A Core Language for Graph Transformation (Extended Abstract)

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
York YO10 5DD, United Kingdom
`det@cs.york.ac.uk`

Abstract. We show how to program in a core programming language based on graph transformation rules. Our programs compute various functions and relations on graphs: the functions generating the transitive closure of a graph and the disjoint union of all subgraphs of a graph, the relation yielding all spanning trees of a graph, and functions testing for connectedness, acyclicity, and planarity. The language has a simple syntax consisting of just three constructs: nondeterministic one-step application of a set of rules, sequential composition, and iteration. It also has a simple formal semantics, and was shown to be computationally complete and minimal.

1 Introduction

In [3] we introduced a programming language for graph transformation consisting of three constructs: nondeterministic application of a rule from a set of graph transformation rules (according to the double-pushout approach), sequential composition of programs, and iteration in the form that a program is applied as long as possible. The language has a simple formal semantics and is computationally complete in that it allows to compute every computable partial function on labelled graphs. Moreover, the language is minimal in that omitting either sequential composition or iteration results in an incomplete language.

In this paper we show how to program in this language, by giving programs for the following problems: generating the transitive closure of a graph, generating the disjoint union of all subgraphs of a graph, generating all spanning trees of a graph, and testing whether a graph is connected, acyclic, or planar, respectively. The test functions are non-destructive in that they preserve the input graph, hence their programs can be used as components of programs which do further computations.

We consider the proposed programming language as a (declarative) core language because it lacks types and high-level constructs like procedures and modules. Having a simple yet computationally complete core language has several advantages:

- High-level languages for graph transformation which provide more programming comfort can be defined by mapping high-level constructs to the core. In this way possibly complex languages can be obtained which by their definition automatically get a formal semantics. Moreover, new languages are known to be computationally complete if they just cover the core language.
- Implementations for new languages based on the core can be rapidly obtained by extending an implementation of the core with translations from high-level constructs into core constructs.
- Frameworks and systems for formal reasoning on graph programs can be restricted to deal with the core language, since high-level programs can be translated into semantically equivalent programs in the core language. For example, frameworks for both program verification and program transformation can benefit from this approach.

The next section reviews the syntax and semantics of the core language and mentions computational completeness and minimality. Section 3 constitutes the main part of this paper, showing how to program in the language by means of several examples. Finally, two auxiliary program schemes implementing a copy operation and a conditional statement are given in the Appendix.

2 The language

Programs are based on sets of graph transformation rules according to the double-pushout approach, where rules are matched injectively and may have non-injective right-hand morphisms. See [3] and [2] for details.

Definition 1 (Syntax). *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 and can both be written as $\langle P_1; P_2; P_3 \rangle$; this is justified in that sequential composition is semantically the composition of binary relations, which is associative (see below).

We consider graph transformation over abstract graphs (isomorphism classes of graphs), denoting by $\mathcal{A}_\mathcal{C}$ the set of all abstract graphs over a label alphabet $\mathcal{C}$. 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. The *domain* of $\to$, denoted by $\mathrm{Dom}(\to)$, consists of all elements a in S such that $a \to b$ for some b.

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

(1) $\rightarrow_P \,=\, \Rightarrow_\mathcal{R}$ if P is an elementary program $\mathcal{R}$.
(2) $\rightarrow_{\langle P_1;P_2\rangle} \,=\, \rightarrow_{P_1} \circ \rightarrow_{P_2}$.
(3) $\rightarrow_{P\downarrow} \,=\, \{\langle G, H\rangle \mid G \rightarrow^*_P H$ and $H \notin \mathrm{Dom}(\rightarrow_P)\}$.

Consider now subalphabets $\mathcal{C}_1$ and $\mathcal{C}_2$ of $\mathcal{C}$ and a relation $\mathit{Rel} \subseteq \mathcal{A}_{\mathcal{C}_1} \times \mathcal{A}_{\mathcal{C}_2}$. We say that P *computes Rel* if $\mathit{Rel} = \rightarrow_P \cap\, (\mathcal{A}_{\mathcal{C}_1} \times \mathcal{A}_{\mathcal{C}_2})$, 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 $f\colon \mathcal{A}_{\mathcal{C}_1} \rightarrow \mathcal{A}_{\mathcal{C}_2}$, which are just special relations.

We remark that our programs can be formulated as semantically equivalent *graph transformation units* in the sense of [6]. Hence the results stated below apply to a certain sublanguage of graph transformation units, too.

Next we mention results from [3] on the computational completeness and the minimality of our language, without defining when a partial function on graphs is computable. The definition is based on an encoding of graphs as expressions. Intuitively, a partial function f on abstract graphs 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 $f(G)$. Moreover, f' is not defined on graph expressions denoting graphs on which f is not defined.

Theorem 1 (Completeness). *For every computable partial function f on abstract graphs there exists a program that computes f.*

The language is also minimal, meaning that omitting either sequential composition or iteration results in a computationally incomplete language.

Theorem 2 (Minimality).

1. *The set of programs without sequential composition is computationally incomplete.*
2. *The set of programs without iteration is computationally incomplete.*

For example, in [3] it is shown that the function converse: $\mathcal{A}_\mathcal{C} \rightarrow \mathcal{A}_\mathcal{C}$ which swaps source and target of each edge in a graph, is not computable by any program in the above two program classes.

3 Programming

In this section we show how to use our language for solving the following graph problems: generating the transitive closure of a graph, generating the disjoint union of all subgraphs of a graph, generating all spanning trees of a graph, and testing for connectedness, acyclicity, and planarity.

The programs below make extensive use of rules that relabel nodes, although the double-pushout approach is usually formulated for totally labelled graphs and label-preserving graph morphisms which prevent such rules. We employ a generalized form of rules where nodes in the interface can be unlabelled and morphisms can send unlabelled nodes to labelled nodes [4]. It can be shown, however, that for every rule r of generalized type there is a program $P(r)$ using ordinary rules such that $\rightarrow_{\{r\}} = \rightarrow_{P(r)}$.

We display rules by showing their left- and right-hand sides, using the convention that the interface graph consists of all numbered nodes.

3.1 Generating the transitive closure

The transitive closure of a graph G is obtained by adding an edge from a node u to another node v, whenever there is in G a directed path from u to v but no direct link. The function trans: $\mathcal{A}_\mathcal{C} \rightarrow \mathcal{A}_\mathcal{C}$, assigning to every graph its transitive closure, is computed by the following program:[1]

$$\texttt{TransClosure} = \underline{\text{if}}\ \emptyset\ \underline{\text{then}}\ \texttt{TransClosure}_1\ \underline{\text{else}}\ \texttt{TransClosure}_2.$$

The subprograms $\texttt{TransClosure}_1$ and $\texttt{TransClosure}_2$ are dealing with an empty and a non-empty input graph, respectively. While $\texttt{TransClosure}_1$ does not alter the input graph, $\texttt{TransClosure}_2$ is given as follows:

$$\langle\langle \texttt{Select}_1;\ \langle \texttt{Select}_2;\ \texttt{Connect}\downarrow;\ \texttt{Forget}_3\rangle\downarrow;\ \texttt{Forget}_2\rangle\downarrow;\ \texttt{Forget}_1\rangle,$$

where $\texttt{Connect} = \langle \texttt{Select}_3;\ \texttt{Unmark}\downarrow;\ \texttt{Link}\downarrow\rangle$. The main rules of $\texttt{TransClosure}_2$ are given in Figure 1. $\texttt{Select}_1$ selects a node and gives it the index 1, $\texttt{Select}_2$

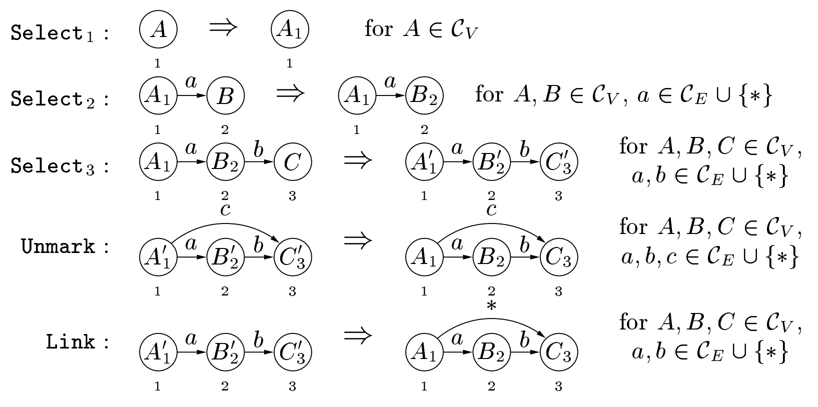

Select₁ :
for A ∈ C_V
Select₂ :
for A, B ∈ C_V, a ∈ C_E ∪ {*}
Select₃ :
for A, B, C ∈ C_V, a, b ∈ C_E ∪ {*}
Unmark :
for A, B, C ∈ C_V, a, b, c ∈ C_E ∪ {*}
Link :
for A, B, C ∈ C_V, a, b ∈ C_E ∪ {*}

Fig. 1. The main rules of $\texttt{TransClosure}_2$

gives a neighbour of this node the index 2, and `Connect` gives a neighbour of the

[1] The program scheme "$\underline{\text{if}}$ _ $\underline{\text{then}}$ _ $\underline{\text{else}}$ _" is defined in Appendix A.1.

latter node the index 3 and, if there is no edge from the first to the third node, links these nodes. To test whether the first and the third node are already linked, $\texttt{Select}_3$ first marks the three nodes. If a link exists, then `Unmark` removes the marks; in this case `Link` is not applicable. If the first and the third node are not yet linked, then `Link` inserts an edge and removes the marks. Note that the downarrows attached to `Unmark` and `Link` ensure that `Connect` is defined in cases where `Unmark` or `Link` is not applicable. Finally, for $i = 1, 2, 3$, the subprogram $\texttt{Forget}_i$ removes each occurrence of mark i.

3.2 Generating the disjoint union of all subgraphs

Our next program transforms an input graph into the disjoint union of all its subgraphs. `PowerGraph` will be used as a subprogram in the planarity test of subsection 3.6, but is interesting in its own right.

We consider here subgraphs that are obtained by edge deletions because only these matter for the planarity test. Extending the program to cover arbitrary subgraphs is straightforward.

In `PowerGraph` we use a subprogram `Tag` which adds a “tag” to a given graph by creating a unique auxiliary node and pointers from this node to all nodes in the graph. This form of tagging allows to identify different subgraphs of a graph. `Tag` is defined by

$$\texttt{Tag} = \langle \texttt{CreateTag};\ \texttt{LinkNode}\!\downarrow;\ \texttt{Restore}\!\downarrow\rangle,$$

where the rule sets are shown in Figure 2. In these rules, the “invisible” edge label, the node label τ, and the labels $\{A^* \mid A \in \mathcal{C}_V\}$ are all fresh.

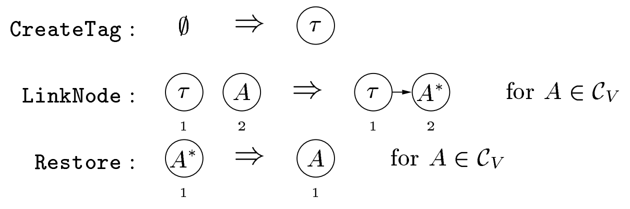

CreateTag :
$\emptyset \Rightarrow \tau$
LinkNode :
τ A $\Rightarrow$ τ A^*
1 2 1 2
for $A \in \mathcal{C}_V$
Restore :
$A^* \Rightarrow A$
1 1
for $A \in \mathcal{C}_V$

Fig. 2. The rules of `Tag`

Now `PowerGraph` is the program

$$\langle \texttt{Tag};\ \langle \texttt{PickGraph};\ \langle \texttt{PickEdge};\ \texttt{Copy};\ \texttt{RemoveEdge}\rangle\!\downarrow;\ \texttt{Untag}\rangle\!\downarrow;\ \texttt{CleanUp}\!\downarrow\rangle,$$

where `Copy` is an auxiliary program for copying a graph which is defined in Appendix A.2. The other subprograms are shown in Figure 3. The program `PickGraph` picks a tagged graph and removes all marks that have possibly been

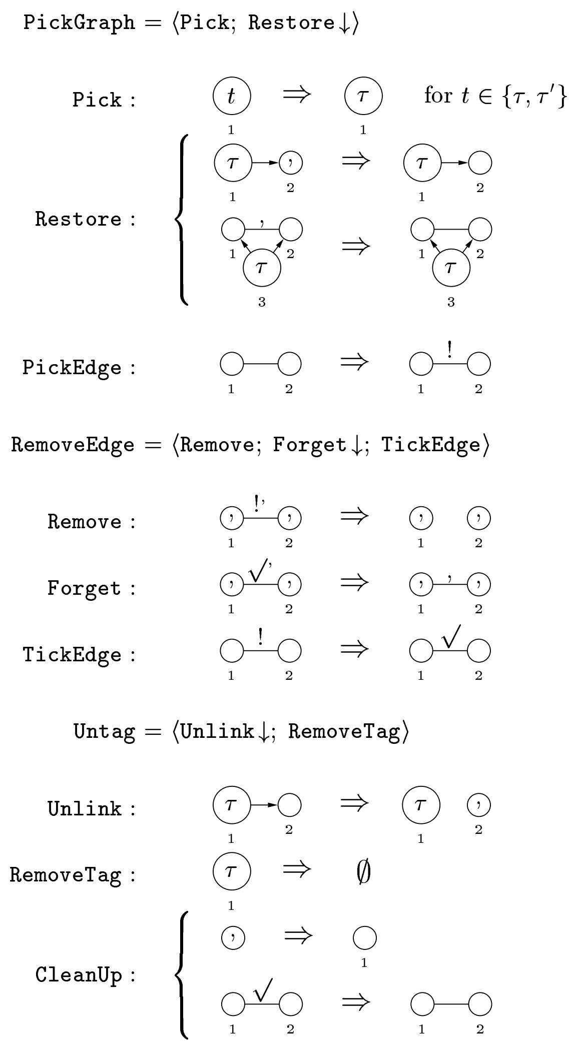

PickGraph = ⟨Pick; Restore↓⟩
Pick :
for $t \in \{\tau, \tau'\}$
Restore :
PickEdge :
RemoveEdge = ⟨Remove; Forget↓; TickEdge⟩
Remove :
Forget :
TickEdge :
Untag = ⟨Unlink↓; RemoveTag⟩
Unlink :
RemoveTag :
CleanUp :

Fig. 3. Some subprograms of `PowerGraph`

created by a previous copy operation, yielding the graph "under consideration". The program `PickEdge` picks an edge in the graph under consideration and marks it with !. Then `Copy` (see Appendix A.2) copies the graph under consideration, and `RemoveEdge` removes the copy of the picked edge (marked with !′) and marks the picked edge with $\surd$. By repeating this process as long as possible, one obtains copies of all graphs that result from the original graph by removing a single edge. Afterwards the graph under consideration is "frozen" by removing its tag and marking all nodes with ′. The latter prevents further copy operations on the graph. Then `PickGraph` picks a new tagged graph and the whole process starts again.

3.3 Generating all spanning trees

A subgraph S of a graph G is a spanning tree of G if the undirected graph underlying S is a tree that contains all nodes of G. Our program will "highlight" the edges of a spanning tree of a connected component of the input graph by marking them with $*$. Let Spanning $\subseteq \mathcal{A}_C \times \mathcal{A}_{C'}$ be the relation with $(G, G') \in$ Spanning if and only if G' is obtained from G by marking the edges of a spanning tree of any connected component of G with $*$. The relation Spanning is computed by the program

$$\texttt{Spanning} = \ \underline{\text{if}}\ \emptyset\ \underline{\text{then}}\ \texttt{Spanning}_1\ \underline{\text{else}}\ \texttt{Spanning}_2,$$

where $\texttt{Spanning}_1$ does not alter its input and $\texttt{Spanning}_2$ is given by

$$\texttt{Spanning}_2 = \langle \texttt{Select};\ \langle \texttt{Activate}{\downarrow};\ \texttt{Backtrack}\rangle{\downarrow};\ \texttt{Forget}\rangle.$$

This program does a depth-first search for finding a spanning tree. The search proceeds in a strictly sequential way since at any time at most one node is active, that is, marked with $\bullet$. The subprogram `Select` initially activates a node, while `Activate` activates a neighbour of the currently active node and deactivates the latter. `Backtrack` reactivates a neighbour of an active node. Finally, `Forget` forgets auxiliary node markings. The spanning tree of a component of a graph is then given by all $*$-marked edges and their incident nodes. The main rules of $\texttt{Spanning}_2$ are shown in Figure 4.

To compute spanning trees without backtracking, we can replace $\texttt{Spanning}_2$ with

$$\texttt{Spanning}'_2 = \langle \texttt{Select};\ \texttt{Activate}'{\downarrow};\ \texttt{Forget}\rangle.$$

The subprogram `Select` activates a node while `Activate`′ looks for an activated node, activates a neighbour, and remains active. So several nodes can be active simultaneously. As before, the spanning tree is determined by the $*$-marked edges. The main rules of $\texttt{Spanning}'_2$ are shown in Figure 5.

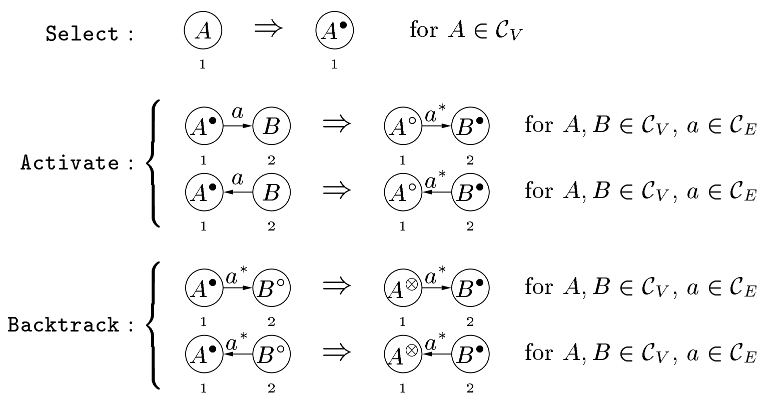

Select :
for $A \in \mathcal{C}_V$
Activate :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
Backtrack :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$

Fig. 4. The main rules of $\texttt{Spanning}_2$

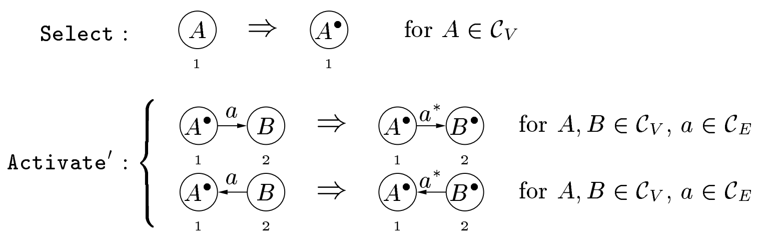

Select :
for $A \in \mathcal{C}_V$
Activate$'$:
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$

Fig. 5. The main rules of $\texttt{Spanning}'_2$

3.4 Testing for connectedness

A directed graph is connected if there is a path between each two nodes in the underlying undirected graph. The function connected?: $\mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}'}$ with

$$\text{connected?}(G) = \begin{cases} G + \textcircled{1} & \text{if } G \text{ is connected,} \\ G + \textcircled{0} & \text{otherwise} \end{cases}$$

is computed by the program

$$\texttt{Connected?} = \underline{\text{if}}\ \emptyset\ \underline{\text{then}}\ \texttt{Connected?}_1\ \underline{\text{else}}\ \texttt{Connected?}_2.$$

Here $\texttt{Connected?}_1$ creates a single node with label 1 and $\texttt{Connected?}_2$ is given by

$$\texttt{Connected?}_2 = \langle \texttt{Select};\ \texttt{Mark}\!\downarrow;\ \texttt{Check};\ \texttt{Forget}\!\downarrow \rangle.$$

The program `Select` picks any node, $\texttt{Mark}\!\downarrow$ marks all nodes that are reachable from the picked node, $\texttt{Check} = \langle \texttt{Initiate};\ \texttt{Test}\downarrow \rangle$ adds an auxiliary node with label 1 to the graph and checks whether any unmarked nodes remain, and $\texttt{Forget}\!\downarrow$ removes all marks. The rules of $\texttt{Connected?}_2$ are shown in Figure 6.

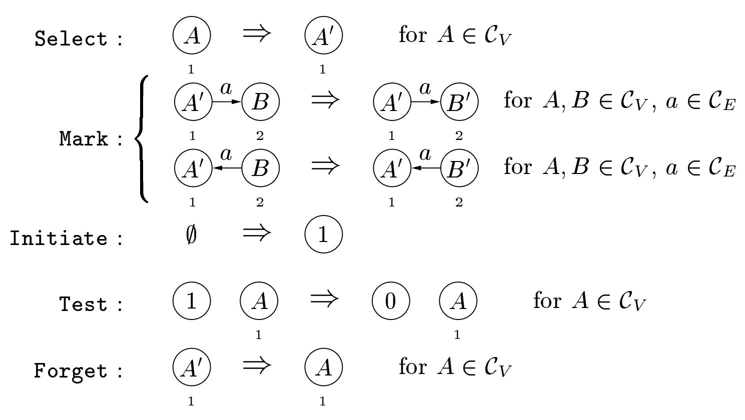

Select :
for $A \in \mathcal{C}_V$
Mark :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
Initiate :
Test :
for $A \in \mathcal{C}_V$
Forget :
for $A \in \mathcal{C}_V$

Fig. 6. The rules of `Connected?`$_2$

3.5 Testing for acyclicity

The function acyclic?: $\mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}'}$ with

$$\text{acyclic?}(G) = \begin{cases} G + \textcircled{1} & \text{if } G \text{ is acyclic} \\ G + \textcircled{0} & \text{otherwise} \end{cases}$$

is computed by the following program:

$$\texttt{Acyclic?} = \langle \texttt{Copy};\ \texttt{Reduce};\ \texttt{Check};\ \texttt{GarColl}\!\downarrow \rangle.$$

The idea behind this program is that a graph is acyclic if and only if the rules of Figure 7 reduce it to a graph without edges. The program `Reduce` for reducing the copy of the input graph is given by

$$\texttt{Reduce} = \langle \texttt{MarkNonLeaf}\!\downarrow;\ \texttt{DeleteEdge};\ \texttt{DeleteEdge}\!\downarrow;\ \texttt{Restore}\!\downarrow \rangle\!\downarrow,$$

with rules as shown in Figure 7. `Reduce` first marks all nodes with at least one outgoing edge, so that all other nodes must be leaves (nodes without outgoing edges). Edges pointing to leaves cannot belong to cycles and hence can be safely removed. This may result in new leaves, so we remove all marks and start again. The reduction finishes if no edges pointing to leaves remain. Note that the twofold occurrence of `DeleteEdge` guarantees that `Reduce` terminates.

The program `Check` = ⟨`Add`; `Test`↓⟩ adds an auxiliary node with label 1 to the graph, checks whether the reduced copy contains an edge and, if so, changes the label of the auxiliary node to 0. Finally, `GarColl`↓ removes the remainder of the copy. See Figure 8 for the rules of `Check` and `GarColl`.

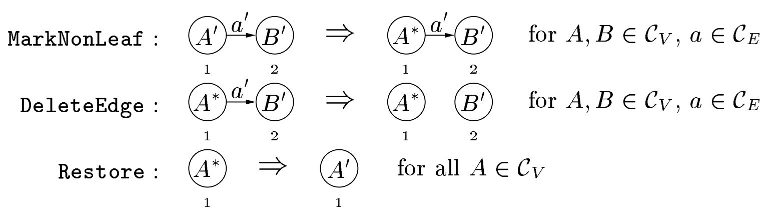

MarkNonLeaf :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
DeleteEdge :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
Restore :
for all $A \in \mathcal{C}_V$

Fig. 7. The rules of `Reduce`

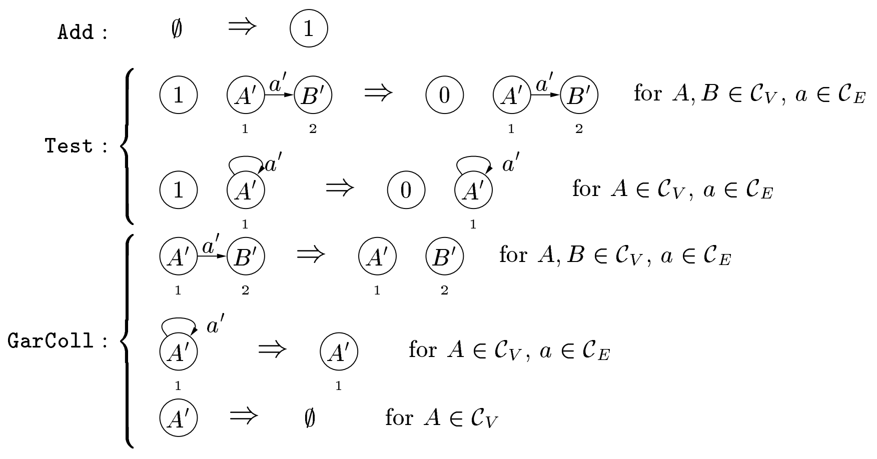

Add :
Test :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
for $A \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
GarColl :
for $A, B \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
for $A \in \mathcal{C}_V$, $a \in \mathcal{C}_E$
for $A \in \mathcal{C}_V$

Fig. 8. The rules of `Check` and `GarColl`

3.6 Testing for planarity

A graph is planar if it can be drawn on the plane without edge crossings. Our program for computing the function planar?: $\mathcal{A}_{\mathcal{C}} \to \mathcal{A}_{\mathcal{C}'}$ with

$$\text{planar?}(G) = \begin{cases} G + \textcircled{1} & \text{if } G \text{ is planar,} \\ G + \textcircled{0} & \text{otherwise} \end{cases}$$

is based on Kuratowski's Theorem.[2] By this theorem, an undirected and unlabelled graph is planar if and only if it has no subgraph homeomorphic to K_5 or $K_{3,3}$ [5]. Here K_5 is the complete graph with five nodes, and $K_{3,3}$ is the complete bipartite graph whose node sets both have 3 nodes, see Figure 9. Furthermore,

[2] It is known that this leads to an algorithm of exponential complexity, which is evident for our solution as `PowerGraph` produces a graph of exponential size. But simulating one of the subtle linear algorithms for planarity testing (see for example [1]) is beyond the scope of this paper.

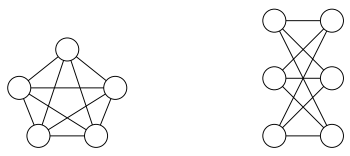


Fig. 9. The graphs K_5 and $K_{3,3}$

two graphs are homeomorphic if both can be obtained from the same graph by a sequence of edge subdivisions. It follows that a graph is planar if and only if no subgraph reduces by repeated applications of the rule `Contract` in Figure 10 to a graph containing K_5 or $K_{3,3}$.

The function planar? is computed by the program

$$\texttt{Planar?} = \langle \texttt{Copy};\ \texttt{Simplify}\!\downarrow;\ \texttt{PowerGraph};\ \texttt{Contract}\!\downarrow;\ \texttt{Check};\ \texttt{GarColl}\rangle$$

whose rules are shown in Figure 10. The subprogram `Simplify`$\downarrow$ first transforms the copy of the input graph to an undirected, unlabelled graph without multiple edges and loops. We draw unlabelled nodes as unfilled circles, and undirected edges as lines without arrowheads. Implicitly, unlabelled nodes and edges carry a special "invisible"label and undirected edges are pairs of edges pointing in opposite directions.

The program `PowerGraph` of Section 3.2 is used to generate the disjoint union of all subgraphs of the copied input graph (where we only consider subgraphs resulting from edge deletions). Then `Contract`$\downarrow$ contracts the obtained graph as long as possible, and

$$\texttt{Check} = \langle \texttt{Initiate};\ \texttt{Test}(K_5)\!\downarrow;\ \texttt{Test}(K_{3,3})\!\downarrow\rangle$$

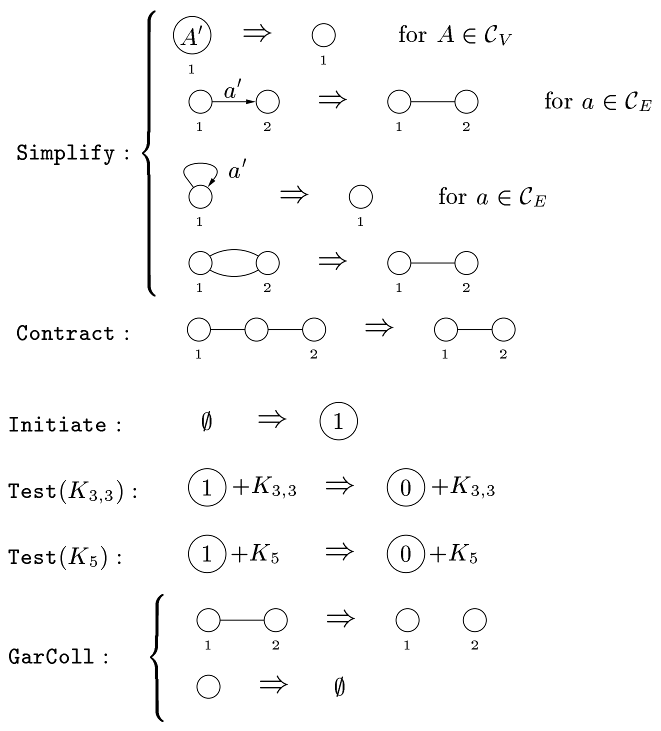
checks whether the contracted graph contains K_5 or $K_{3,3}$. The program first adds an auxiliary node with label 1 which, if the check was successful, is changed to 0. The interfaces of the rules `Test`(K_5) and `Test`$(K_{3,3})$ consist of K_5 and $K_{3,3}$, respectively. Finally, `GarColl` removes the remainder of the simple graph.

A Appendix

A.1 The program scheme $\underline{\text{if}}$ _ $\underline{\text{then}}$ _ $\underline{\text{else}}$ _

We use a program scheme $\underline{\text{if}}\ K\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2$ 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 $G \rightarrow_{\underline{\text{if}}\ K\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2} H$ if and only if $G = K$ and $G \rightarrow_{P_1} H$ or $G \neq K$ and $G \rightarrow_{P_2} H$. The scheme is defined by

$$\underline{\text{if}}\ K\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2 = \langle \texttt{Check}(K);\ \langle \texttt{Delete}_1;\ P_1\rangle\!\downarrow;\ \langle \texttt{Delete}_2;\ P_2\rangle\!\downarrow\rangle,$$



Simplify :
A' ⇒ for $A \in \mathcal{C}_V$
a' ⇒ for $a \in \mathcal{C}_E$
a' ⇒ for $a \in \mathcal{C}_E$
Contract :
Initiate : $\emptyset$ ⇒ 1
Test($K_{3,3}$) : 1 $+K_{3,3}$ ⇒ 0 $+K_{3,3}$
Test(K_5) : 1 $+K_5$ ⇒ 0 $+K_5$
GarColl :
⇒ $\emptyset$

Fig. 10. The rules of `Planar`?

where `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 `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 `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.

A.2 The program scheme `Copy`

We also use a program scheme `Copy` for copying graphs. Given a label alphabet $C = \langle C_V, C_E \rangle$, let $C^{©} = \langle C_V \cup (\mathcal{C}_V \times \{'\}) \cup (\mathcal{C}_V \times \{^*\}),\ C_E \cup (\mathcal{C}_E \times \{'\}) \cup (\mathcal{C}_E \times \{^*\}) \rangle$. Labels $\langle l, ' \rangle$ and $\langle l, ^* \rangle$ from $C^{©}$ are written l' and l^*, respectively. `Copy` transforms a graph G over C into the graph $G + G'$ over $C^{©}$, where G' is obtained from G by replacing each label l with l'. `Copy` is defined as follows:

$$\texttt{Copy} = \langle \texttt{CopyNode}{\downarrow};\ \texttt{CopyEdge}{\downarrow};\ \texttt{Restore}{\downarrow} \rangle.$$

The rules of `Copy` are shown in Figure 11.

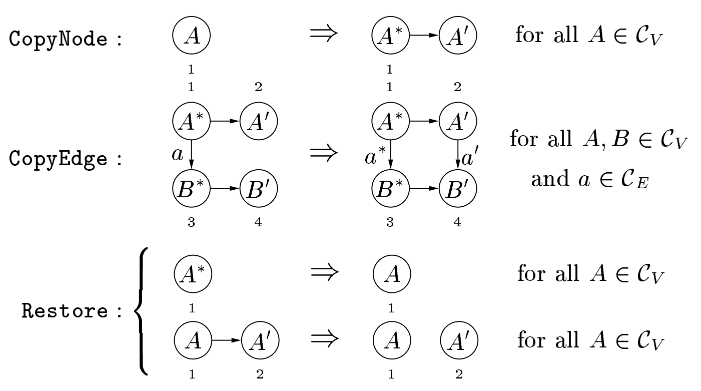

CopyNode :
for all $A \in \mathcal{C}_V$
CopyEdge :
for all $A, B \in \mathcal{C}_V$ and $a \in \mathcal{C}_E$
Restore :
for all $A \in \mathcal{C}_V$
for all $A \in \mathcal{C}_V$

Fig. 11. The rules of `Copy`

References

1. Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.
2. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
3. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS '01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.
4. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In preparation, 2002.
5. Frank Harary. *Graph Theory.* Addison-Wesley, 1969.
6. Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11:690–723, 1999.