; q_2; p_1; p_2)$$

and it cannot be expressed in terms of $res(p) = res(p_1; p_2)$ and $res(q) = res(q_1; q_2)$.

The lack of compositionality w. r. to the parallel composition shows that there is no chance for a direct input-output semantics of the considered programs. In particular, an operational or denotational semantics as presented cannot be avoided even if we are interested only in the resulting relations of programs. On the contrary, in the situation in which such relations cannot be derived from programs directly, a semantics of this kind becomes an important tool of defining the resulting relations.

The reasoning about resulting relations of programs can be supported by a number of properties of such relations.

**6.1. Proposition.** The following properties hold true for the resulting relations of programs:

(1) $res(\mathbf{nil}) = identity$.

(2) If $p$ is a parametrized production then $(G, v)res(p)(H, w)$ iff $p$ transforms $(G, v)$ into $(H, w)$ in the sense defined in section 2.

(3) $(G, v)res(p\gamma)(G', v')$ iff $(G, \gamma v)res(p)(G', \gamma v')$.

(4) $(G, v)res(\mathbf{if}\ f\ \mathbf{then}\ p\ \mathbf{else}\ q)(G', v')$ iff either $f$ is satisfied for $G$ and $v$ and $(G, v)res(p)(G', v')$ or $f$ is not satisfied and $(G, v)res(q)(G', v')$.

(5) $res(p; q) = res(p)res(q)$, the standard composition of $res(p)$ and $res(q)$.

(6) $res(p + q) = res(p) \cup res(q)$. $\square$

A proof is of this proposition follows from 5.1.

The class of resulting relations of programs is rich enough to represent the usual derivability relation.

Taking into account the definition of the resulting relation of a program and the definition of the semantics of programs, we obtain the following realization of the relation of derivability.

**6.2. Proposition.** Let $\Pi = \{p_1, ..., p_m\}$ be a finite set of parametrized productions. There is a program $p$ such that $H$ is derivable from $G$ via productions from $\Pi$ iff $(G, v)res(p)(H, v')$ for some $v$ and $v'$. In particular, we may define such a program as

$$p = X \textbf{ where } (X = q; X + \textbf{nil})$$

where

$$q = (\Sigma x_1)...(\Sigma x_m)(p_1 + ... + p_m)$$

and $\{x_1, ..., x_m\}$ is the set of parameters of productions $p_1, ..., p_m$. $\square$

# 7 Recapitulation

We have presented conceptual means for programming concurrent processes of rewriting graphs by applying productions. These means are flexible enough to cover the usual rewriting. However, their possibilities go much beyond such particular cases due to the powerful mechanisms of parameters, recursion, concurrency, and operating on colours.

The presented formalism is brought to the form of (a kernel of) a programming language with a precise syntax and semantics.

The semantics defines computations of each program $p$. These computations represent the possible ways of transforming a given graph $G$ and a given valuation of variables in this graph, and thus they determine a resulting relation $res(p)$ of $p$. The possibility of determining such a relation is important since there is no way of defining it directly from the resulting relations of program components.

The semantics presented in the paper represents concurrency as an arbitrary interleaving of actions. Nevertheless, it contains implicitly all the information about the existing concurrency.

**Acknowledgements.** The authors are grateful to the anonymous referees of early versions of this paper for their comments and suggestions.

# References

[B 79] Bunke, H., *Programmed Graph Grammars*, in [CER 79], pp.155-166

[CER 79] Claus, V., Ehrig, H., Rozenberg, G., (Eds.) *Proceedings of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, Springer LNCS 73, 1979.

[CMREL 91] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Löwe, M., *Logic Programming and Graph Grammars*, in [EKR 91], pp.221-237.

[CR 93] Corradini, A., Rossi, F., *Hyperedge Replacement Jungle Rewriting for Term-Rewriting Systems and Logic Programming*, Theoretical Computer Science 109 (1993) pp.7-48.

[CRP 91] Corradini, A., Rossi, F., Parisi-Presicce, F., *Logic Programming as Hypergraph Rewriting*, in the Proceedings of CAAP'91, Springer LNCS 493, 1991. pp.275-295.

[EKMRW 82] Ehrig, H., Kreowski, H.-J., Maggiolo-Schettini, A., Rosen, B.K., Winkowski, J., *Transformations of Structures: An Algebraic Approach*, Math. Systems Theory 14 (1981) pp.305-334.

[EKR 91] Ehrig, H., Kreowski, H.-J., Rozenberg, G., (Eds.) *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer LNCS 532, 1991.

[ENR 83] Ehrig, H., Nagl, M., Rozenberg, G., (Eds.) *Proceedings of the 2nd Workshop on Graph-Grammars and Their Application to Computer Science*, Springer LNCS 153, 1983.

[ENRR 87] Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A., (Eds.) *Proceedings of the 3rd Workshop on Graph-Grammars and Their Application to Computer Science*, Springer LNCS 291, 1987.

[EPS 73] Ehrig, H., Pfender, H., Schneider, H. J., *Graph Grammars: An Algebraic Approach*, Proc. of the IEEE Conf. on Automata and Switching Theory, Iowa City 1973, pp.167-180.

[JaRo 91] Janssens, D., Rozenberg, G., *Structured Transformations and Computation Graphs for Actor Grammars*, in [EKR 91], pp.446-460.

[MW 83] Maggiolo-Schettini, A., Winkowski, J., *Towards a Programming Language for Manipulating Relational Data Bases*, in: Bjorner, D., (Ed.), Formal Description of Programming Concepts II, North-Holland, 1983, pp.265-278.

[MW 91] Maggiolo-Schettini, A., Winkowski, J., *Programmed Derivations of Relational Structures*, in [EKR 91], pp.582-598.

[MW 92] Maggiolo-Schettini, A., Winkowski, J., *A Programming Language for Deriving Hypergraphs*, in Springer LNCS 581, 1992, pp.221-231.

[MW 94] Maggiolo-Schettini, A., Winkowski, J., *A Formalism for Programmed Rewriting of Hypergraphs*, Report 745 of the Institute of Computer Science, Warsaw, 1994.

[Na 79] Nagl, M., *A Programming Language for Handling Dynamic Problems on Graphs*, in Pape, U. (ed.): Discrete Structures and Algorithms, Proc. of WG'79 5th Workshop on Graphtheoretic Concepts in Computer Science, Hanser Verlag, 1979.

[Plo 81] Plotkin, G., *A Structural Approach to Operational Semantics*, Technical Report, Computer Sc. Dept., Aarhus Univ., Denmark, DAIMI-FN-19, 1981.

[P 91] Plump, D., *Graph-Reducible Term Rewriting Systems*, in [EKR 91], pp.622-636.

[S 91] Schürr, A., *PROGRESS: A VHL-Language Based on Graph Grammars*, Springer LNCS 532, pp.641-659, 1991

[Sh 89] Shapiro, E., *The Family of Concurrent Logic Programming Languages*, ACM Computing Surveys, 21, 1989, pp.413-510.

[ZS 92] Zündorf, A., Schürr, A., *Nondeterministic Control Structures for Graph Rewriting Systems*, Springer LNCS 570, pp.48-62, 1992.