Computational Completeness of Programming Languages Based on Graph Transformation

Annegret Habel[1] and Detlef Plump[2]

[1] Fachbereich Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg, Germany
`habel@informatik.uni-oldenburg.de`

[2] Department of Computer Science, The University of York
Heslington, York YO10 5DD, United Kingdom
`det@cs.york.ac.uk`

Abstract. We identify a set of programming constructs ensuring that a programming language based on graph transformation is computationally complete. These constructs are (1) nondeterministic application of a set of graph transformation rules, (2) sequential composition and (3) iteration. This language is minimal in that omitting either sequential composition or iteration results in a computationally incomplete language. By computational completeness we refer to the ability to compute every computable partial function on labelled graphs. Our completeness proof is based on graph transformation programs which encode arbitrary graphs as strings, simulate Turing machines on these strings, and decode the resulting strings back into graphs.

1 Introduction

The use of graphs to represent and visualise complex structures is ubiquitous in computer science, and often these structures occur in contexts where they have to be dynamically changed. Functional and logic programming languages, on the other hand, are successful examples of high-level programming languages based on rules. Thus a natural idea is to design programming languages based on graph transformation rules, in order to combine the strengths of graphs and rule-based programming.

Existing programming languages of this type include PROGRES [SWZ99], AGG [ERT99], GAMMA [FM98], GRRR [Rod98] and DACTL [GKS91]. These languages have in common that they are based on graph transformation rules, but they vary strongly with respect to both the formalisms underlying the rules and the available constructs for controlling rule applications. In view of the variety of control mechanisms, the question arises what programming constructs are really needed on top of graph transformation rules to obtain a computationally complete language. By computational completeness we mean the ability to compute every computable partial function on labelled graphs. Identifying such a kernel language for graph transformation will benefit to both the understanding of existing languages and the design of new programming languages of this kind.

In this paper we show that three programming constructs suffice to guarantee computational completeness: (1) nondeterministic application of a rule from a set of graph transformation rules (according to the so-called double-pushout approach), (2) sequential composition and (3) iteration in the form that rules are applied as long as possible. This language is not only complete but also minimal in that omitting either sequential composition or iteration makes the language computationally incomplete.

One may wonder why plain sets of graph transformation rules with the semantics "apply as long as possible" are not computationally complete. Indeed it is not difficult to simulate Turing machines by sets of graph transformation rules (in the double-pushout approach), but this only means that all computations on *representations* of graphs can be modelled. The ability to transform a string representation of a graph G into a string representation of the graph $f(G)$, where f is some graph function, does not imply that there is a set of rules transforming G directly into $f(G)$.

So what is different to the case of string rewriting, where sets of rules do suffice to compute all computable functions on strings? (See, for example, Lewis' and Papadimitriou's concept of a grammatically computable function [LP98].) The point is that in a string-based model, prior to computations input strings are provided with some context of auxiliary symbols which must not occur in inputs but which can be used in the rules. This context allows to control the application of rules, ensuring that computations have universal power. It is open whether there is a similar concept that makes sets of graph transformation rules universally powerful. The problem is that in contrast to strings, arbitrary graphs do not possess distinguished points for attaching context.

For the programming language introduced below we do not assume that input graphs come in any particular format, the idea is rather to provide just enough control constructs to ensure computational completeness. Our completeness proof is based on the sequential composition of three programs: the first encodes arbitrary graphs as certain strings, the second simulates Turing machines on these strings, and the third decodes the resulting strings back into graphs. The strings for representing graphs are similar to those of Uesu [Ues78] who showed that graph grammars according to the double-pushout approach can generate all recursively enumerable sets of labelled graphs.

Finally we show that our programming language is minimal, by proving that the function converse which swaps sources and targets of all edges in a graph cannot be computed if either sequential composition or iteration is missing.

2   Rules

This section recalls the application of graph transformation rules according to the "double-pushout" approach. Details and pointers to the literature can be found in [HMP99].

A *label alphabet* $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ is a pair of finite sets of *node labels* and *edge labels*. A *graph* over $\mathcal{C}$ is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_G, \mathrm{m}_G)$ consisting of

two finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, two *source* and *target functions* $s_G, t_G \colon E_G \to V_G$, and two *labelling functions* $l_G \colon V_G \to \mathcal{C}_V$ and $m_G \colon E_G \to \mathcal{C}_E$.

A *graph morphism* $g \colon G \to H$ between two graphs G and H consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources, targets, and labels, that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $l_H \circ g_V = l_G$, and $m_H \circ g_E = m_G$. A morphism g is *injective (surjective)* if g_V and g_E are injective (surjective), and an *isomorphism* if it is both injective and surjective. In the latter case G and H are *isomorphic*, which is denoted by $G \cong H$. A morphism g is an *inclusion* if $g_V(v) = v$ and $g_E(e) = e$ for all $v \in V_G$ and $e \in E_G$.

A *rule* $r = \langle L \leftarrow K \to R \rangle$ consists of two graph morphisms with a common domain K, which is the *interface* of r. We throughout assume that $K \to L$ is an inclusion. The *application* of r to a graph G amounts to the following steps:

(1) Find an injective graph morphism $g \colon L \to G$ satisfying the *dangling condition*: No edge in $G - g(L)$ is incident to a node in $g(L) - g(K)$.
(2) Remove $g(L) - g(K)$ from G, yielding a graph D, a graph morphism $K \to D$ which is the restriction of g, and the inclusion $D \to G$.
(3) Construct the pushout of $K \to D$ and $K \to R$, yielding a graph H and graph morphisms $D \to H$ and $R \to H$. (See [Ehr79] or the appendix of [HMP99] for the construction of graph pushouts.)

This construction yields the pushout diagrams (1) and (2) in Figure 1. Roughly, H is obtained from the intermediate graph D by merging items according to the morphism $K \to R$ (in case this morphism is not injective) and adding the items of R that are not in the image of K.

The transformation of G into H is denoted by $G \Rightarrow_{r,g} H$. We write $G \Rightarrow_r H$ to express that there is a graph morphism g such that $G \Rightarrow_{r,g} H$. Given a set $\mathcal{R}$ of rules, $G \Rightarrow_{\mathcal{R}} H$ means that there is a rule r in $\mathcal{R}$ such that $G \Rightarrow_r H$. So the relation $\Rightarrow_{\mathcal{R}}$ is nondeterministic with respect to both the rule chosen from $\mathcal{R}$ and the position in the given graph where this rule is applied.
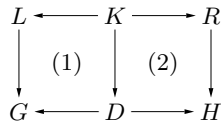
$$
\begin{array}{ccccc}
L & \longleftarrow & K & \longrightarrow & R \\
\downarrow & (1) & \downarrow & (2) & \downarrow \\
G & \longleftarrow & D & \longrightarrow & H
\end{array}
$$

Fig. 1. A transformation step in form of a double-pushout

We will use graph transformation to compute relations on abstract graphs rather than on concrete graphs as above, so we identify isomorphic graphs and lift transformation steps to isomorphism classes of graphs. An *abstract graph* over a label alphabet $\mathcal{C}$ is an isomorphism class of graphs over $\mathcal{C}$. We write $[G]$ for the isomorphism class of a graph G and denote by $\mathcal{A}_{\mathcal{C}}$ the set of all abstract graphs

over $\mathcal{C}$. The relation $\Rightarrow_{\mathcal{R}}$ is lifted to $\mathcal{A}_{\mathcal{C}}$ by: $[G] \Rightarrow_{\mathcal{R}} [H]$ if $G \Rightarrow_{\mathcal{R}} H$. This yields a well-defined relation since, by the definition of transformation steps as double-pushouts, we have for all graphs G, G', H and H' over $\mathcal{C}$: $G' \cong G \Rightarrow_{\mathcal{R}} H \cong H'$ implies $G' \Rightarrow_{\mathcal{R}} H'$.

3   Programs

The programs we are going to define are based on sets of graph transformation rules. In this paper we do not address the issue how to represent rules syntactically, we rather assume that sets of rules, single rules and graphs have names to which programs can refer.

Definition 1 (Program). *Programs* over a label alphabet $\mathcal{C}$ are inductively defined as follows:

(1) Every finite set $\mathcal{R}$ of rules over $\mathcal{C}$ is a program.
(2) If P_1 and P_2 are programs, then $\langle P_1; P_2 \rangle$ is a program.
(3) If P is a program according to (1) or (2), then $P{\downarrow}$ is a program.

Programs according to (1) are *elementary*, the program $\langle P_1; P_2 \rangle$ is the *sequential composition* of P_1 and P_2, and $P{\downarrow}$ is the *iteration* of P. Programs of the form $\langle P_1; \langle P_2; P_3 \rangle \rangle$ and $\langle \langle P_1; P_2 \rangle; P_3 \rangle$ are considered as equal; by convention, both can be written as $\langle P_1; P_2; P_3 \rangle$.

Next we provide programs with a relational input/output semantics. Given a binary relation $\to$ on a set S, we denote by $\to^+$ the transitive closure of $\to$ and by $\to^*$ the reflexive-transitive closure. An element a in S is a *normal form* with respect to $\to$ if there is no b in S such that $a \to b$.

Definition 2 (Semantics). Given a program P over a label alphabet $\mathcal{C}$, the *semantics* of P is a binary relation $\to_P$ on $\mathcal{A}_{\mathcal{C}}$ which is inductively defined as follows:

(1) $\to_P = \Rightarrow_{\mathcal{R}}$ if P is an elementary program $\mathcal{R}$.
(2) $\to_{\langle P_1; P_2 \rangle} = \to_{P_1} \circ \to_{P_2}$.
(3) $\to_{P{\downarrow}} = \{\langle G, H \rangle \mid G \to_P^* H$ and H is a normal form with respect to $\to_P\}$.

Consider now subalphabets $\mathcal{C}_1$ and $\mathcal{C}_2$ of $\mathcal{C}$ and a relation $Rel \subseteq \mathcal{A}_{\mathcal{C}_1} \times \mathcal{A}_{\mathcal{C}_2}$. We say that P *computes* Rel if $Rel = \to_P \cap (\mathcal{A}_{\mathcal{C}_1} \times \mathcal{A}_{\mathcal{C}_2})$, that is, if Rel coincides with the semantics of P restricted to $\mathcal{A}_{\mathcal{C}_1}$ and $\mathcal{A}_{\mathcal{C}_2}$. The same applies to partial functions $f: \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$, which are just special relations.

Example 1 (Functions computed by programs).

1. Given a graph K in $\mathcal{A}_{\mathcal{C}}$, the constant function $\mathrm{const}_K: \mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}}$ with $\mathrm{const}_K(G) = K$ for all $G \in \mathcal{A}_{\mathcal{C}}$ is computed by the program

$$\mathtt{Const}_K = \langle \mathtt{Delete}{\downarrow}; \mathtt{Add}_K \rangle,$$

   where $\mathtt{Delete}$ is an elementary program deleting nodes and edges (with arbitrary labels in $\mathcal{C}$), and $\mathtt{Add}_K$ is the elementary program consisting of the single rule $\langle \emptyset \leftarrow \emptyset \to K \rangle$.

2. The function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ swaps source and target of each edge in a graph. It is computed by the program

$$\texttt{Converse} = \langle \texttt{Redirect}\downarrow;\ \texttt{Relabel}\downarrow \rangle$$

over the label alphabet $\langle \mathcal{C}_V, \mathcal{C}_E \cup (\mathcal{C}_E \times \{'\}) \rangle$. The rules of $\texttt{Converse}$ are shown in Figure 2.[1] Note that the redirected edges temporarily get auxiliary labels to prevent further redirection. After termination of $\texttt{Redirect}\downarrow$ all edges get their original labels by the subprogram $\texttt{Relabel}\downarrow$.
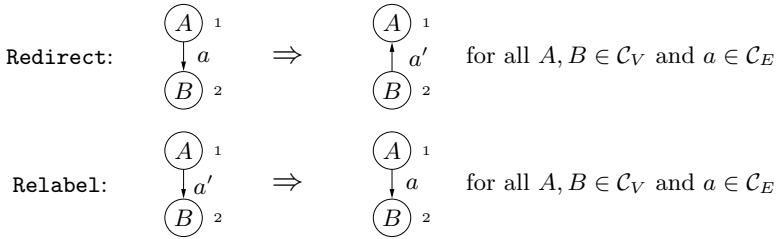
Fig. 2. The rules of the program $\texttt{Converse}$

In the proof of our completeness result in Section 6, we will use a program scheme $\texttt{Ite}(K, P_1, P_2)$ which checks whether the input graph equals K and executes P_1 or P_2 depending on whether the check is successful or not. More precisely, the semantics is given by $G \to_{\texttt{Ite}(K,P_1,P_2)} H$ if and only if $G = K$ and $G \to_{P_1} H$ or $G \neq K$ and $G \to_{P_2} H$. The scheme is defined by

$$\texttt{Ite}(K, P_1, P_2) = \langle \texttt{Check}(K);\ \langle \texttt{Delete}_1;\ P_1 \rangle\downarrow;\ \langle \texttt{Delete}_2;\ P_2 \rangle\downarrow \rangle,$$

where $\texttt{Check}(K)$ copies the input graph G and reduces the copy to a node with label 1 if $G = K$, and to a node with label 2 otherwise. For $i = 1, 2$, $\texttt{Delete}_i$ deletes a node with label i. If $\texttt{Check}(K)$ yields 1, then $\langle \texttt{Delete}_1;\ P_1 \rangle$ can be executed only once because the node with label 1 is deleted and $\langle \texttt{Delete}_2;\ P_2 \rangle$ is executed zero times because there is no node with label 2. Vice versa, if $\texttt{Check}(K)$ yields a node with label 2, then $\langle \texttt{Delete}_1;\ P_1 \rangle$ is executed zero times and $\langle \texttt{Delete}_2;\ P_2 \rangle$ is executed once. We omit the rules of this program scheme for space reasons.

4 Computable Graph Functions

In this section we introduce the notion of a computable partial function on abstract graphs, by using Turing computability on strings and an encoding of abstract graphs as strings. This is consistent with Weihrauch's concept of *(strong) relative computability* [Wei87].

---

[1] In order to present rules concisely, we show only the left- and right-hand sides. The interfaces consist of the numbered nodes of the left-hand sides and have no edges.

We start by defining graph expressions as certain well-formed strings and a surjective partial function gra from strings to abstract graphs which assigns to every graph expression an abstract graph. To this end, let $\mathcal{C}$ be a label alphabet and set $\Sigma = \mathcal{C}_V \cup \mathcal{C}_E \cup \{1, 2, \#\}$. We assume $\mathcal{C}_V \cap \mathcal{C}_E = \emptyset$ and that 1, 2 and $\#$ do not occur in $\mathcal{C}_V$ and $\mathcal{C}_E$.

Definition 3 (Graph expression). The set Exp of *graph expressions* over Σ and the graph $w^\square$ represented by a graph expression w are inductively defined as follows:

(1) The empty string λ is in Exp and $\lambda^\square = \emptyset$.
(2) For all $A \in \mathcal{C}_V$, $\#A1\# \in$ Exp and $\#A1\#^\square$ is the graph consisting of a single node 1 with label A.
(3) If $v\#w \in$ Exp and $A \in \mathcal{C}_V$, then $v\#A1^n\#w \in$ Exp with $n = |V_{v\#w^\square}| + 1$ and $v\#A1^n\#w^\square$ is obtained from $v\#w^\square$ by adding a node n with label A.
(4) If $v\#w \in$ Exp, $F \in \mathcal{C}_E$ and $v\#w$ contains substrings $A1^m\#$ and $B1^n\#$, then $v\#A2^m F2^n B\#w \in$ Exp and $v\#A2^m F2^n B\#w^\square$ is obtained from $v\#w^\square$ by adding an edge $|E_{v\#w^\square}| + 1$ which has label F, source node m and target node n.

A substring $\#A1^n\#$ in a graph expression represents a node with name n and label A, while a substring $\#A2^m F2^n B\#$ stands for an edge with label F, source node m and target node n. Note that the order of nodes and edges in a graph expression is arbitrary and that a graph $w^\square$ has the node set $\{1, 2, \ldots, |V_{w^\square}|\}$.

Definition 4 (Representation of abstract graphs). The partial function gra: $\Sigma^* \to \mathcal{A}_\mathcal{C}$ is defined as follows:

$$\text{gra}(w) = \left\{ \begin{array}{ll} [w^\square] & \text{if } w \text{ is a graph expression,} \\ \text{undefined} & \text{otherwise.} \end{array} \right.$$

The function gra is surjective since every isomorphism class of graphs contains a graph represented by a graph expression.

Let now $\mathcal{C}_1$ and $\mathcal{C}_2$ be any subalphabets of $\mathcal{C}$ and define for $i = 1, 2$, $\Sigma_i = \mathcal{C}_{iV} \cup \mathcal{C}_{iE} \cup \{1, 2, \#\}$.

Definition 5 (Computable graph function). A partial function $f: \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$ on abstract graphs is *computable* if there exists a computable partial function $f': \Sigma_1^* \to \Sigma_2^*$ on strings such that for every graph expression w,

$$f(\text{gra}(w)) = \text{gra}(f'(w))$$

and $\text{gra}(w) \notin \text{Dom}(f)$ implies $w \notin \text{Dom}(f')$.[2]

Thus f is computable if there is a computable function f' on strings such that for every abstract graph G for which f is defined and every graph expression w denoting G, f' is defined for w and yields a graph expression denoting $f(G)$. This situation is illustrated in Figure 3. Moreover, f' is not defined on graph expressions denoting graphs on which f is not defined.

---

[2] We denote by $\text{Dom}(f)$ the domain of a partial function f.

$$\begin{array}{ccc}
\mathcal{A}_{\mathcal{C}_1} & \xrightarrow{\ f\ } & \mathcal{A}_{\mathcal{C}_2} \\
{\scriptstyle\text{gra}}\Big\uparrow & & \Big\uparrow{\scriptstyle\text{gra}} \\
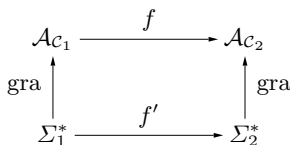\Sigma_1^* & \xrightarrow{\ f'\ } & \Sigma_2^*
\end{array}$$

Fig. 3. Computability of a graph function f

5   Simulation of Turing Machines

In the next section we will show that every computable graph function can be computed by a program. An essential argument will be that every computable partial function on strings can be computed by a graph transformation program working on so-called *string graphs*. The string graph of a string $w = a_1 a_2 \ldots a_n$ is the abstract graph shown in Figure 4 and is denoted by $w^\bullet$. (It is understood that $w^\bullet$ consists of a single node if w is the empty string.)
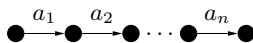

Fig. 4. The string graph for $a_1 a_2 \ldots a_n$

Theorem 1. *For every computable partial function* $f\colon \Sigma_1^* \to \Sigma_2^*$ *there is a program* P_f *over a label alphabet* $\mathcal{C}$ *with* $\mathcal{C}_1, \mathcal{C}_2 \subseteq \mathcal{C}$ *such that for all* w *in* Σ_1^* *and* G *in* $\mathcal{A}_{\mathcal{C}}$,

$$w^\bullet \to_{P_f} G \text{ if and only if } G = f(w)^\bullet.$$

To prove Theorem 1 we will simulate Turing machines by programs, so we first recall the definition of Turing machines and their computed functions (using a version similar to that in [LP98]).

A *Turing machine* is a system $M = \langle Q, \Gamma, \delta, q_0, \square \rangle$ where Q is a finite set of states, Γ is a finite set of tape symbols, $\square \in \Gamma$ is the blank symbol, δ is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{r, n, l\}$ called the transition function and $q_0 \in Q$ is the start state.

A *configuration* of M is a string $c = uqav$ such that $q \in Q$, $a \in \Gamma$ and $u, v \in \Gamma^*$. The start configuration of M with respect to a string w in Σ^* is given by $\alpha(w) = q_0 w$ if w is not empty and by $q_0 \square$ otherwise. Given a configuration $c = uqav$, we write $c \vdash_M c'$ and call c' the successor configuration of c if c' is given by

$$c' = \begin{cases} ua'q'v & \text{if } \delta(q,a) = (q',a',r) \text{ and } v \neq \lambda, \\ ua'q'\square & \text{if } \delta(q,a) = (q',a',r) \text{ and } v = \lambda, \\ uq'a'v & \text{if } \delta(q,a) = (q',a',n), \\ u'q'ba'v & \text{if } \delta(q,a) = (q',a',l) \text{ and } u = u'b, \\ q'\square a'v & \text{if } \delta(q,a) = (q',a',l) \text{ and } u = \lambda. \end{cases}$$

A configuration is final if it has no successor configuration. The *result* of a configuration $c = uqv$ is the string $\omega(uqv) = \bar{u}\bar{v}$ where $\bar{u}$ is the shortest string with $u = \square\ldots\square\bar{u}$ and $\bar{v}$ is the shortest string with $v = \bar{v}\square\ldots\square$.

Let Σ_1 and Σ_2 be subsets of $\Gamma - \{\square\}$. The partial function $f_M \colon \Sigma_1^* \to \Sigma_2^*$ *computed* by M is given by $f_M(v) = w$ if there is a final configuration c such that $\alpha(v) \vdash_M^* c$, $\omega(c) = w$ and $w \in \Sigma_2^*$, and undefined otherwise.

Proof of Theorem 1. For every computable partial function $f \colon \Sigma_1^* \to \Sigma_2^*$ there exists a Turing machine M that computes f. So we have to show that for every Turing machine M there exists a program $\texttt{Turing}(M)$ that transforms string graphs into string graphs such that for all v in Σ_1^* and w in Σ_2^*, $f_M(v) = w$ if and only if $v^\bullet \to_{\texttt{Turing}(M)} w^\bullet$. To this end, let

$$\texttt{Turing}(M) = \langle\texttt{Initiate};\ \texttt{Simulate}{\downarrow};\ \texttt{Eliminate}{\downarrow};\ \texttt{Finish}\rangle$$

where $\texttt{Initiate}$ is an elementary program attaching a state node labelled with the start state to an input string, $\texttt{Simulate}\downarrow$ simulates the working of M, $\texttt{Eliminate}\downarrow$ removes all blanks from the final configuration, and $\texttt{Finish}$ deletes the state node. The rules of $\texttt{Turing}(M)$ are given in Figure 5. $\qquad\square$

6   Computational Completeness

We are now ready to state our main result, namely that every computable partial function on abstract graphs is computed by a program in the programming language defined in Section 3.

Theorem 2. *For every computable partial function* $f \colon \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$ *there exists a program that computes* f.

Proof. Let $f \colon \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$ be computable. Then, by Definition 5, there is a computable partial function $f' \colon \Sigma_1^* \to \Sigma_2^*$ such that for every graph expression w, $f(\text{gra}(w)) = \text{gra}(f'(w))$. Let $P_{f'}$ be the program of Theorem 1 which simulates f', and let $\texttt{Encode}$ and $\texttt{Decode}$ be the programs of Lemmata 1 and 2 below. Without loss of generality, we may assume that $\texttt{Encode}$, $P_{f'}$, and $\texttt{Decode}$ are programs over a common label alphabet $\mathcal{C}$. We show that for all G in $\mathcal{A}_{\mathcal{C}_1}$ and H in $\mathcal{A}_{\mathcal{C}_2}$,

$$G \to_{\langle\texttt{Encode};\ P_{f'};\ \texttt{Decode}\rangle} H \text{ if and only if } f(G) = H.$$

"Only if": Let $G \to_{\langle\texttt{Encode};\ P_{f'};\ \texttt{Decode}\rangle} H$. Then there are G_1 and G_2 in $\mathcal{A}_{\mathcal{C}}$ such that $G \to_{\texttt{Encode}} G_1 \to_{P_{f'}} G_2 \to_{\texttt{Decode}} H$. By Lemma 1 we have $G_1 = w^\bullet$ for
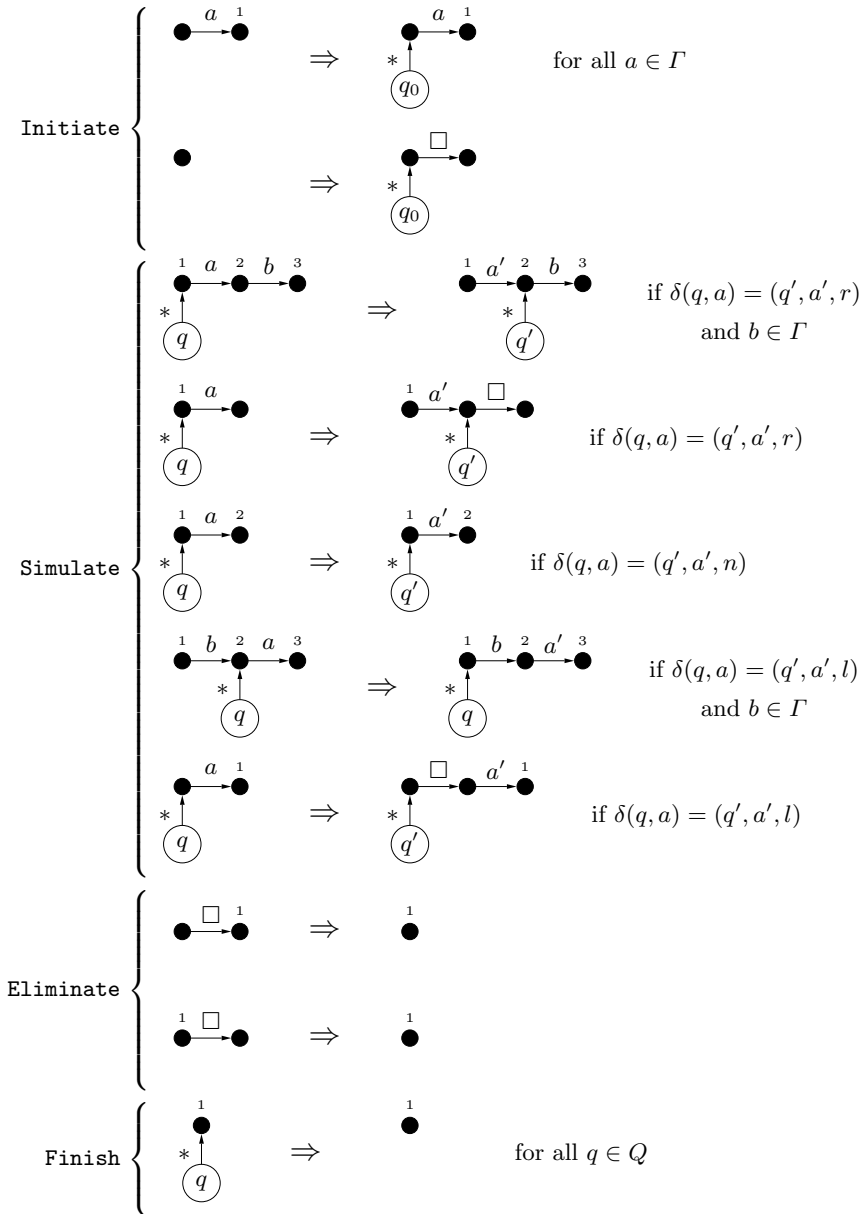
Fig. 5. The rules of the program $\mathtt{Turing}(M)$

some graph expression w with $\mathrm{gra}(w) = G$. Then $G_2 = f'(w)^\bullet$ by Theorem 1. Hence $w \in \mathrm{Dom}(f')$ and, by Definition 5, $\mathrm{gra}(w) \in \mathrm{Dom}(f)$ and $f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w))$. So $f'(w)$ is a graph expression and hence, by Lemma 2, $\mathrm{gra}(f'(w)) = H$. Thus $f(G) = f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w)) = H$.

"If": Let $f(G) = H$, and let w be a graph expression with $\mathrm{gra}(w) = G$. Then, by Lemma 1, $G \rightarrow_{\texttt{Encode}} w^\bullet$. By the computability of f, $f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w))$. Thus $w \in \mathrm{Dom}(f')$ and $f'(w)$ is a graph expression. By Theorem 1, $w^\bullet \rightarrow_{P_{f'}} f'(w)^\bullet$. By Lemma 2, $\mathrm{gra}(f'(w)) = f(\mathrm{gra}(w)) = f(G) = H$ implies $f'(w)^\bullet \rightarrow_{\texttt{Decode}} H$. Thus $G \rightarrow_{\texttt{Encode}} w^\bullet \rightarrow_{P_{f'}} f'(w)^\bullet \rightarrow_{\texttt{Decode}} H$.   □

The rest of this section is devoted to Lemmata 1 and 2 which give programs for encoding abstract graphs as graph expressions, and decoding graph expressions back into abstract graphs. Let $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}$ be any label alphabets such that $\mathcal{C}_{1V} \cup \mathcal{C}_{2V} \cup \mathcal{C}_{1V} \times \{'\} \cup \{\bullet, *\} \subseteq \mathcal{C}_V$ and $\mathcal{C}_{1E} \cup \mathcal{C}_{2E} \cup \mathcal{C}_{1V} \cup \{1, 2, 2', \#, =, ?, >, \bowtie\} \subseteq \mathcal{C}_E$, where we assume that the symbols $\bullet$, $*$, $2'$, $=$, $?$, $>$ and $\bowtie$ do not occur in $\mathcal{C}_{iV}$ and $\mathcal{C}_{iE}$ $(i = 1, 2)$.

Lemma 1. *There is a program* $\texttt{Encode}$ *such that for all G in $\mathcal{A}_{\mathcal{C}_1}$ and H in $\mathcal{A}_{\mathcal{C}}$,*

$$G \rightarrow_{\texttt{Encode}} H \text{ if and only if } G = \mathrm{gra}(w) \text{ and } H = w^\bullet \text{ for some graph}$$

expression w.

Proof. The program $\texttt{Encode}$ is given by

$$\texttt{Encode} = \texttt{Ite}(\emptyset, \texttt{Encode}_1, \texttt{Encode}_2)$$

where $\emptyset$ is the empty graph and $\texttt{Encode}_1$ and $\texttt{Encode}_2$ encode the empty graph and non-empty abstract graphs, respectively. While $\texttt{Encode}_1$ just creates a single node with label $\bullet$, $\texttt{Encode}_2$ consists of three subprograms:

$$\texttt{Encode}_2 = \langle \texttt{Prepare}; \texttt{Bundle}; \texttt{Compose} \rangle.$$

$\texttt{Prepare}$ prepares an abstract graph for encoding by representing node labels as edge labels and decorating each node by a chain of edges labelled with 1. The program $\texttt{Bundle}$ transforms an abstract graph into a bundle of string graphs each of which represents a node or an edge of the original graph. $\texttt{Compose}$ composes the string graphs in the bundle by connecting them with #-labelled edges and attaching a #-edge at the begin and the end of the resulting graph.

The program $\texttt{Prepare}$ consists of four subprograms:[3]

$$\texttt{Prepare} = \langle \texttt{Choose}; \texttt{Inc}\downarrow; \texttt{Relabel}\downarrow; \texttt{Stop} \rangle \downarrow.$$

Here $\texttt{Choose}$ selects a labelled node, relabels it into $\bullet$, attaches a 1-labelled edge, and decorates the source of this edge by a loop with the original node label. The

---

[3] $\texttt{Prepare}$ contains rules that *relabel* nodes, that is, rules in which the node labelling function of the interface is partial. These rules can be simulated by programs with ordinary rules. We omit the details for space reasons; they will be given in a long version of this paper.

program $\mathtt{Inc}\!\downarrow$ attaches a chain of 1-labelled edges to a loop-marked node and marks the begin of the chain by a loop. The length of the chain coincides with the number of not yet relabelled nodes in the current graph. Note that visited nodes temporarily get auxiliary labels to prevent further visiting. After termination of $\mathtt{Inc}\!\downarrow$ the nodes get their original labels be the subprogram $\mathtt{Relabel}\!\downarrow$. The elementary program $\mathtt{Stop}$ replaces a loop by an ordinary edge.

The program $\mathtt{Bundle}$ is defined by

$$\mathtt{Bundle} = \langle \mathtt{Separate};\ \mathtt{Copy}\!\downarrow;\ \mathtt{Redirect}\!\downarrow\rangle\!\downarrow,$$

where $\mathtt{Separate}$ separates an edge from the graph and initiates copying, $\mathtt{Copy}\!\downarrow$ copies the information about the source and the target of an edge, and $\mathtt{Redirect}\!\downarrow$ redirects edges such that a bundle of string graphs is obtained.

The program $\mathtt{Compose}$ is given by

$$\mathtt{Compose} = \langle\langle \mathtt{Initiate};\ \mathtt{Search}\!\downarrow\rangle\!\downarrow; \mathtt{Extend}_1; \mathtt{Extend}_2\rangle.$$

Here $\mathtt{Initiate}$ initiates the connection of two string graphs by connecting the begin nodes of two different string graphs with a #-labelled edge. The program $\mathtt{Search}\!\downarrow$ searches for the end of the first string graph and redirects the source of the #-labelled edge to the end of it. Finally, $\mathtt{Extend}_1$ and $\mathtt{Extend}_2$ add a node and a #-labelled edge at the begin and the end of the string graph.

The rules of $\mathtt{Encode}_2$ are given in Figure 6. By inspecting the rules, it is not difficult to check that $\mathtt{Encode}$ behaves as stated in the proposition.   □

Lemma 2. *There is a program* $\mathtt{Decode}$ *such that for every graph expression* w *in* Σ^* *and every abstract graph* G *in* $\mathcal{A}_{\mathcal{C}_2}$,

$$w^{\bullet} \rightarrow_{\mathtt{Decode}} G \text{ if and only if } \mathrm{gra}(w) = G.$$

Proof. Let $\mathtt{Decode}$ be the program

$$\mathtt{Decode} = \mathtt{Ite}(\bullet, \mathtt{Decode}_1, \mathtt{Decode}_2)$$

where $\bullet$ is the abstract graph consisting of a single node labelled with $\bullet$, and $\mathtt{Decode}_1$ and $\mathtt{Decode}_2$ are programs for decoding $\bullet$ and string graphs representing nonempty graph expressions, respectively. While $\mathtt{Decode}_1$ consists of a single rule deleting $\bullet$, $\mathtt{Decode}_2$ consists of three subprograms:

$$\mathtt{Decode}_2 = \langle \mathtt{Decompose};\ \mathtt{Interweave};\ \mathtt{MakeNode}\rangle.$$

$\mathtt{Decompose}$ decomposes the string graph of a nonempty graph expression into a bundle of string graphs representing nodes and edges, $\mathtt{Interweave}$ interweaves this bundle into an edge-labelled graph, and $\mathtt{MakeNode}$ transforms this graph into a graph with node and edge labels and removes auxiliary information.

The program $\mathtt{Decompose}$ is given by

$$\mathtt{Decompose} = \langle \mathtt{Cut}_1;\ \mathtt{Cut}_2;\ \mathtt{Deco}\!\downarrow\rangle,$$
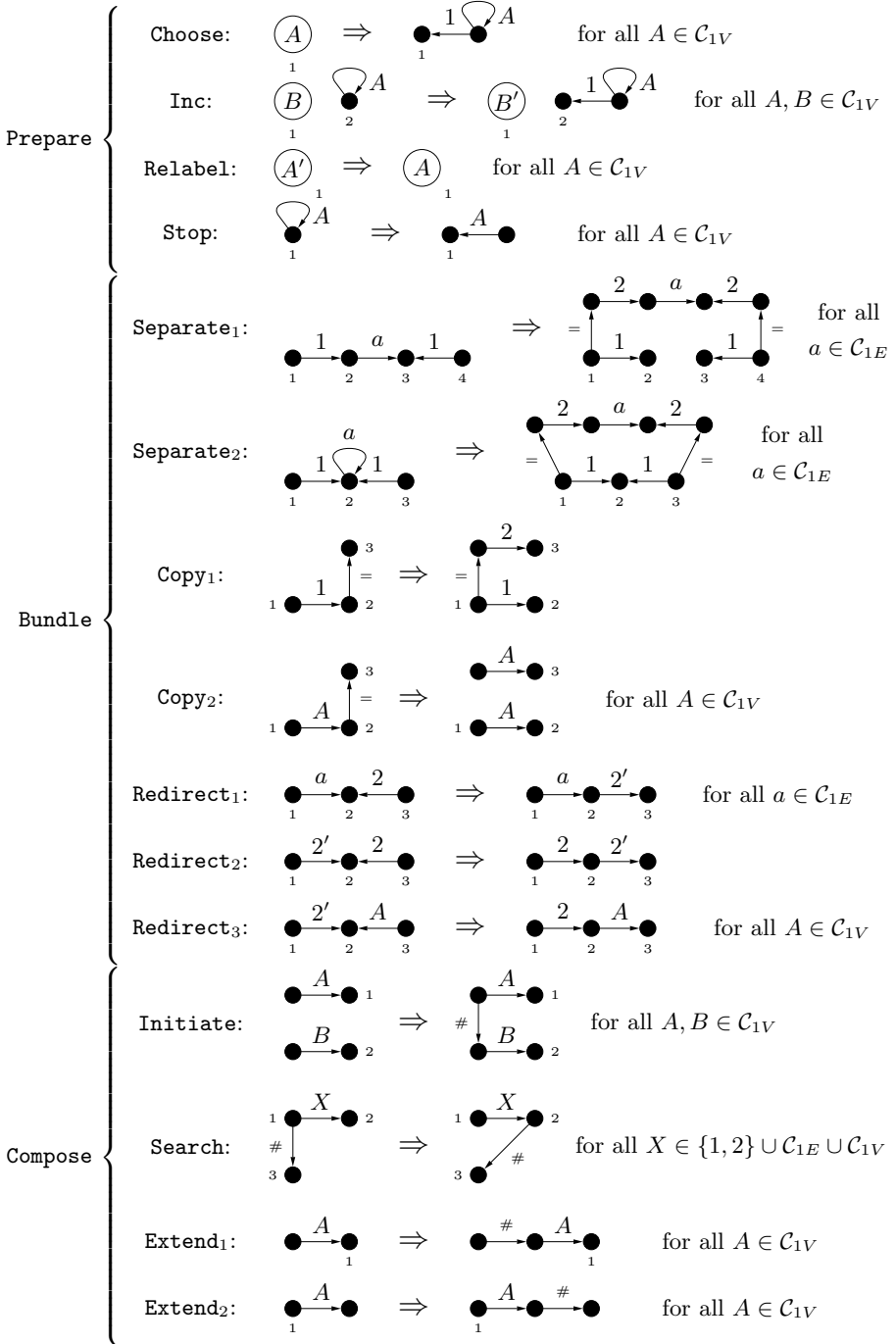
Fig. 6. The rules of the program $\texttt{Encode}_2$

where Cut_1 and Cut_2 remove the outer #-labelled edges together with the outermost nodes, and $\text{Deco}{\downarrow}$ removes the inner #-labelled edges. This results in a bundle of string graphs representing nodes and edges.

The program Interweave is defined as follows:

$$\text{Interweave} = \langle \text{Redirect}{\downarrow};\ \text{MarkAll};\ \text{Checkid}{\downarrow} \rangle.$$

Here Redirect $\downarrow$ redirects the edges representing the target node of an edge, MarkAll with

$$\text{MarkAll} = \langle \text{Select};\ \text{Connect}{\downarrow};\ \text{Relabel}{\downarrow} \rangle{\downarrow}$$

marks all pairs of representations which have to be checked with respect to coincidence, and Checkid with

$$\text{Checkid} = \langle \text{Initiate};\text{Compare}{\downarrow};\text{Delete}{\downarrow};\ \langle \text{Ident};\ \text{GarColl}{\downarrow} \rangle{\downarrow} \rangle$$

selects a pair and compares the representations. If the representations do not coincide, it finishes the comparison by deleting the comparison edge. Otherwise, it identifies two nodes[4] and performs "garbage collection".

Finally, MakeNode is given by

$$\text{MakeNode} = \langle \text{MakeNode}_1;\ \text{MakeNode}_2{\downarrow} \rangle{\downarrow},$$

where MakeNode_1 transforms edges representing nodes into nodes and removes auxiliary edges and nodes, and $\text{MakeNode}_2{\downarrow}$ removes further auxiliary information from the graph.

The rules of Decode_2 are given in Figure 7. By inspecting the rules one can see that Decode transforms graph expressions in form of string graphs into abstract graphs, and that it is correct in the sense that for every graph expression w in Σ^* and every abstract graph G in $\mathcal{A}_{\mathcal{C}_2}$, $w^\bullet \to_{\text{Decode}} G$ if and only if $\text{gra}(w) = G$.   □

7   Minimality

Our programming language defined in Section 3 is minimal in that omitting either sequential composition or iteration results in a computationally incomplete language. For the proof of this fact we call a function $f\colon \mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}}$ *cyclic* if there are some G in $\mathcal{A}_{\mathcal{C}}$ and $n \geq 2$ such that $f(G) \neq G$ and $f^n(G) = G$.

Lemma 3. *No cyclic function is computable by a program of the form $P{\downarrow}$.*

Proof. Let $f\colon \mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}}$ be cyclic and consider some G in $\mathcal{A}_{\mathcal{C}}$ and $n \geq 2$ such that $f(G) \neq G$ and $f^n(G) = G$. Suppose that there is a program $P{\downarrow}$ such that $\to_{P{\downarrow}} = f$. Then $f^{n-1}(G) \to_{P{\downarrow}} G$ and hence G is a normal form with respect to $\to_P$. But $G \to_{P{\downarrow}} f(G)$ and $G \neq f(G)$ imply $G \to_P^+ f(G)$. Thus G cannot be a normal form, a contradiction.   □

---

[4] Ident contains a rule $\langle L \leftarrow K \to R \rangle$ where $K \to R$ is not injective, but it can be replaced by a program in which all rules have injective morphisms. We omit this program for space reasons.
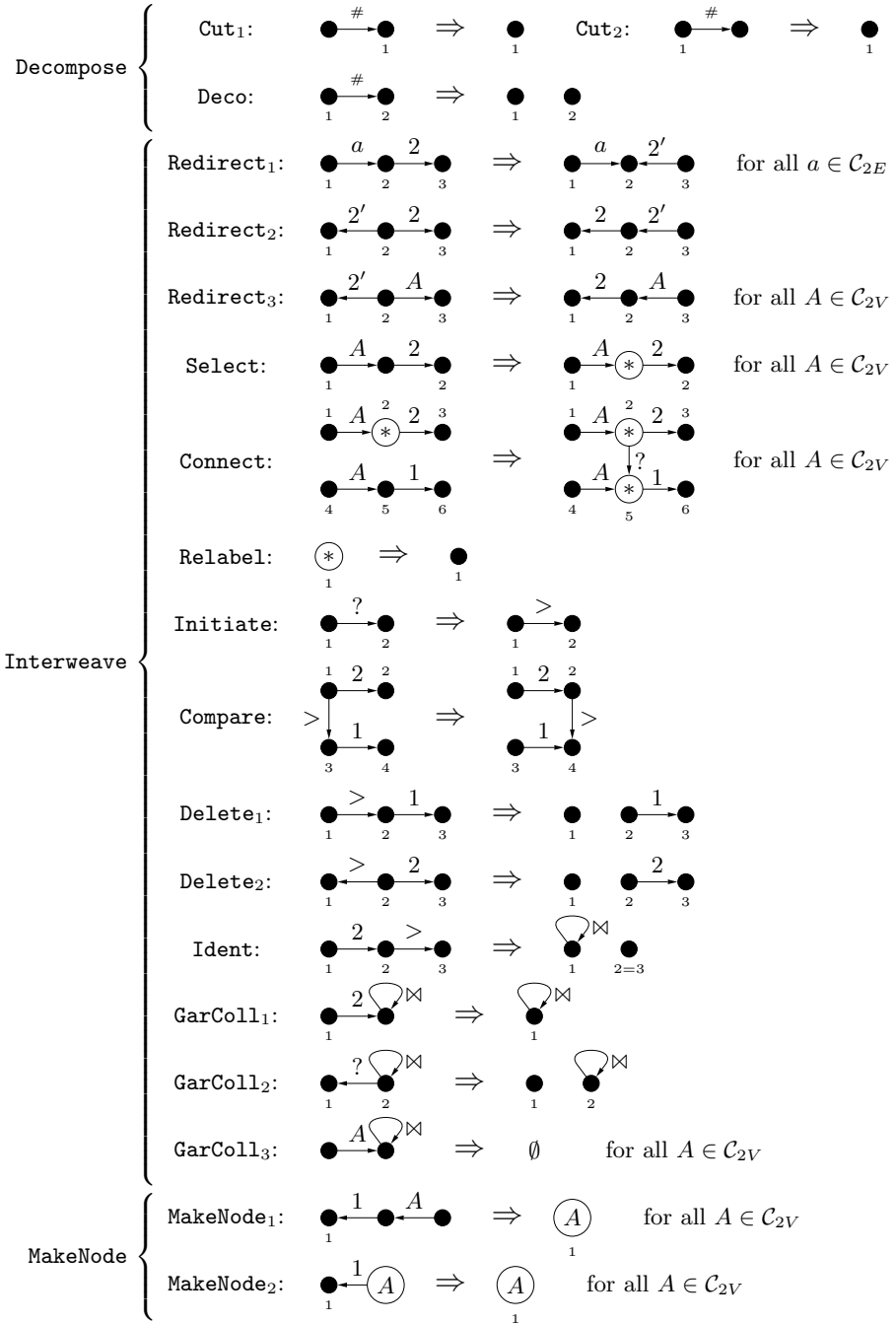
Fig. 7. The rules of the program Decode$_2$

For example, the function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ discussed in Example 1.2 is cyclic since converse(converse(G)) = G for every G in $\mathcal{A}_\mathcal{C}$. Hence a program computing this function cannot have an outermost iteration construct.

Theorem 3. *The set of programs without sequential composition is computationally incomplete.*

Proof. The function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ of Example 1.2 is computable and cyclic. By Lemma 3, a program P without sequential composition that computes this function has to be elementary. Let n be the largest number of edges occurring in the left-hand side of any rule in P. Consider $[G]$ in $\mathcal{A}_\mathcal{C}$ with $V_G = \{0, 1, \ldots, n+2\}$, $E_G = \{1, \ldots, n+2\}$ and for $i = 1, \ldots, n+2$, $s_G(i) = 0$ and $t_G(i) = i$. Now if $[G] \to_P [H]$, then H contains at least two edges with a common source node. So $[H] \neq$ converse($[G]$) and hence P does not compute converse. $\qquad\square$

An argument similar to the one in the above proof also shows that programs without iteration are computationally incomplete. For, it is clear that a program $\mathcal{R}_1; \ldots; \mathcal{R}_n$ whose component programs are elementary cannot convert graphs of arbitrary size.

It is worth mentioning that Theorem 3 has an alternative proof showing that programs without sequential composition cannot compute any function $f: \mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ satisfying (1) $f(\emptyset) \neq \emptyset$ and (2) for every $n \geq 0$ there is a graph G such that $\text{size}(G) + n < \text{size}(f(G))$. So the class of functions not computable without sequential composition does not just contain cyclic functions. The proof of this fact will be given in a long version of this paper.

8   Conclusion

We have answered the question what programming constructs are needed on top of (double-pushout) graph transformation rules to obtain a computationally complete programming language. It turned out that sequential composition and iteration of programs suffice for this purpose. Moreover, we have shown that omitting either of these two constructs makes the language incomplete.

These results should help to better understand the semantics and power of existing programming languages based on graph transformation rules, and they should also be useful for the design of new languages of this kind. In particular, due to the simplicity of our language, it should be feasible to prove computational completeness for a language in question by translating our programs into semantically equivalent programs of that language.

References

[Ehr79]   Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of Lecture Notes in Computer Science, 1–69. Springer-Verlag, 1979.

[ERT99]   Claudia Ermel, Michael Rudolf, and Gabi Taentzer. The AGG approach:
            Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and
            G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by
            Graph Transformation*, volume 2, chapter 14, 551–603. World Scientific,
            1999.

[FM98]   Pascal Fradet and Daniel Le Métayer. Structured Gamma. *Science of
            Computer Programming*, 31(2–3):263–289, 1998.

[GKS91]   John Glauert, Richard Kennaway, and Ronan Sleep. Dactl: An experi-
            mental graph rewriting language. In *Proc. Graph Grammars and Their
            Application to Computer Science*, volume 532 of *Lecture Notes in Compu-
            ter Science*, pages 378–395. Springer-Verlag, 1991.

[HMP99]   Annegret Habel, Jürgen Müller, Detlef Plump. Double-pushout graph
            transformation revisited. Bericht Nr. 7/99, Fachbereich Informatik, Uni-
            versität Oldenburg, 1999. Revised version to appear in *Mathematical Struc-
            tures in Computer Science*.

[LP98]   Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of
            Computation*. Prentice-Hall, second edition, 1998.

[Rod98]   Peter J. Rodgers. A graph rewriting programming language for graph dra-
            wing. In *Proc. 14th IEEE Symposium on Visual Languages*. IEEE Com-
            puter Society Press, 1998.

[SWZ99]   Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES ap-
            proach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski,
            and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing
            by Graph Transformation*, volume 2, chapter 13, 487–550. World Scientific,
            1999.

[Ues78]   Tadahiro Uesu. A system of graph grammars which generates all recursively
            enumerable sets of labelled graphs. *Tsukuba J. Math.* 2, 11–26, 1978.

[Wei87]   Klaus Weihrauch. *Computability*. Volume 9 of EATCS Monographs on
            Theoretical Computer Science. Springer-Verlag, 1987.