GCM 2021

Graph Computation Models

12th International Workshop

Pre-Proceedings

Berthold Hoffmann and Mark Minas, Editors

Bremen, München, June 2021

Table of Contents

Session 1: Grammars and Term Rewriting

Session 2: Applications

Session 3: Programming

Preface

This volume contains the proceedings of the Twelfth International Workshop on Graph Computation Models (GCM 2021)[1]. Due to the restrictions related to Covid-19, GCM 2021 will be held as an online workshop on June 22, 2021.

Graphs are common mathematical structures that are visual and intuitive. They constitute a natural and seamless way for system modelling in science, engineering and beyond, including computer science, biology, business process modelling, etc. Graph computation models constitute a class of very high-level models where graphs are first-class citizens. The aim of the International GCM Workshop series is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation. It promotes the cross-fertilizing exchange of ideas and experiences among senior and young researchers from the different communities interested in the foundations, applications, and implementations of graph computation models and related areas.

Previous editions of GCM series were held in Natal, Brazil (2006), Leicester, UK (2008), Enschede, The Netherlands (2010), Bremen, Germany (2012), York, UK (2014), L'Aquila, Italy (2015), Wien, Austria (2016), Marburg, Germany (2017), Toulouse, France (2018), Eindhoven, The Netherlands (2019), and online (2020).

These proceedings contain seven accepted papers. All submissions were subject to careful refereeing. The topics of accepted papers range over a wide spectrum, including theoretical aspects of graph transformation, verification and parsing techniques as well as application issues of graph computation models. Selected papers from these proceedings will be published online by Electronic Proceedings in Theoretical Computer Science (EPTCS, `http://www.eptcs.org/`). We would like to thank all the people who contributed to the success of GCM 2021, especially the Program Committee and the additional reviewers for their valuable contributions to the selection process, as well as the contributing authors without whom this volume would not exist.

June 15, 2021 Berthold Hoffmann and Mark Minas

Program chairs of GCM 2021

[1] GCM 2021 web site: `https://sites.google.com/view/gcm2021/`

Program Committee of GCM 2021

Andrea Corradini, Universita di Pisa, Italy
Frank Drewes, Umeå universitet, Sweden
Rachid Echahed, CNRS and Univ. Grenoble Alpes, France
Annegret Habel, Universität Oldenburg, Germany
Berthold Hoffmann, Universität Bremen, Germany (co-chair)
Gabor Karsai, Vanderbilt University, USA
Barbara König, Universität Duisburg-Essen, Germany
Hans-Jörg Kreowski, Universität Bremen, Germany
Leen Lambers, Universität Potsdam, Germany
Mark Minas, Universität der Bundeswehr München, Germany (co-chair)
Fernando Orejas, Universitat Politècnica de Catalunya, Spain
Detlef Plump, University of York, United Kingdom

Additional Reviewers

Graham Campbell, Newcastle University, United Kingdom
Brian Courtehoute, University of York, United Kingdom
Okan Özkan, Universität Oldenburg, Germany

Grammars Based on a Logic of Hypergraph Languages

Tikhon Pshenitsyn
Department of Mathematical Logic and Theory of Algorithms *
Faculty of Mathematics and Mechanics
Lomonosov Moscow State University
GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation
tpshenitsyn at lpcs.math.msu.su

The hyperedge replacement grammar (HRG) formalism is a natural and well-known generalization of the context-free grammar. HRGs inherit a number of properties of context-free grammars, e.g. the pumping lemma. This lemma turns out to be a strong restriction in the hypergraph case: e.g. it implies that languages of unbounded connectivity cannot be generated by HRGs. We introduce a formalism that turns out to be more powerful than HRGs while having the same algorithmic complexity (NP-complete). This formalism is called hypergraph Lambek grammars; such grammars are based on the hypergraph Lambek calculus, which may be considered as a logic of hypergraph languages. We explain the underlying principles of hypergraph Lambek grammars, establish their basic properties, and show some languages of unbounded connectivity that can be generated by them (e.g. the language of all graphs, the language of all bipartite graphs, the language of all regular graphs).

1 Introduction: Productions vs Categories

Formal language theory is an area at the intersection of linguistics, logic, programming, mathematics etc., which studies an issue of how complex, unboundedly large but in some sense regular families of objects (strings, terms, graphs, pictures, ...) can be described using some kind of finite-sized mechanism. There exist numerous kinds of formal grammars based on different principles. For instance, the standard context-free grammar approach deals with strings; it includes terminal and nonterminal alphabets, the start nonterminal symbol S, and the list of productions of the form $X \to \alpha$, which allow one to replace the nonterminal symbol X by the string α.

Example 1. Look at the following "linguistic" example (which is extremely primitive from the linguistic point of view):

$$\begin{array}{rl} \text{Productions:} & S \to NP \text{ sleeps} \qquad NP \to \text{the } N \qquad N \to \text{cat} \\ \text{Derivation:} & S \Rightarrow NP \text{ sleeps} \Rightarrow \text{the } N \text{ sleeps} \Rightarrow \text{the cat sleeps} \end{array}$$

The production $[S \to NP \text{ sleeps}]$ is *lexicalized*; that is, there is exactly one terminal object (*sleeps* here) in its right-hand side. Let us transform it as follows: sleeps $\triangleright NP\backslash S$. The $\triangleright$ sign is to be read as "is of the type", and $NP\backslash S$ is the type of such objects u that, whenever we add an object v of the type NP (NP stands for *noun phrase*) to the left of u, vu forms an object of the type S (S stands for *sentence*). Therefore, sleeps $\triangleright NP\backslash S$ means that the verb *sleeps* is such an object that whenever a noun phrase (singular) appears to its left, they together form a sentence. This is correct, to a first approximation: *the cat sleeps*, *Helen sleeps*, *a green colorless idea sleeps* etc. are correct English sentences. Similarly, we can transform the production $NP \to \text{the } N$ into a correspondence: the $\triangleright NP/N$ (N stands for nouns that

*The study was funded by RFBR, project number 20-01-00670 and by the Interdisciplinary Scientific and Educational School of Moscow University "Brain, Cognitive Systems, Artificial Intelligence". The author is a Scholarship holder of "BASIS" Foundation.

do not represent a specific object but rather represent a class of objects). Note that the direction of the division is different from the previous one. The following reduction laws hold: the left one $A, A\backslash B \to B$, and the right one $B/A, A \to B$. The type A/B ($B\backslash A$) can be understood as the type of functions that take an argument of the type B from the right (from the left resp.) and return a value of the type A.

In linguistics, however, it is not enough to have the above reduction laws to describe language phenomena of interest. E.g. sometimes it is useful to have the rule $NP \to S/(NP\backslash S)$ or $A/B, B/C \to A/C$. Besides, there is a need for an operation that would store pairs of units of certain types. E.g. when we consider sentences like *Tim gave the lemon to Melany and the lime to Amelie*, we would like to assign the type $((NP \cdot PP)\backslash(NP \cdot PP))/(NP \cdot PP)$ to the word *and* (where PP is the type of prepositional phrases with *to*: *to Melany*, *to Amelie*) since it receives a pair consisting of a noun phrase and a prepositional phrase from the right, and a similar pair from the left. $A \cdot B$ is the type of pairs uv such that u is of the type A, and v is of the type B; therefore, $A \cdot B$ works as concatenation.

The above connectives $\backslash, \cdot, /$ belong to the Lambek calculus (L) — a logical calculus introduced in [7]. Types in the Lambek calculus are built from primitive types $p_1, p_2, \ldots \in Pr$ using $\backslash, \cdot, /$. We focus on the Lambek calculus in the Gentzen style; this means that it deals with sequents, which are structures of the form $A_1, \ldots, A_n \to A$ for A_i, A being types. The calculus has one axiom and six inference rules of L:

$$\frac{}{A \to A}\ (\mathrm{Ax}) \qquad \frac{\Pi \to A \quad \Gamma, B, \Delta \to C}{\Gamma, \Pi, A\backslash B, \Delta \to C}\ (\backslash \to) \qquad \frac{A, \Pi \to B}{\Pi \to A\backslash B}\ (\to \backslash) \qquad \frac{\Gamma, A, B, \Delta \to C}{\Gamma, A \cdot B, \Delta \to C}\ (\cdot \to)$$

$$\frac{\Pi \to A \quad \Gamma, B, \Delta \to C}{\Gamma, B/A, \Pi, \Delta \to C}\ (/ \to) \qquad \frac{\Pi, A \to B}{\Pi \to B/A}\ (\to /) \qquad \frac{\Pi \to A \quad \Psi \to B}{\Pi, \Psi \to A \cdot B}\ (\to \cdot)$$

Hereinafter, primitive types are denoted by small Latin letters (in particular, from now on we write s instead of S, np instead of NP etc.), types are denoted by capital Latin letters, and sequences of types are denoted by capital Greek letters; besides, Π, Ψ above are nonempty. A sequent $\Pi \to A$ is called *derivable in* L (denoted $\mathrm{L} \vdash \Pi \to A$) if it can be obtained from axioms using rules.

Example 2. The following sequents are derivable in L (their derivations are presented below them):

- $np/n, n, np\backslash s \to s$;
- $np \to s/(np\backslash s)$;
- $p \to (p \cdot q)/q$.

$$\frac{s \to s \qquad \dfrac{np \to np \quad n \to n}{np/n, n \to np}\ (/ \to)}{np/n, n, np\backslash s \to s}\ (\backslash \to) \qquad \frac{\dfrac{s \to s \quad np \to np}{np, np\backslash s \to s}\ (\backslash \to)}{np \to s/(np\backslash s)}\ (\to /) \qquad \frac{\dfrac{p \to p \quad q \to q}{p, q \to p \cdot q}\ (\to \cdot)}{p \to (p \cdot q)/q}\ (\to /)$$

Summing up the above discussion, we note that the Lambek calculus, which has an algebraic and logical nature, can be used as a grammar formalism: a grammar is a correspondence between terminal objects and types. The intuition of the Lambek calculus types is formalized by the following

Definition 1. Given an alphabet Σ, we call any function $w : Pr \to \mathscr{P}(\Sigma^*)$ *a valuation*; this function assigns a formal language to each primitive type. It can be extended to types and sequents according to principles stated above; namely, $\overline{w}$ is defined as follows:

1. $\overline{w}(B\backslash A) = \{u \in \Sigma^* \mid \forall v \in \overline{w}(B)\ vu \in \overline{w}(A)\}$;
2. $\overline{w}(A/B) = \{u \in \Sigma^* \mid \forall v \in \overline{w}(B)\ uv \in \overline{w}(A)\}$;
3. $\overline{w}(A \cdot B) = \{uv \mid u \in \overline{w}(A), v \in \overline{w}(B)\}$;
4. $\overline{w}(A_1, \ldots, A_n) = \overline{w}(A_1) \cdot \ldots \cdot \overline{w}(A_n)$ where $\overline{w}(A) \cdot \overline{w}(B) = \overline{w}(A \cdot B)$;
5. $\overline{w}(\Pi \to A)$ is true if and only if $\overline{w}(\Pi) \subseteq \overline{w}(A)$.

Pentus [9] proved that $\mathrm{L} \vdash \Pi \to A$ if and only if $\overline{w}(\Pi \to A)$ is true for all valuations w. This allows us to call the Lambek calculus the logic of formal string languages in the sense that it describes all the true facts about formal languages in the signature $\backslash, \cdot, /, \subseteq$ and only them.

It is often important to work with more complex structures than strings. This is the reason why a wide variety of extensions of generative grammars to terms, graphs etc. has been introduced. In this paper, we focus on a particular approach called *hyperedge replacement grammars (HRGs)*. A survey on HRGs can be found in [3, 5]; in this paper, we mainly follow the definitions and notation from [3]. We chose HRGs as the basis for our studies since they are very close to context-free grammars in the sense of definitions, underlying mechanisms and properties. Our main goal is to extend the Lambek calculus and Lambek grammars to hypergraphs in a natural way and to study the resulting notions.

In Section 2, we define fundamental notions regarding hypergraphs and hyperedge replacement. In Section 3, we introduce the hypergraph Lambek calculus extending notions of types, sequents, axioms and rules. The formal definition of hypergraph Lambek grammars will be given in Section 4. There we also study power of these grammars; it turns out that they can generate more languages than HRGs, e.g. the language of all graphs, the language of all bipartite graphs, the language of all regular graphs. Since the membership problem for HL-grammars is NP-complete and since they generate all isolated-node bounded languages generated by HRGs, they can be considered as an attractive alternative to HRGs. In Section 6, we conclude and outline further research directions regarding HL and HL-grammars.

2 Preliminaries

Formal definitions of hypergraphs and of hyperedge replacement are given in this section according to the handbook chapter [3] on HRGs.

$\mathbb{N}$ includes 0. Σ^* is the set of all strings over the alphabet Σ including the empty string Λ. $\Sigma^{\circledast}$ is the set of all strings consisting of distinct symbols. The length $|w|$ of the word w is the number of symbols in w. The set of all symbols contained in a word w is denoted by $[w]$. If $f : \Sigma \to \Delta$ is a function from one set to another, then it is naturally extended to a function $f : \Sigma^* \to \Delta^*$ ($f(\sigma_1 \dots \sigma_k) = f(\sigma_1) \dots f(\sigma_k)$).

Let C be some fixed set of labels with the ranking function $rank : C \to \mathbb{N}$. In [3], this function is denoted as *type* instead of *rank*; here we use *rank* to avoid confusion with types of the calculus.

Definition 2. *A hypergraph G over C* is a tuple $G = \langle V, E, att, lab, ext \rangle$ where V is a set of *nodes*, E is a set of *hyperedges*, $att : E \to V^{\circledast}$ assigns a string (i.e. an ordered set) of *attachment* nodes to each edge, $lab : E \to C$ labels each edge by some element of C in such a way that $rank(lab(e)) = |att(e)|$ whenever $e \in E$, and $ext \in V^{\circledast}$ is a string of *external* nodes.

Components of a hypergraph G are denoted by $V_G, E_G, att_G, lab_G, ext_G$ resp.

In the remainder of the paper, hypergraphs are usually called just graphs, and hyperedges are called edges. Usual graphs (with hyperedges of rank 2) are called binary graphs. The set of all graphs with labels from C is denoted by $\mathscr{H}(C)$. Graphs are usually named by letters G and H.

In drawings of graphs, black dots correspond to nodes, labeled squares correspond to edges, *att* is represented by numbered lines, and external nodes are depicted by numbers in brackets. If an edge has exactly two attachment nodes, it can be depicted by an arrow (which goes from the first attachment node to the second one).

Note that Definition 2 implies that attachment nodes of each hyperedge are distinct, and so are external nodes. This restriction can be removed (i.e. we can consider graphs with loops), and all definitions will be preserved; however, in this paper, we stick to the above definition.

Definition 3. The function $rank_G$ (or *rank*, if G is clear) returns the number of nodes attached to an edge in a graph G: $rank_G(e) := |att_G(e)|$. If G is a graph, then $rank(G) := |ext_G|$.

Definition 4. A sub-hypergraph (or just subgraph) H of a graph G is a hypergraph such that $V_H \subseteq V_G$, $E_H \subseteq E_G$, and for all $e \in E_H$ $att_H(e) = att_G(e)$, $lab_H(e) = lab_G(e)$.

Definition 5. If $H = \langle \{v_i\}_{i=1}^n, \{e_0\}, att, lab, v_1 \dots v_n \rangle$, $att(e_0) = v_1 \dots v_n$ and $lab(e_0) = a$, then H is called *a handle*. In this work, we denote it by $a^\bullet$.

Definition 6. *An isomorphism* between graphs G and H is a pair of bijective functions $\mathscr{E} : E_G \to E_H$, $\mathscr{V} : V_G \to V_H$ such that $att_H \circ \mathscr{E} = \mathscr{V} \circ att_G$, $lab_G = lab_H \circ \mathscr{E}$, $\mathscr{V}(ext_G) = ext_H$.

In this work, we do not distinguish between isomorphic graphs.

Strings can be considered as graphs with the string structure. This is formalized in

Definition 7. *A string graph* induced by a string $w = a_1 \dots a_n$ is a graph of the form $\langle \{v_i\}_{i=0}^n, \{e_i\}_{i=1}^n, att, lab, v_0 v_n \rangle$ where $att(e_i) = v_{i-1} v_i$, $lab(e_i) = a_i$. It is denoted by $w^\bullet$.

We additionally introduce the following definition (not from [3]):

Definition 8. Let $H \in \mathscr{H}(C)$ be a graph, and let $f : E_H \to C$ be a function. Then $f(H) = \langle V_H, E_H, att_H, lab_{f(H)}, ext_H \rangle$ where $lab_{f(H)}(e) = f(e)$ for all e in E_H. It is required that $rank(lab_H(e)) = rank(f(e))$ for $e \in E_H$.

If one wants to relabel only one edge e_0 within H with a label a, then the result is denoted by $H[e_0 := a]$.

Definition 9. *Hyperedge replacement* is defined in [3], and it plays a fundamental role in hyperedge replacement grammars. The replacement of an edge e_0 in G with a graph H can be done if $rank(e_0) = rank(H)$ as follows:

1. Remove e_0;
2. Insert an isomorphic copy of H (H and G have to consist of disjoint sets of nodes and edges);
3. For each i, fuse the i-th external node of H with the i-th attachement node of e_0.

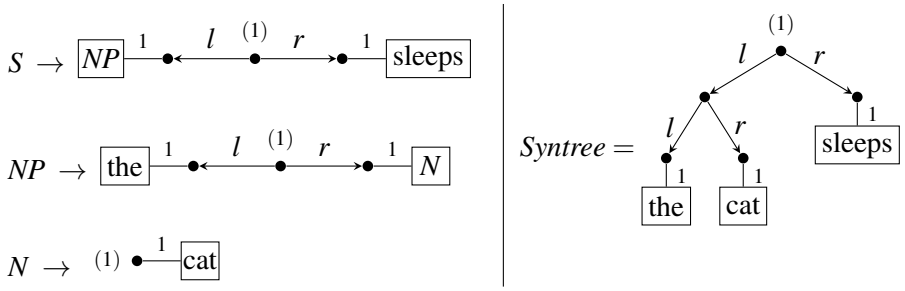
The result is denoted by $G[e_0/H]$.

It is known that if several edges of a graph are replaced by other graphs, then the result does not depend on the order of replacements; moreover the result is not changed if replacements are done simultaneously (see [3]). The following notation is in use: if $e_1, \dots, e_k$ are distinct edges of a graph H and they are simultaneously replaced by graphs $H_1, \dots, H_k$ resp. (this requires $rank(H_i) = rank(e_i)$), then the result is denoted $H[e_1/H_1, \dots, e_k/H_k]$.

3 Hypergraph Lambek Calculus

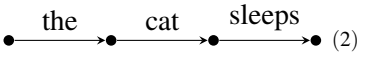
HRGs can be used to describe linguistic structures as well as context-free grammars since linguistic objects often have an underlying structure, which is more complex than a string. One of the recently studied applications is using HRGs for abstract meaning representation (see e.g. [2, 4, 6]): the meaning of a sentence is represented by a graph. In [1], HRGs are used for modelling nonprojective dependencies in Dutch. Another example where graph structures occur in linguistics is syntactic trees. Given a context-free grammar, it is natural to represent an internal hierarchical structure of constituents of a generated sentence by a tree.

Example 3. The HRG $HGr = \langle \{S, NP, N\}, \{\text{the}, \text{cat}, \text{sleeps}, l, r\}, P, S \rangle$ with the list of productions P defined below generates the graph *Syntree*:



Syntree is a simplified syntactic tree for the sentence *the cat sleeps*; l and r distinguish left and right directions in the tree.

Since there are such cases in linguistics where we need to work with graphs rather than with strings, we would like to generalize the categorial point of view discussed in Section 1 to hypergraphs. Our first attempt was a generalization of basic categorial grammars to hypergraphs; the resulting formalism called *hypergraph basic categorial grammar* was introduced at ICGT 2020 [10]. However, this formalism did not significantly improve our insight since most results were proved in the same way as for strings; in particular, such grammars generate the same set of languages as HRGs (with some nonsubstantial exceptions related to the number of isolated nodes). In this paper, we aim to go further and to discuss how the Lambek calculus along with its grammars can be generalized to hypergraphs. Note that, in the string case, Lambek grammars (i.e. grammars based on the Lambek calculus) are equivalent to context-free grammars and to basic categorial grammars (this nontrivial result was proved in [8]).

It is known that strings can be represented as string graphs, e.g. 

represents the string *the cat sleeps* (this graph is denoted as $(\text{the cat sleeps})^\bullet$). The production $S \to NP$ sleeps then is transformed into the graph production $S \to (NP \text{ sleeps})^\bullet$. We want to transform this production into a correspondence as it was done in Section 1; there we "took out" the terminal unit *sleeps* from the right-hand side, and assigned the type $NP\backslash S$ to it. Now, we are going to do the same operation but we shall mark an edge, from which we took out the label *sleeps*, by a special \$ symbol:

$$S \to (NP \text{ sleeps})^\bullet \quad \rightsquigarrow \quad \text{sleeps} \triangleright S/(NP\ \$)^\bullet$$

In general, \$ denotes the hyperedge, from which we took out its label. Similarly, the production $[NP \to \text{the } N]$ is transformed as follows: $\text{the} \triangleright NP/(\$\ N)^\bullet$. Note that now we do not need two divisions $\backslash$ and $/$ anymore since the difference between them is now expressed by the position of the \$-labeled edge. In order to distinguish the string divisions and the new graph division, and also to stress that the latter is undirected, we write $A \div D$ instead of A/D for the latter.

The conversion of a production into a correspondence between a terminal unit (a symbol, a label, a word) and a type requires that there is exactly one terminal unit in the right-hand side of the production. This property is called lexicalized normal form, or weak Greibach normal form. Note that, if we have e.g. a production $S \to (ABaCD)^\bullet$ where S,A,B,C,D are nonterminal, and a is terminal, then we can also use the above \$-notation and write $a \triangleright S \div (AB\$CD)^\bullet$. As in Section 1, this means that a is such an object that whenever objects of types A, B, C and D are placed instead of corresponding edges in the graph $(AB\$CD)^\bullet$, and a is placed on the \$-labeled edge, the resulting structure forms an object of the type S. We can proceed similarly with an arbitrary hypergraph production, if the grammar is in the weak Greibach normal form. This is the main idea of hypergraph basic categorial grammars. However, as in the string case, we would like to go further and to consider more complex types; besides, we would also like to generalize the operation $A \cdot B$. This results in the following

Definition 10. The set $Tp(\mathrm{HL})$ of types of the hypergraph Lambek calculus HL is defined inductively as the least set satisfying the following conditions:

1. Primitive types Pr are types. Pr is a countable set along with the function $rank : Pr \to \mathbb{N}$ such that for each n there are infinitely many types $p \in Pr$ such that $rank(p) = n$.
2. Let N ("numerator") be a type. Let D ("denominator") be a hypergraph such that exactly one of its hyperedges (call it e_0) is labeled by \$, and the other hyperedges (possibly, there are none of them) are labeled by elements of $Tp(\mathrm{HL})$; let also $rank(N) = rank(D)$. Then $T = (N \div D)$ is a type, and $rank(T) := rank_D(e_0)$.
3. Let M be a hypergraph such that all its hyperedges are labeled by types from $Tp(\mathrm{HL})$ (possibly, there are no hyperedges at all). Then $T = \times(M)$ is also a type, and $rank(T) := rank(M)$.

Below we often write $N \div (D)$ or $N \div D$ instead of $(N \div D)$. Paragraph 2 generalizes the concept of $\div$ explained earlier: $N \div D$ is the type of such hypergraphs H that, if we replace the \$-labeled edge in D by H and for all the remaining edges $e_i, i > 0$, which are labeled by some types T_i, we replace them by hypergraphs H_i, which are of types T_i, then we obtain a hypergraph of the type N. In particular, this explains why we require $rank(N) = rank(D)$ and $rank(N \div D) = rank_D(e_0)$.

Example 4. The first production from Example 3 can be transformed into the following correspondence:

$$\text{sleeps} \triangleright s \div \left(\boxed{np} \xrightarrow{1} \bullet \xleftarrow{p_l} \bullet^{(1)} \xrightarrow{p_r} \bullet \xrightarrow{1} \boxed{\$} \right) \tag{1}$$

Here p_l, p_r are primitive types introduced to deal with special "technical" labels l (r resp.). According to our general understanding of types, one may say that l (r) is of the type p_l (p_r resp.). Now, (1) means that *sleeps* is such an object that, if we place it instead of \$ within the graph $\boxed{np} \xrightarrow{1} \bullet \xleftarrow{p_l} \bullet^{(1)} \xrightarrow{p_r} \bullet \xrightarrow{1} \boxed{\$}$, and we place any object (syntactic tree) of the type *np* instead of the *np*-labeled edge, then we obtain an object of the type s (we also need to replace p_l by l and p_r by r).

The operation $\times(M)$ can be called a hypergraph product, or a hypergraph concatenation. Its general semantics it the following: if $E_M = \{m_1, \ldots, m_l\}$, $lab_M(m_i) = T_i$, and H_i is a hypergraph of the type T_i ($i = 1, \ldots, l$), then the hypergraph $M[H_1/m_1, \ldots, H_l/m_l]$ is a hypergraph of the type $\times(M)$. Thus, $\times(M)$ is the type of all substitution instances of M.

Example 5. If str is the primitive type of all string graphs labeled by the blank symbol $*$, then the type $\times \left((1) \bullet \underset{str}{\overset{str}{\rightrightarrows}} \bullet (2) \right)$ is the type of all graphs consisting of two parallel strings with the common start and finish nodes, e.g. $(1) \bullet \rightrightarrows \bullet (2)$.

Moving away from the intuition of $\div$ and $\times$, we would like to introduce a syntactic *calculus*, which would work with types introduced in Definition 10 by means of axioms and rules. We expect that this calculus should generalize the Lambek calculus in the Gentzen style introduced in Section 1. This is done in our preprint [12]; in this paper, we only introduce the axiom and rules of the hypergraph Lambek calculus (HL) without a detailed discussion of why they actually generalize those of L.

Definition 11. *A hypergraph sequent* is a structure of the form $H \to A$, where $A \in Tp(\mathrm{HL})$ is a type, $H \in \mathscr{H}(Tp(\mathrm{HL}))$ is a hypergraph labeled by types and $rank(H) = rank(A)$. H is called the *antecedent* of the sequent, and A is called the *succedent* of the sequent.

Remark 1. Returning to our intuition, $H \to A$ could be understood as the statement "each hypergraph of the type $\times(H)$ is also of the type A".

The hypergraph Lambek calculus HL deals with hypergraph sequents and explains, which of them are *derivable* using axioms and rules. The only **axiom** of HL is the following: $p^\bullet \to p, \quad p \in Pr$ ($p^\bullet$ here is the p-handle). Rules are presented below along with some simple examples.

1. **Rule** $(\div \to)$**.** Let $N \div D$ be a type and let $E_D = \{d_0, d_1, \ldots, d_k\}$ where $lab(d_0) = \$$, $lab(d_i) = T_i$ for $i \geq 1$. Let $H \to A$ be a hypergraph sequent and let $e \in E_H$ be labeled by N. Let finally $H_1, \ldots, H_k$ be hypergraphs labeled by types. Then the rule $(\div \to)$ is the following:

$$\frac{H \to A \quad H_1 \to T_1 \quad \ldots \quad H_k \to T_k}{H[e/D][d_0 := N \div D][d_1/H_1, \ldots, d_k/H_k] \to A} \ (\div \to)$$

This rule explains how a type with division may appear in the antecedent of a sequent: we replace a hyperedge e by D, put a label $N \div D$ instead of \$ and replace the remaining labels of D by corresponding antecedents.

Example 6. Consider the following rule application with T_i being some types and with T being equal to $q \div (T_2 \$ T_3)^\bullet$ ($w^\bullet$ here denotes a string graph induced by w):

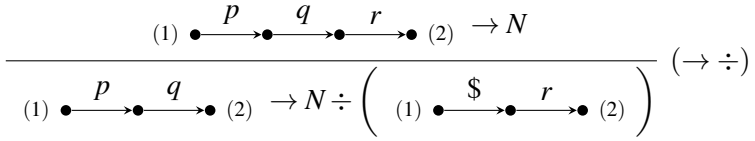
$$\frac{(pq)^\bullet \to T_1 \quad (rs)^\bullet \to T_2 \quad (tu)^\bullet \to T_3}{(prsTtu)^\bullet \to T_1} \ (\div \to)$$

2. **Rule** $(\to \div)$**.** Let $F \to N \div D$ be a hypergraph sequent; let $e_0 \in E_D$ be labeled by \$. Then

$$\frac{D[e_0/F] \to N}{F \to N \div D} \ (\to \div)$$

This rule is understood as follows: if there are such hypergraphs D, F and such a type N that in a sequent $H \to N$ the hypergraph H equals $D[e_0/F]$ and $H \to N$ is derivable, then $F \to N \div D$ is also derivable.

Example 7. Consider the following rule application where N equals $\times((pqr)^\bullet)$ (here we draw string graphs instead of writing $w^\bullet$ to visualize the rule application):

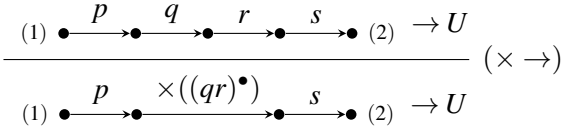


3. **Rule** $(\times \to)$**.** Let $G \to A$ be a hypergraph sequent and let $e \in E_G$ be labeled by $\times(F)$. Then

$$\frac{G[e/F] \to A}{G \to A} \ (\times \to)$$

This rule is formulated from bottom to top as the previous one. Intuitively speaking, there is a subgraph of an antecedent in a premise, and it is "compressed" into a single $\times(F)$-labeled hyperedge.

Example 8. Consider the following rule application where U equals $\times((pqrs)^\bullet)$:



4. **Rule** $(\to \times)$**.** Let $\times(M)$ be a type and let $E_M = \{m_1, \ldots, m_l\}$. Let $H_1, \ldots, H_l$ be graphs. Then

$$\frac{H_1 \to lab(m_1) \quad \ldots \quad H_l \to lab(m_l)}{M[m_1/H_1, \ldots, m_l/H_l] \to \times(M)} \ (\to \times)$$

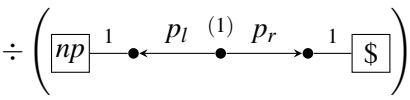
This rule is quite intuitive: several sequents can be combined into a single one via some hypergraph structure M.

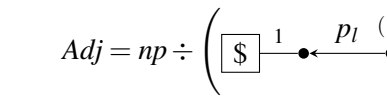
Example 9. Consider the following rule application with T_i being some types:

$$\frac{(pq)^\bullet \to T_1 \quad (rs)^\bullet \to T_2 \quad (tu)^\bullet \to T_3}{(pqrstu)^\bullet \to \times((T_1T_2T_3)^\bullet)} \ (\to \times)$$

Definition 12. A hypergraph sequent $H \to A$ is *derivable in* HL (HL $\vdash H \to A$) if it can be obtained from axioms using rules of HL. A corresponding sequence of rule applications is called *a derivation* and its representation as a tree is called *a derivation tree*.

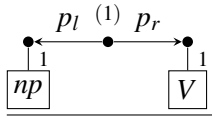
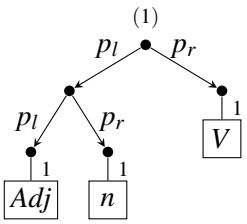
Example 10. Let $rank(s) = rank(np) = rank(n) = 1$, $rank(p_l) = rank(p_r) = 2$, and let

$V = s \div$ 

; $Adj = np \div$ 

.

Then the following is the derivation of the below sequent:

$$\frac{s^\bullet \to s \quad np^\bullet \to np \quad p_l^\bullet \to p_l \quad p_r^\bullet \to p_r}{\begin{array}{l} \text{  } \\ \to s \end{array} \qquad \begin{array}{l} n^\bullet \to n \quad p_l^\bullet \to p_l \quad p_r^\bullet \to p_r \\ \text{  } \\ \to \quad s \end{array}} \ (\div \to) \ (\div \to)$$

In [12], we show that L and its different variants can be embedded in HL; we also show that certain structural properties of L can be straightforwardly lifted to HL. Hence HL can be considered as an appropriate extension of the Lambek calculus to hypergraphs, as desired. Note that introduction of the division $\div$ and of the product $\times$ was motivated by the intuitive understanding of types as of families of hypergraphs (i.e. hypergraph languages). Although HL was defined as a purely syntactic formalism that formally explains how hypergraph sequents can be rewritten, one would expect that hypergraph languages can be considered as models of HL. In Section 5, we formally define language models (L-models in short) for HL in a way similar to how we introduced the notion of valuation $\overline{w}$ in Section 1. We establish correctness of HL with respect to L-models, and completeness of its $\times$-free fragment.

The following statements can be proved in a similar way as for strings:

Theorem 1 (cut elimination)**.** *If $H \to A$ and $G \to B$ are derivable in* HL*, and $e_0 \in E_G$ is labeled by A, then $G[e_0/H] \to B$ is also derivable in* HL*.*

Proposition 1 (reversibility of $(\times \to)$ and $(\to \div)$)**.**

1. *If* $\mathrm{HL} \vdash H \to C$ *and* $e_0 \in E_H$ *is labeled by* $\times(M)$*, then* $\mathrm{HL} \vdash H[e_0/M] \to C$.
2. *If* $\mathrm{HL} \vdash H \to N \div D$ *and* $e_0 \in E_D$ *is labeled by* \$*, then* $\mathrm{HL} \vdash D[e_0/H] \to N$.

These statements will be used in proofs of some results in this paper.

Although in this paper we devote a great deal of attention to the hypergraph Lambek calculus itself and to its model-theoretic motivation, our main goal is to study the concept of *hypergraph Lambek grammars* (HL-grammars in short). They are defined in a similar way to Lambek grammars (in the string case). A grammar is essentially a finite set of correspondences of the form $a \triangleright T$ where a is a terminal label, and T is a type; besides, in a grammar some type S (not necessarily primitive) is distinguished. Then a hypergraph G belongs to the language generated by the grammar if we can replace labels of its hyperedges by corresponding types (let us denote the resulting graph G') and derive the sequent $G' \to S$ in HL. The formal definition of hypergraph Lambek grammars will be given in the next section.

4 Hypergraph Lambek Grammars and Their Power

Our goal in this section is to generalize the definition of Lambek grammars to hypergraphs. We consider an alphabet Σ along with the function $rank : \Sigma \to \mathbb{N}$.

Definition 13. A *hypergraph Lambek grammar (HL-grammar)* is a tuple $HGr = \langle \Sigma, S, \triangleright \rangle$ where Σ is a finite set (alphabet), $S \in Tp(\mathrm{HL})$ is a distinguished type, and $\triangleright \subseteq \Sigma \times Tp(\mathrm{HL})$ is a finite binary relation such that $a \triangleright T$ implies $rank(a) = rank(T)$.

Definition 14. *The type set* of an HL-grammar $HGr = \langle \Sigma, S, \triangleright \rangle$ is the set $\mathrm{ts}(HGr) = \{T \mid \exists a \in \Sigma : a \triangleright T\}$.

Definition 15. *The language* $L(HGr)$ *generated by a hypergraph Lambek grammar* $HGr = \langle \Sigma, S, \triangleright \rangle$ is the set of all hypergraphs $G \in \mathscr{H}(\Sigma)$ for which a function $f_G : E_G \to Tp(\mathrm{HL})$ exists such that:

1. $lab_G(e) \triangleright f_G(e)$ whenever $e \in E_G$;
2. $\mathrm{HL} \vdash f_G(G) \to S$.

Example 11. The HL-grammar $SGr = \langle \{a, b\}, s, \triangleright \rangle$ where s is primitive, and

- $a \triangleright s \div (\$sp)^\bullet = Q$,
- $b \triangleright p$, $b \triangleright s$

generates the language $\{(a^n b^{n+1})^\bullet \mid n \geq 0\}$. For example, if one wants to check that $G = (aabbb)^\bullet$ belongs to $L(SGr)$, he/she follows such steps:

1. Relabel each edge in G in such a way that each label is replaced by a type corresponding to it. We do this as follows: $G = (aabbb)^\bullet \rightsquigarrow f_G(G) = (QQspp)^\bullet$.
2. Consider the sequent $f_G(G) \to s$ and derive it in HL:

$$\dfrac{\dfrac{s^\bullet \to s \quad s^\bullet \to s \quad p^\bullet \to p}{(Qsp)^\bullet \to s}\,(\div \to) \qquad s^\bullet \to s \quad p^\bullet \to p}{(QQspp)^\bullet \to s}\,(\div \to)$$

Now, notice the following: if a sequent $G' \to s$ is derivable, and G' is labeled only by types from the type set of SGr, then each derivation of $G' \to s$ consists only of applications of $(\div \to)$. The rule $(\div \to)$

consists of several replacements in the antecedent of a sequent, and hence the grammar *SGr* works in a way similar to the hyperedge replacement grammar with the following set of productions:

$$S \to (aSP)^{\bullet} \qquad S \to b^{\bullet} \qquad P \to b^{\bullet}$$

Note that the conversion of this HRG back into *SGr* can be made according to the principles explained in Section 3. The new grammar is actually a graph variant of a context-free grammar with the productions $S \to aSP, S \to b, P \to b$, which, clearly, generates the language $\{a^n b^{n+1} \mid n \geq 0\}$.

The transformation considered in Example 4 and in Example 11 is possible, if there is exactly one terminal label (say a) in a production; then we place \$ instead of a, and establish a correspondence $\triangleright$ between a and a type made on the basis of this production.

Definition 16. An HRG *HGr* is in *the weak Greibach normal form* if there is exactly one terminal edge in the right-hand side of each production.

Denote by $isize(H)$ the number of isolated nodes in H.

Definition 17. A hypergraph language L is isolated-node bounded if there is a constant $M > 0$ such that for each $H \in L$ $isize(H) < M \cdot |E_H|$.

In [11] we prove the following

Theorem 2. *For each HRG generating an isolated-node bounded language there is an equivalent HRG in the weak Greibach normal form.*

Using it, we can prove the following theorem applying standard techniques.

Theorem 3. *For each HRG generating an isolated-node bounded language there is an equivalent HL-grammar.*

The proof of this theorem is placed in Appendix A. It is not, however, of interest in this paper; we formulate this theorem here only to show the reader that HL-grammars are not weaker than HRGs (isolated-node boundedness is a nonsubstantial limitation). Our objective now is to show that HL-grammars are more powerful than HRGs; to do this, we will introduce several examples of grammars generating languages that can be generated by no HRGs.

4.1 All Binary Graphs

One of restrictions known for languages generated by HRGs is that they are of bounded connectivity (see [3]); this follows from the pumping lemma (see [5], Chapter IV.2). Consequently, no HRG can generate the set of all binary graphs (i.e. of usual graphs with edges of rank 2). This might seem unnatural because HRGs represent a context-free formalism, and the language of all binary graphs seems to be very simple and regular. Below we show that HL-grammars are powerful enough to generate such a language.

Consider the language $\mathscr{L}_1$ of all binary graphs without isolated nodes (the empty graph is not included in $\mathscr{L}_1$ as well) over the alphabet $\{*\}$ ($rank(*) = 2$) that are, besides, without external nodes. Consequently, each graph in this language has at least two nodes. Let s, p be primitive types ($rank(s) = 0$, $rank(p) = 1$). Let us define the following types:

$$Q_1 = p, \quad Q_2 = p \div \left(\begin{array}{cc} \boxed{\$} & \boxed{p} \\ 1 \big| & \big| 1 \\ \bullet & \bullet \\ (1) & \end{array} \right), \quad Q_3 = s \div \left(\begin{array}{cc} \boxed{\$} & \boxed{p} \\ 1 \big| & \big| 1 \\ \bullet & \bullet \end{array} \right);$$

$$M_{11}^{ij} = \times\left(\begin{array}{cc} \boxed{Q_i} & \boxed{Q_j} \\ 1\,|\,\bullet & \bullet\,|\,1 \\ (1) & (2) \end{array}\right), M_{12}^{i} = \times\left(\begin{array}{cc} \boxed{Q_i} & \\ 1\,|\,\bullet & \bullet \\ (1) & (2) \end{array}\right), M_{21}^{j} = \times\left(\begin{array}{cc} & \boxed{Q_j} \\ \bullet & \bullet\,|\,1 \\ (1) & (2) \end{array}\right), M_{22} = \times\left(\begin{array}{cc} \bullet & \bullet \\ (1) & (2) \end{array}\right).$$

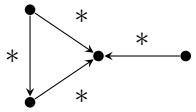
Consider a grammar $HGr_1 = \langle\{*\}, s, \triangleright\rangle$ where $* \triangleright N$ whenever $N \in \{M_{11}^{ij}, M_{12}^{i}, M_{21}^{j}, M_{22} | 1 \le i, j \le 3\}$.

Theorem 4. $L(HGr_1) = \mathscr{L}_1$.

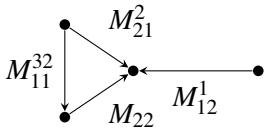
Proof. To prove that $L(HGr_1) \subseteq \mathscr{L}_1$ it suffices to note that denominators of types in $\mathrm{ts}(HGr_1)$ do not contain isolated nodes; since isolated nodes may appear only after applications of rules $(\div \to)$ or $(\to \times)$, all graphs in $L(HGr_1)$ do not have them.

The other inclusion $L(HGr_1) \supseteq \mathscr{L}_1$ is of central interest. We start with an example of a specific derivation in this grammar. After, we provide the proof in a general case, but we suppose that this example is enough to understand the construction of HGr_1.

Example 12. Consider a binary graph

$H =$ 

In order to check that H belongs to $L(HGr_1)$ we relabel it by corresponding types as follows:

$f_H(H) =$ 

Then we check derivability of $f_H(H) \to s$ (see Figure 1).

In general, let H be in $\mathscr{L}_1$. Since there are no isolated nodes in H there exists a function $h : V_H \to E_H$ such that $h(v)$ is attached to v whenever $v \in V_H$. We choose two arbitrary nodes v_b (begin) and v_e (end) such that $v_b \neq v_e$. After that, we define a function $c : V_H \to \{1,2,3\}$ as follows: $c(v_b) = 1$, $c(v_e) = 3$, and $c(v) = 2$ whenever $v \notin \{v_b, v_e\}$.

Now we present a relabeling $f_H : E_H \to Tp(\mathrm{HL})$. Let e belong to E_H and let $att_H(e) = v_1 v_2$.

- If $h(v_1) = h(v_2) = e$, then $f_H(e) := M_{11}^{c(v_1)c(v_2)}$;
- If $h(v_1) \neq e, h(v_2) = e$, then $f_H(e) := M_{21}^{c(v_2)}$;
- If $h(v_1) = e, h(v_2) \neq e$, then $f_H(e) := M_{12}^{c(v_1)}$;
- If $h(v_1) \neq e, h(v_2) \neq e$, then $f_H(e) := M_{22}$.

We aim to check derivability of the sequent $f_H(H) \to s$. Its derivation from bottom to top starts with the rule $(\times \to)$ applied $|E_H|$ times to all types in the antecedent. It turns out that, after such applications of $(\times \to)$, the antecedent of a sequent includes one edge labeled by Q_1, one edge labeled by Q_3, and the remaining edges labeled by Q_2; besides, for each node there is exactly one edge attached to it (this is satisfied by the definition of the function h). Then we apply (again from bottom to top) the rule $(\div \to)$, and using it we "reduce" the only Q_1-labeled edge (recall that $Q_1 = p$) with a Q_2-labeled edge; after this we obtain a new p-labeled edge and repeat the procedure. Thus we eliminate all nodes and edges one-by-one. Finally, we obtain a graph with two nodes, with a Q_3-labeled edge attached to the first one and a p-labeled edge attached to the second one. Applying $(\div \to)$ once more, we "contract" Q_3 with p and obtain the sequent $s^\bullet \to s$, which is an axiom. □

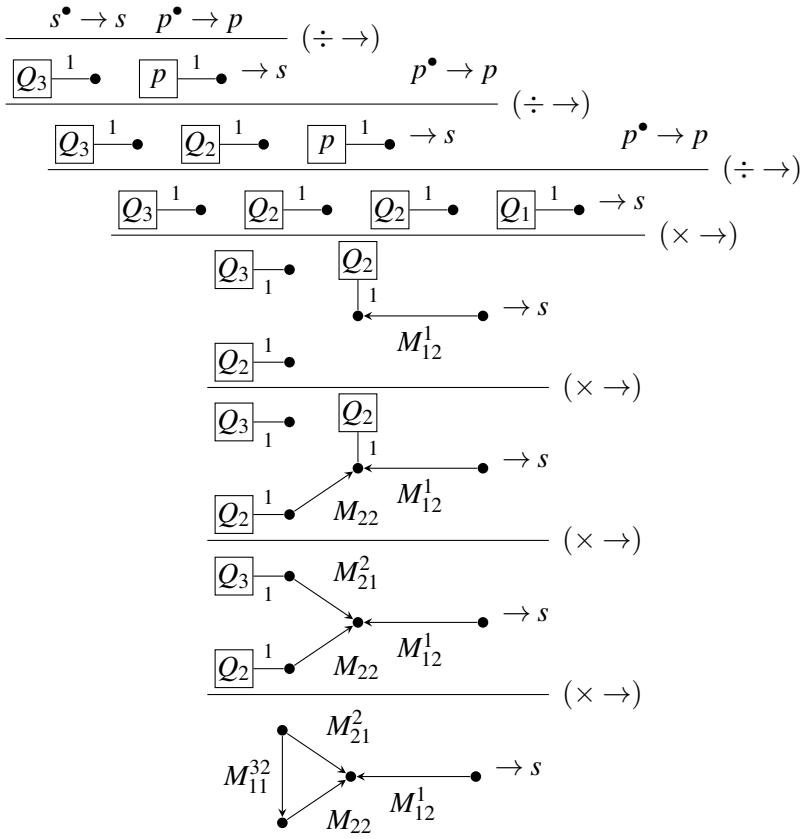


Figure 1: Derivation of a sequent corresponding to a binary graph.

Therefore, we have established that HL-grammars are stronger than HRGs and that they moreover disobey the pumping lemma.

4.2 Bipartite graphs

Another example is the language $\mathscr{L}_2 \subseteq \mathscr{L}_1$ of all bipartite binary graphs without isolated nodes.

Definition 18. A binary graph H is *bipartite* if its nodes can be divided into two disjoint subsets V_1 and V_2 in such a way that each edge of H outgoes from a node belonging to V_1 to a node belonging to V_2.

Let us define the following types (where p, q are primitive, $rank(p) = rank(q) = 1$):

- $R_1(r) := r$;
- $R_2(r) := r \div \left(\begin{array}{c} \boxed{\$} \quad \boxed{r} \\ 1 \quad\; 1 \\ \quad (1) \end{array} \right)$;
- $R_3(r) := r \div \left(\begin{array}{cc} \boxed{\$} & \boxed{r} \\ 1 & 1 \\ (1) & \end{array} \right)$;
- $R_4(r) := r \div \left(\begin{array}{ccc} \boxed{\$} & \boxed{r} & \boxed{r} \\ 1 & 1 & 1 \\ & (1) & \end{array} \right)$;
- $M^{ij} := \times \left(\begin{array}{cc} \boxed{R_i(p)} & \boxed{R_j(q)} \\ 1 & 1 \\ (1) & (2) \end{array} \right)$, $1 \le i, j \le 4$;
- $S := \times \left(\begin{array}{cc} \boxed{p} & \boxed{q} \\ 1 & 1 \end{array} \right)$.

We define $HGr_2 := \langle \{*\}, S, \triangleright \rangle$ as follows: $* \triangleright M^{ij}$ for all $1 \le i, j \le 4$.

Proposition 2. $\mathscr{L}_2 = L(HGr_2)$.

The proof of this proposition is given in Appendix B. It is not hard to straightforwardly show that $\mathscr{L}_2 \subseteq L(HGr_2)$ by deriving sequents corresponding to graphs from $\mathscr{L}_2$. To prove the other inclusion, we use Proposition 1 and then notice that there is no way for two hyperedges, one of which is labeled by $R_i(p)$ and the other one is labeled by $R_j(q)$, to be attached to the same node in the antecedent (to prove this, it suffices to analyze variants of how the rule $(\div \to)$ can be applied).

4.3 Regular graphs

A less trivial example of a hypergraph language generated by a HL-grammar and by no HRGs is the language of regular binary graphs.

Definition 19. A binary graph H is *regular* if there is an integer $k \ge 1$ such that the indegree and the outdegree of each node equals k.

Let $\mathscr{L}_3 \subseteq \mathscr{L}_1$ be the language of all regular binary graphs (without the empty graph).

Theorem 5. *$\mathscr{L}_3$ can be generated by some HL-grammar.*

To prove this theorem, we need the following result proved in the paper accepted for ICGT 2021 (we place this proof in Appendix C):

Theorem 6. *If $L_1, \ldots, L_k$ are languages generated by some HRGs, then $L_1 \cap \ldots \cap L_k$ can be generated by some HL-grammar.*

Less formally, this means that HL-grammars can generate finite intersections of languages generated by HRGs.

Definition 20. Let $MS_1,\dots,MS_n$, $n \geq 1$ be some multisets with elements from $C' \subseteq C$ (that is, they are multisets of labels from C'). Let $b \in C \setminus C'$ be some symbol with $rank(b) = 2$. We denote $MS_i = \{a_i^1,\dots,a_i^{k_i}\}$, and $rank(a_i^j) = t_i^j$. *A flowerbed* $\mathfrak{F}(MS_1,\dots,MS_n,b)$ over C' is the hypergraph $\langle V,E,att,lab,ext\rangle$ where

1. $V = \{v_i^{jk} \mid i = 1,\dots,n,\ j = 1,\dots,k_i,\ k = 1,\dots,t_i^j - 1\} \cup \{u_1,\dots,u_n\}$;
2. $E = \{e_i^j \mid i = 1,\dots,n,\ j = 1,\dots,k_i\} \cup \{f_1,\dots,f_{n-1}\}$;
3. (a) $att(e_i^j) = u_i v_i^{j1} v_i^{j2} \dots v_i^{j(t_i^j-1)}$ (if $t_i^j = 1$, then $att(e_i^j) = u_i$);
   (b) $att(f_k) = v_k v_{k+1}$;
4. (a) $lab(e_i^j) = a_i^j$;
   (b) $lab(f_k) = b$;
5. $ext = \Lambda$.

Definition 21. Given a multiset MS, $|MS|^a$ denotes the number of occurences of a in MS.

Proof (of Theorem 5). Let $C' = \{a,z\}$ with $rank(a) = 1$, $rank(z) = 3$. We set $\Sigma = \{a,z,b\}$ with $rank(b) = 2$. Let us introduce the following languages:

- L_1 is the set of all flowerbeds over C' of the form $\mathfrak{F}(MS_1,\dots,MS_n,b)$ such that $|MS_{2k}|^a = |MS_{2k+1}|^a = |MS_{2k}|^z = |MS_{2k+1}|^z$ for $k = 1,2,\dots$
- L_2 is the set of all flowerbeds over C' of the form $\mathfrak{F}(MS_1,\dots,MS_n,b)$ such that $|MS_{2k-1}|^a = |MS_{2k}|^a = |MS_{2k-1}|^z = |MS_{2k}|^z$ for $k = 1,2,\dots$

L_1 and L_2 can be generated by some HRGs (see Appendix D); thus, according to Theorem 6, $L = L_1 \cap L_2$ can be generated by an HL-grammar. Let us denote such a grammar $HGr = \langle \Sigma,S,\triangleright\rangle$: $L(HGr) = L$. Note that L is the set of all flowerbeds over C' of the form $\mathfrak{F}(MS_1,\dots,MS_n,b)$ such that $|MS_k|^a = |MS_{k+1}|^a = |MS_k|^z = |MS_{k+1}|^z$ for $k = 1,2,\dots,n-1$.

Let us denote all types corresponding to a via $\triangleright$ as A_i (i.e. $a \triangleright A_i$), all types corresponding to z as Z_j, and all types corresponding to b as B_k. Let

$$T_{ij} := \times\left(\begin{array}{cc} \boxed{A_i} & \boxed{Z_j} \\ 1 & 1 \\ \bullet & \bullet \\ (1) & (2) \end{array}\right), \qquad T_{ijk} := \times\left(\begin{array}{ccc} \boxed{A_i} & & \boxed{Z_j} \\ 1 & B_k & 1 \\ \bullet & \longrightarrow & \bullet \\ (1) & & (2) \end{array}\right).$$

Using these types we define an HL-grammar $\widetilde{HGr} := \langle\{*\},S,\widetilde{\triangleright}\rangle$ as follows: $* \mathrel{\widetilde{\triangleright}} T_{ij}, T_{ijk}$ for all possible i, j, k. We argue that $L(\widetilde{HGr}) = L_3$. Indeed, $H \in L(\widetilde{HGr})$ if and only if there exists a relabeling f_H such that $lab_H(e) \mathrel{\widetilde{\triangleright}} f_H(e)$, and $\mathrm{HL} \vdash f_H(H) \to S$. $f_H(H)$ consists of types T_{ij} and T_{ijk}, which are of the form $\times(M)$. Using Proposition 1, we draw the conclusion that $\mathrm{HL} \vdash f_H(H) \to S$ if and only if $\mathrm{HL} \vdash \hat{H} \to S$ where $\hat{H}$ is obtained from $f_H(H)$ by replacing each hyperedge labeled by a type of the form $\times(M)$ by M. $\hat{H}$ is labeled by types A_i, Z_j, and B_k. Note that for all i,j,k $A_i \neq Z_j$, $Z_j \neq B_k$, $A_i \neq B_k$ since $type(A_i) = 1$, $type(B_k) = 2$, $type(Z_j) = 3$. Let $g : E_{\hat{H}} \to \Sigma$ be such a function that $g(e) = a$ if $lab_{\hat{H}}(e) = A_i$, $g(e) = b$ if $lab_{\hat{H}}(e) = B_k$, and $g(e) = z$ if $lab_{\hat{H}}(e) = Z_j$. Since $\mathrm{HL} \vdash \hat{H} \to S$, $g(\hat{H})$ belongs to $L(HGr) = L$. To complete the proof, observe that the number of a-labeled edges attached to a node in $g(\hat{H})$ equals the outdegree of this node in H, and the number of z-labeled edges attached to a node in $g(\hat{H})$ equals the indegree of this node in H; according to the definition of L, this number is the same for all nodes.

Formally, in the above reasonings we made a one-way transition when we introduced g; hence, we only proved that $\widetilde{HGr}$ generates regular binary graphs. However, given a regular binary graph H, we can construct graphs of the form $f_H(H)$, $\hat{H}$, and $g(\hat{H})$ corresponding to it and then repeat the above reasonings. □

Remark 2. Consider the language L_3 of all flowerbeds over C' of the form $\mathfrak{F}(MS_1,\dots,MS_n,b)$ such that $|MS_1|^a = n-1$. This language can also be generated by some HRG (this is left as an exercise to the reader). If we defined L in the above proof as $L_1 \cap L_2 \cap L_3$, then $L(\widetilde{HGr})$ would consist of all regular binary graphs with n nodes such that the indegree and the outdegree of each node equals $n-1$. Note that numbers of edges in graphs of $L(\widetilde{HGr})$ in such a case form the set $\left\{\frac{n(n-1)}{2} \mid n \geq 2\right\}$, which grows with the pace $O(n^2)$ (this violates the Linear-Growth theorem, see [5], Chapter IV.2).

5 Language Models for HL

Let us return to the model-theoretic point of view discussed in Sections 1 and 3. There we looked on types of the Lambek calculus (either in its string or in its hypergraph versions) as hypergraph languages; divisions and product were interpreted as operations on languages. In this section, we are going to formalize this idea for HL in a way similar to that in Definition 1 and in Examples 4 and 5.

Definition 22. Given an alphabet Σ along with the function $rank : \Sigma \to \mathbb{N}$, we call a function $w : Pr \to \mathscr{P}(\mathscr{H}(\Sigma))$ *a valuation* if for each $p \in Pr$ there exists $n \in \mathbb{N}$ such that $rank(H) = n$ whenever $H \in w(p)$. This function assigns a hypergraph language to each primitive type. Its extension $\overline{w}$ is the following function from the set of hypergraph types to $\mathscr{P}(\mathscr{H}(\Sigma))$:

1. Let $N \div D$ be a type and let $E_D = \{d_0,\dots,d_k\}$, $lab_D(d_0) = \$$, $lab_D(d_i) = T_i$. Then $\overline{w}(N \div D)$ consists of all graphs G such that $D[d_0/G, d_1/H_1, \dots, d_k/H_k]$ belongs to $\overline{w}(N)$ whenever $H_1 \in \overline{w}(T_1)$, ..., $H_k \in \overline{w}(T_k)$.
2. Let $\times(M)$ be a type and let $E_M = \{m_1,\dots,m_l\}$, $lab_M(m_i) = T_i$. $\overline{w}(\times(M))$ consists of all graphs of the form $M[m_1/H_1,\dots,m_l/H_l]$ for all $H_i \in \overline{w}(T_i)$.
3. We additionally define $\overline{w}(H \to A)$ as the statement $\overline{w}(\times(H)) \subseteq \overline{w}(A)$.

Thus we defined models for the hypergraph Lambek calculus. Now we formulate some standard model-theoretic results (their proof in the hypergraph case does not differ from that in the string case).

Theorem 7. *If* $\mathrm{HL} \vdash H \to A$*, then* $\overline{w}(H \to A)$ *is true for each valuation* w*.*

This theorem is proved by a straightforward induction on length of a derivation. The other direction (i.e. completeness) is an open question (in the string case, this direction was a hard open problem until it was proved in [9]). We expect that it holds in the hypergraph case but we have no idea how to generalize the proof from [9]. However, if we consider the product-free fragment of HL (that is, we will consider types with $\div$ only), then the completeness theorem can be easily proved using the canonical model.

Theorem 8. *If* $\overline{w}(H \to A)$ *is true for each valuation* w*, and types in* $H \to A$ *do not contain* $\times$*, then* $\mathrm{HL} \vdash H \to A$*.*

Proof. Let us denote the fragment of HL, in which we consider only types without $\times$, as $\mathrm{HL}(\div)$. We fix the alphabet $\Sigma = Tp(\mathrm{HL}(\div))$ (i.e. types without $\times$ are now symbols of the alphabet) and introduce a valuation w_0 for all primitive types: $w_0(p) = \{G \in \mathscr{H}(\Sigma) \mid \mathrm{HL} \vdash G \to p\}$. Note that such a definition of w_0 can be considered not only for primitive types p but for all types $T \in Tp(\mathrm{HL}(\div))$. We claim that $\overline{w_0}(T) = w_0(T)$ for all such types; that is, the function $\overline{w_0}$ obtained from Definition 22 coincides with w_0. Indeed: if $T = N \div D$, $lab_D(d_0) = \$$, and $lab_D(d_i) = T_i$ $(i = 1,\dots,k)$, then

$$G \in w_0(N \div D) \Leftrightarrow \mathrm{HL} \vdash G \to N \div D \Leftrightarrow \mathrm{HL} \vdash D[d_0/G] \to N \text{ (see Proposition 1)} \Leftrightarrow$$
$$\Leftrightarrow \forall H_i : (\mathrm{HL} \vdash H_i \to T_i,\ i = 1,\ldots,k) \quad \mathrm{HL} \vdash D[d_0/G, d_1/H_1, \ldots, d_k/H_k] \to N \Leftrightarrow G \in \overline{w_0}(N \div D).$$

The penultimate equivalence follows from the fact that $\mathrm{HL} \vdash T_i^\bullet \to T_i$ and from the cut elimination Theorem 1. Since $\overline{w_0}(H \to A)$ is true, $w_0(\times(H)) \subseteq w_0(A)$; H belongs to $w_0(\times(H))$ ($\mathrm{HL} \vdash H \to \times(H)$), hence H belongs to $w_0(A)$. By the definition of w_0, this yields that $\mathrm{HL} \vdash H \to A$. □

Unfortunately, a similar proof does not work for HL with $\times$ (like in the string case). Thus, there is much space for further investigations. Nevertheless, Theorem 7 and Theorem 8 partially justify the periphrastic name of the hypergraph Lambek calculus given in the title: it is a logic of hypergraph languages.

6 Conclusion

Hypergraph Lambek grammars is a logical formalism, which extends hyperedge replacement grammars. It deals with hypergraph types and sequents, which have a model-theoretic semantics of hypergraph languages. We showed that HL-grammars are more powerful than HRGs; in particular, they violate the pumping lemma and the Linear-Growth theorem.

Note that the membership problem for HL-grammars is NP-complete: this follows from the fact that, if H belongs to $L(HGr)$ for $HGr = \langle \Sigma, S, \triangleright \rangle$, then this can be certified by a function f_H and by a derivation of $f_H(H) \to S$. Description of f_H and the derivation have polynomial size with respect to H and HGr, hence the problem is in NP. It is NP-complete since HL-grammars can generate an NP-complete language (which can be generated by some HRG without isolated nodes). Therefore, being equal in complexity to HRGs, HL-grammars represent a promising instrument for generating hypergraph languages.

As is often the case, there are more questions than answers. Some of them are listed below:

1. Is it true that, if $\overline{w}(H \to A)$ is true for all valuations, then $\mathrm{HL} \vdash H \to A$?
2. Do HL-grammars generate the language of (a) complete binary graphs? (b) planar binary graphs? (c) directed acyclic binary graphs?
3. What string languages can be generated by HL-grammars?
4. Is the class of languages generated by HL-grammars closed under intersections?
5. What nontrivial upper bounds (like the pumping lemma for HRGs) exist for languages generated by HL-grammars?
6. Can HL-grammars be embedded in some known kind of graph grammars?

We are interested in further and deeper study of generalizations of logical approaches and concepts to as graphs; we think that this allows one to better understand the nature of the considered notions.

Acknowledgments

I am grateful to my scientific advisor Mati Pentus for his careful attention to my studies and anonymous reviewers for their valuable advice.

References

[1] Daniel Bauer & Owen Rambow (2016): *Hyperedge Replacement and Nonprojective Dependency Structures*. In David Chiang & Alexander Koller, editors: *Proceedings of the 12th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+12), June 29 - July 1, 2016, Heinrich Heine University, Düsseldorf, Germany*, The Association for Computer Linguistics, pp. 103–111. Available at `https://www.aclweb.org/anthology/W16-3311/`.

[2] Frank Drewes & Anna Jonsson (2017): *Contextual Hyperedge Replacement Grammars for Abstract Meaning Representations*. In Marco Kuhlmann & Tatjana Scheffler, editors: *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms, TAG 2017, Umeå, Sweden, September 4-6, 2017*, Association for Computational Linguistics, pp. 102–111. Available at `https://www.aclweb.org/anthology/W17-6211/`.

[3] Frank Drewes, Hans-Jörg Kreowski & Annegret Habel (1997): *Hyperedge Replacement Graph Grammars*. In Grzegorz Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, World Scientific, pp. 95–162, doi:10.1142/9789812384720_0002.

[4] Sorcha Gilroy, Adam Lopez & Sebastian Maneth (2017): *Parsing Graphs with Regular Graph Grammars*. In Nancy Ide, Aurélie Herbelot & Lluís Màrquez, editors: *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics, *SEM @ACM 2017, Vancouver, Canada, August 3-4, 2017*, Association for Computational Linguistics, pp. 199–208, doi:10.18653/v1/S17-1024.

[5] Annegret Habel (1992): *Hyperedge Replacement: Grammars and Languages*. *Lecture Notes in Computer Science* 643, Springer, doi:10.1007/BFb0013875.

[6] Bevan K. Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann & Kevin Knight (2012): *Semantics-Based Machine Translation with Hyperedge Replacement Grammars*. In Martin Kay & Christian Boitet, editors: *COLING 2012, 24th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, 8-15 December 2012, Mumbai, India*, Indian Institute of Technology Bombay, pp. 1359–1376. Available at `https://www.aclweb.org/anthology/C12-1083/`.

[7] Joachim Lambek (1958): *The Mathematics of Sentence Structure*. *The American Mathematical Monthly* 65(3), pp. 154–170. Available at `http://www.jstor.org/stable/2310058`.

[8] Mati Pentus (1993): *Lambek Grammars Are Context Free*. In: *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, IEEE Computer Society, pp. 429–433, doi:10.1109/LICS.1993.287565.

[9] Mati Pentus (1995): *Models for the Lambek Calculus*. *Ann. Pure Appl. Log.* 75(1-2), pp. 179–213, doi:10.1016/0168-0072(94)00063-9.

[10] Tikhon Pshenitsyn (2020): *Hypergraph Basic Categorial Grammars*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Online, June 25-26, 2020, Proceedings*, *Lecture Notes in Computer Science* 12150, Springer, pp. 146–162, doi:10.1007/978-3-030-51372-6_9.

[11] Tikhon Pshenitsyn (2020): *Weak Greibach Normal Form for Hyperedge Replacement Grammars*. In Berthold Hoffmann & Mark Minas, editors: *Proceedings of the Eleventh International Workshop on Graph Computation Models, GCM@STAF 2020, Online-Workshop, 24th June 2020, EPTCS* 330, pp. 108–125, doi:10.4204/EPTCS.330.7.

[12] Tikhon Pshenitsyn (2021): *Introduction to a Hypergraph Logic Unifying Different Variants of the Lambek Calculus*. Available at `https://arxiv.org/abs/2103.01199`.

Appendices

A Proof of Theorem 3

All the results proved in this section are presented in a paper accepted for ICGT 2021.

Definition 23. A type A is called simple if one of the following holds:

- A is primitive;
- $A = \times(M)$, $E_M = \{m_1, \dots, m_l\}$ and $lab(m_1), \dots, lab(m_l)$ are simple;
- $A = N \div D$, $E_D = \{d_0, \dots, d_k\}$, $lab(d_0) = \$$, N is simple, and $lab(d_1)$, …, $lab(d_k)$ are *primitive*.

Firstly, let us prove the following

Theorem 9. *Let* HL $\vdash H \to P$ *where H is labeled by simple types and P is either primitive or is of the form $\times(K)$ where all edge labels in K are primitive. Then there exists a* simple derivation *of $H \to P$, i.e. such a derivation that*

1. *The rule $(\to \times)$ either is not applied or is applied once to one of the leaves of the derivation tree.*
2. *In each application of $(\div \to)$, all the premises, possibly except for the first one, are of the form $q^\bullet \to q$, $q \in Pr$.*
3. *If a sequent $H' \to p$ within the derivation tree contains a type of the form $\times(M)$ in the antecedent, then the rule, after which this sequent appears, must be $(\times \to)$.*

Proof. Firstly, note that if $P = \times(K)$, then the rule $(\to \times)$ has to be applied one time in a derivation. Let this be as follows (where $E_K = \{k_1, \dots, k_m\}$):

$$\frac{G_1 \to lab_K(k_1) \quad \dots \quad G_m \to lab_K(k_m)}{K[k_1/G_1, \dots, k_m/G_m] \to \times(K)} \ (\to \times)$$

A derivation of $G_i \to lab_K(k_i)$, $i = 1, \dots, m$ is a sequence of applications of rules $(\times \to)$ and $(\div \to)$ only since $lab_K(k_i)$ is primitive. Thus we can repeat this sequence of derivations within $K[k_1/G_1, \dots, k_m/G_m] \to \times(K)$ from bottom to top for $i = 1$, $i = 2$, ..., $i = m$. After this we obtain the sequent $K \to \times(K)$ and now apply the rule $(\to \times)$. Therefore, each derivation of $H \to P$ can be remodeled in such a way that the condition 3 is met. Let us further consider such a derivation (name it Δ).

Now let us prove that Δ can be remodeled in such a way that a new one will satisfy conditions 2 and 3 as well. This is done by induction on length of Δ.

If $H \to P$ is an axiom (particularly, P is primitive), then both requirements are satisfied.

If H contains an edge e_0 labeled by a type $\times(M)$, then we can derive a sequent $H[e_0/M] \to P$ (see Proposition 1). Since length of a derivation equals the total number of symbols $\times$ and $\div$ included in types of an antecedent and a succedent, length of derivation of $H[e_0/M] \to P$ is less than that of $H \to P$; thus we can apply the induction hypothesis and obtain a simple derivation for $H[e_0/M] \to P$. The it suffices to apply the rule $(\times \to)$ to this sequent:

$$\frac{H[e_0/M] \to P}{H \to P} \ (\times \to)$$

Hence we obtained a simple derivation for $H \to P$.

Let H not contain types of the form $\times(M)$. Then the last step of ane derivation must be of the form

$$\frac{G \to P \quad H_1 \to lab(d_1) \quad \dots \quad H_k \to lab(d_k)}{G[e/D][d_0 := N \div D][d_1/H_1, \dots, d_k/H_k] \to P} \ (\div \to)$$

where $H = G[e/D][d_0 := N \div D][d_1/H_1, \dots, d_k/H_k]$ (otherwise, if the last step is not $(\div \to)$, we have $H \to P = K \to \times(K)$, and this sequent obviously has a simple derivation). Applying the induction hypothesis, we obtain that there are simple derivations for sequents $H_i \to lab(d_i)$; each of such derivations is a sequence of applications of the rules $(\div \to)$ and $(\times \to)$. Now we construct a derivation of $H \to P$ from bottom to top as follows: firstly, we repeat the simple derivation of $H_1 \to lab(d_1)$ (but now we consider H_1 to be a subgraph of H and disregard $lab(d_i)$ in the succedent), then we repeat the simple derivation of $H_2 \to lab(d_2)$ within H and so on until $H_k \to lab(d_k)$. Now we have a sequent of the form $G[e/D][d_0 := N \div D] \to P$ as a premise. Then we apply $(\div \to)$ to $N \div D$ by "overlaying" the denominator on edges of D; thus, each premise except for the first one is of the form $lab(d_i)^\bullet \to lab(d_i)$, and the first one is $G \to P$. Now we can apply the induction hypothesis to $G \to P$, which shows that $G \to P$ can also be derived in the fashion stated in the lemma. □

Now we proceed with the proof of Theorem 3.

Proof of Theorem 3. Let an HRG be of the form $HGr = \langle N, \Sigma, P, S \rangle$. Applying Theorem 2 we can assume that HGr is in the weak Greibach normal form.

Consider elements of N as elements of Pr with the same function *rank* defined on them. Since HGr is in the weak Greibach normal form, each production in P is of the form $\pi = X \to G$ where G contains exactly one terminal edge e_0 (say $lab_G(e_0) = a \in \Sigma$). We convert this production into a type $T_\pi := X \div G[e_0 := \$]$. Then we introduce the HL-grammar $HGr' = \langle \Sigma, S, \triangleright \rangle$ where $\triangleright$ is defined as follows: $a \triangleright T_\pi$. If $G = a^\bullet$, then we can simply write $a \triangleright X$ instead.

The main objective is to prove that $L(HGr) = L(HGr')$. Firstly, we are going to prove that $T^\bullet \overset{k}{\Rightarrow} H$ for $T \in N$, $H \in \mathscr{H}(\Sigma)$ **only if** $\mathrm{HL} \vdash f(H) \to T$ for some $f : E_H \to Tp(\mathrm{HL})$ such that $lab_H(e) \triangleright f(e)$ for all $e \in E_H$. This is done by induction on k.

Basis. In such a case $\pi = T \to H$ belongs to P and $E_H = \{e_0\}$. Then we can derive $\mathrm{HL} \vdash H[e_0 := S \div H[e_0 := \$]] \to T$ in one step using $(\div \to)$ (since $|E_H| = 1$).

Step. Let the first step of the derivation be of the form $T \Rightarrow G$ ($\pi = T \to G \in P$) and let $E_G = \{e_0, \dots, e_n\}$ where $lab_G(e_0) \in \Sigma$ and $lab_G(e_i) \in N$ otherwise. Let $G_i \in \mathscr{H}(\Sigma)$ be a graph that is obtained from $T_i = lab_G(e_i)$ in the derivation process ($i = 1, \dots, n$). Note that $H = G[e_1/G_1, \dots, e_n/G_n]$. By the induction hypothesis, $\mathrm{HL} \vdash f_i(G_i) \to T_i$ for such $f_i : E_{G_i} \to Tp(\mathrm{HL})$ that $lab_{G_i}(e) \triangleright f_i(e)$. Then f_i can be combined into a single function f as follows: $f(e) := f_i(e)$ whenever $e \in G_i$ and $f(e_0) := T_\pi$. Then we construct the following derivation (recall that $T_\pi = T \div G[e_0 := \$]$):

$$\frac{T^\bullet \to T \quad f_1(G_1) \to T_1 \quad \dots \quad f_n(G_n) \to T_n}{(G[e_0 := \$])[e_0 := T_\pi][e_1/f_1(G_1), \dots, e_n/f_n(G_n)] \to T} \ (\div \to)$$

This completes the proof since $G[e_0 := T_\pi][e_1/f_1(G_1), \dots, e_n/f_n(G_n)] = f(H)$.

Secondly, we explain why $L(HGr') \subseteq L(HGr)$. Note that types in the dictionary of HGr' are simple; thus for each derivable sequent of the form $H \to S$ over this dictionary we can apply Theorem 9 and obtain a derivation where each premise except for, possibly, the first one is an axiom. Now we can transform a derivation tree of HL into a derivation tree in the HRG HGr: each application of $(\div \to)$ such that a type T_π appears after it is transformed into an application of π in HGr. Formally, we have to use induction again. □

B Proof of Proposition 2

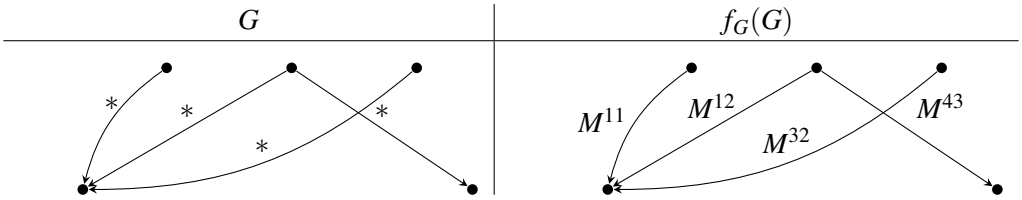
Proof sketch. Let us prove that $\mathscr{L}_2 \subseteq L(HGr_2)$. Given a bipartite binary graph $G \in \mathscr{L}_2$, we call its nodes, from which edges outgo, *out-nodes*, and its nodes, to which edges arrive, *in-nodes*. Firstly, given an edge going from an out-node v_1 to an in-node v_2, we remove it, attach a hyperedge labeled by $R_i(p)$ to v_1, and attach a hyperedge labeled by $R_j(q)$ to v_2 (for now, we do not choose i and j). Notice that, since G is bipartite, there is no node, to which hyperedges with labels including different primitive types are attached. Let us look at some former out-node, to which only hyperedges labeled by types of the form $R_i(p)$ are attached now. We do the following:

1. Case 1: if only one hyperedge labeled by $R_i(p)$ is attached to this node, then we do not choose i yet.
2. Case 2: if there are two or more (say, k) hyperedges, then we label one of them by $R_1(p)$, $(k-2)$ ones by $R_2(p)$, but we do not choose a specific label for the remaining hyperedge yet.

Secondly, let us fix some numbering on former out-nodes. We look at the first out-node, and choose $R_1(p)$ as a label for the only hyperedge without a fixed label, if Case 1 is the case for this node; otherwise, we choose $R_2(p)$. As for the remaining out-nodes, we choose $R_3(p)$, if Case 1 is the case, and we choose $R_4(p)$, if Case 2 is the case.

Thirdly, we perform a similar procedure with in-nodes (while replacing p by q everywhere). Finally, we return at the first step of our procedure, and do the opposite: we combine each pair of hyperedges labeled by $R_i(p)$ and $R_j(q)$ into a type M^{ij} and put it as a new label for a corresponding edge of G (notice that now i and j are defined). Thus, we have presented a relabeling $f_G : E_G \to \{M^{ij} \mid 1 \le i, j \le 4\}$. We argue that $\mathrm{HL} \vdash f_G(G) \to S$. The above construction and the way we are going to prove this statement can be understood with the use of the following

Example 13. Let G be the following bipartite binary graph:



$f_G(G)$ above is a picture of G after the relabeling defined above. Finally, a derivation of $f_G(G) \to S$ is given in Figure 2.

The other direction of the proof is to show that HGr_2 accepts only bipartite graphs. If a graph H belongs to $L(HGr_2)$, then there exists a function f_H such that $lab_H(e) \triangleright f_H(e)$ and $\mathrm{HL} \vdash f_H(H) \to S$. We notice that $f_H(H) \to S$ satisfies all the conditions of Theorem 9, and hence there exists a derivation of this sequent meeting the following requirements:

1. The first part of a derivation (if we consider it from bottom to top) consists of applications of the rule $(\times \to)$ to all types in the antecedent.
2. The second part of a derivation consists of applications of the rule $(\div \to)$, in which all premises except for the first one are axioms.
3. The third part of a derivation (from bottom to top) is one application of $(\to \times)$; consequently, it is applied to the sequent $\bullet \overset{1}{-} \boxed{p}\,\boxed{q} \overset{1}{-} \bullet \to S$.

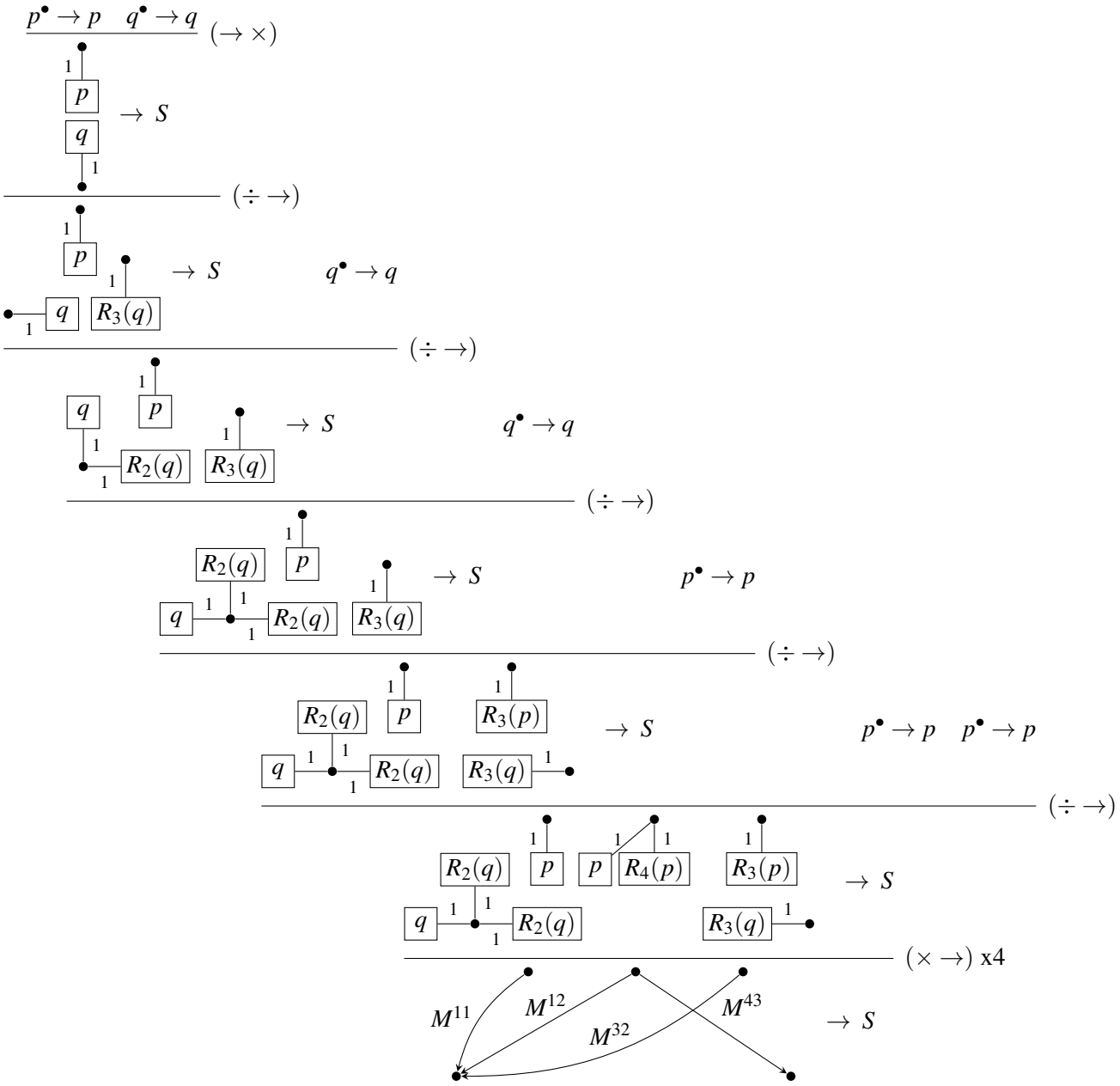
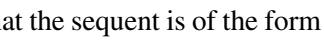


Figure 2: Derivation of a sequent corresponding to a bipartite graph.

For example, a derivation given in Figure 2 is simple, and it satisfies the above conditions. Now, we look at this derivation from bottom to top and notice that there is no way for hyperedges of the form $R_i(p)$ and $R_j(q)$ for some i, j to appear in the antecedent being attached to the same node during the second and the third parts. This is formally proved by induction on length l of the second part. The base case ($l = 0$) means that the sequent is of the form  where p- and q-labeled hyperedges are clearly attached to different nodes. To prove the step case, it suffices to look at the construction of types $R_i(r)$ and to notice that, when such a type appears in the antecendent, it preserves a desired property.

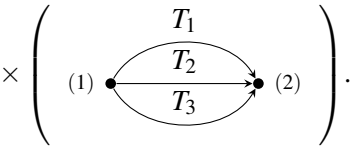
Concluding the proof, we observe that the first part of a derivation joins pairs of types of the form $R_i(p)$ and $R_j(q)$ into an M^{ij}-labeled edge. According to the property established above, nodes, to which $R_i(p)$-labeled edges are attached, may be considered as in-nodes, and nodes, to which $R_j(q)$-labeled edges are attached, may be considered as out-nodes in the final graph; hence, it is bipartite. □

C Proof of Theorem 6

Theorem 6 and its proof is a central part of our paper accepted for ICGT 2021.

Definition 24. An ersatz conjunction $\wedge_E(T_1,\dots,T_k)$ of types $T_1,\dots,T_k \in Tp(\mathrm{HL})$ (such that $rank(T_1) = \dots = rank(T_k) = m$) is the type $\times(H)$ where

1. $V_H = \{v_1,\dots,v_m\}$;
2. $E_H = \{e_1,\dots,e_k\}$;
3. $att_H(e_i) = v_1 \dots v_m$;
4. $lab_H(e_i) = T_i$;
5. $ext_H = v_1 \dots v_m$.

Example 14. Let T_1, T_2, T_3 be types with *rank* equal to 2. Then their ersatz conjunction equals $\wedge_E(T_1,T_2,T_3) =$  .

Proof of Theorem 6. Using the construction from Theorem 3 we construct an HL-grammar HGr_i for each $i = 1\dots,k$ such that $L(HGr_i') = L(HGr_i)$. We assume without loss of generality that types involved in HGr_i and HGr_j for $i \neq j$ do not have common primitive subtypes (let us denote the set of primitive subtypes of types in $dict(HGr_i)$ as Pr_i). Let us denote $HGr_i = \langle \Sigma, s_i, \triangleright_i \rangle$. Note that $rank(s_1) = \dots = rank(s_k)$ (otherwise $L(HGr_1) \cap \dots \cap L(HGr_k) = \emptyset$, and the theorem holds due to trivial reasons). The main idea then is to do the following: given $a \triangleright_i T_i$, $i = 1,\dots,k$ we join $T_1,\dots,T_k$ using ersatz conjunction. A distinguished type of the new grammar will also be constructed from $s_1,\dots s_k$ using $\wedge_E$. Then a derivation is expected to split into k independent parts corresponding to derivations in grammars $HGr_1,\dots,HGr_k$. However, there is a nuance that spoils simplicity of this idea; it is related to the issue of isolated nodes. This nuance leads to a technical trick, which we call "tying balloons".

Let us fix $(k-1)$ new primitive types $b_1,\dots,b_{k-1}$ ("balloon" labels) such that $rank(b_i) = 1$. For $j < k$ we define a function $\varphi_j : dict(HGr_j) \to Tp(\mathrm{HL})$ as follows: $\varphi_j(p) = p$ whenever $p \in Pr$; $\varphi_j(p \div D) = \times(M) \div D'$ where

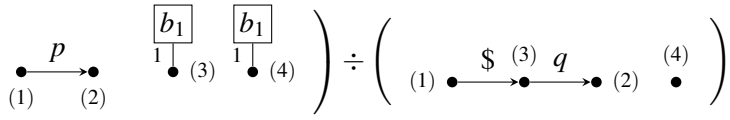
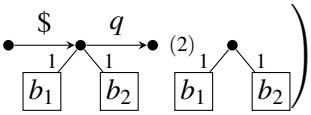
1. $D' = \langle V_D, E_D, att_D, lab_D, ext_D w \rangle$ where $[w] = V_D \setminus [ext_D]$ (that is, w consists of nodes that are not external in D; here $[w]$ is the set of symbols in w).

2. Denote $m = |w| = |V_D| - |ext_D|$, and $t = rank(p)$. Then $M = \langle \{v_1, \ldots, v_{t+m}\}, \{e_0, e_1, \ldots, e_m\}, att, lab, v_1 \ldots v_{t+m} \rangle$ where $att(e_0) = v_1 \ldots v_t$, $lab(e_0) = p$; $att(e_i) = v_{t+i}$, $lab(e_i) = b_j$ whenever $i = 1, \ldots, m$.

Informally, we make all nodes in the denominator D external, while $\times(M)$ "ties a balloon" labeled b_j to each node corresponding to a nonexternal one in D. Presence of these "balloon edges" is compensated by modified types of the grammar HGr_k. Namely, we define a function $\varphi_k : dict(HGr_k) \to Tp(\mathrm{HL})$ as follows: $\varphi_k(p) = p$ whenever $p \in Pr$; $\varphi_k(p \div D) = p \div D'$ where $D' = \langle V_D, E_D \cup \{e_1, \ldots, e_{(k-1)m}\}, att, lab, ext_D \rangle$ such that:

1. $m = |V_D| - |ext_D|$;
2. $e_1, \ldots, e_{(k-1)m}$ are new edges;
3. $att|_{E_D} = att_D$;
4. If $v_1, \ldots, v_m$ are all nonexternal nodes of D, then $att(e_i) = v_{\lceil i/(k-1) \rceil}$ for $i = 1, \ldots, (k-1)m$. In other words, we attach $(k-1)$ new edges to each nonexternal node of D.
5. $lab(e_i) = b_{g(i)}$, $i = 1, \ldots, (k-1)m$ where $g(i) = i \mod (k-1)$ if $(k-1) \nmid i$ and $g(i) = k-1$ otherwise. That is, for each $b_i, i = 1, \ldots, (k-1)$ and for each nonexternal node there is a b_i-labeled edge attached to it.

Example 15. Let $k = 3$ and let $T = p \div \Big($  $\Big)$. Then

- $\varphi_1(T) = \times \Big($  $\Big) \div \Big($ $\Big)$
- $\varphi_3(T) = p \div \Big($  $\Big)$

Now we are ready to introduce HGr: $HGr = \langle \Sigma, S, \triangleright \rangle$ where

- $a \triangleright T \Leftrightarrow T = \wedge_E(\varphi_1(T_1), \ldots, \varphi_k(T_k))$ and $\forall i = 1, \ldots, k \quad a \triangleright_i T_i$;
- $S = \wedge_E(s_1, \ldots, s_k)$.

The proof of $L(HGr) = L(HGr_1) \cap \ldots \cap L(HGr_k)$ is divided into two parts: the $\subseteq$-inclusion proof and the $\supseteq$-inclusion proof.

Proof of the $\supseteq$-inclusion. A hypergraph $H \in \mathscr{H}(\Sigma)$ belongs to $L(HGr_1) \cap \cdots \cap L(HGr_k)$ if and only if there are relabeling functions $f_i : E_H \to Tp(\mathrm{HL})$ such that $lab_H(e) \triangleright_i f_i(e)$ for all $e \in E_H$, and $\mathrm{HL} \vdash f_i(H) \to s_i$. Using these relabelings we construct a relabeling $f : E_H \to Tp(\mathrm{HL})$ as follows: if $f_i(e) = T_i$, then $f(e) := \wedge_E(\varphi_1(T_1), \ldots, \varphi_k(T_k))$. It follows directly from the definition that $lab_H(e) \triangleright f(e)$. Now we construct a derivation of $f(H) \to \wedge_E(s_1, \ldots, s_k)$ from bottom to top:

1. We apply rules $(\times \to)$ to all ersatz conjunctions in the antecedent. This yields a graph without $\times$-labels, which has k "layers" belonging to grammars $HGr_1, \ldots, HGr_k$.
2. We remodel a derivation of $f_1(H) \to s_1$, which consists of $(\div \to)$-applications only, using types of the form $\varphi_1(f_1(e)), e \in E_H$ that are present in $f(H)$. The only difference now is that nonexternal nodes do not "disappear" (recall that a derivation is considered from bottom to top), and edges labeled by types with $\times$ appear. Every time when $\times$ appears in the left-hand side we immediately

apply $(\times \to)$, which results in adding one edge labeled by a primitive type from Pr_1 and in adding balloon edges to all nodes that would disappear in the derivation of $f_1(H) \to s_1$.

The result of this part of a derivation is that now all types corresponding to HGr_1 left the antecedent, except for the only s_1-labeled edge attached to the external nodes of the antecedent in the right order; besides, for each nonexternal node in the antecedent there is now a balloon edge labeled by b_1 attached to it.

3. We perform $(k-2)$ more steps similarly to Step 2 using types of the form $\varphi_i(f_i(e)), 1 < i < k$ and thus remodeling a derivation $f_i(H) \to s_i$. Upon completion of all these steps the antecedent contains:
   - Types of the form $\varphi_k(f_k(e)), e \in E_H$;
   - $(k-1)$ edges labeled by $s_1, \dots, s_{k-1}$ resp. and attached to external nodes of the antecedent;
   - Balloon edges such that for each $j \in \{1, \dots, k-1\}$ and for each nonexternal node there is a b_j-labeled edge attached to it.
4. We remodel a derivation of $f_k(H) \to s_k$ using types of the form $\varphi_k(f_k(e))$. A situation differs from those at steps 2 and 3 because now nonexternal nodes do disappear, and each time when this happens all balloon edges attached to a nonexternal node disappear as well.

   After this step, all balloon edges are removed, and we obtain a graph with $rank(s_1)$ nodes such that all of them are external, and with k edges labeled by $s_1, \dots, s_k$ such that their attachment nodes coincide with external nodes of the graph. This ends the proof since $\wedge_E(s_1, \dots, s_k)$ is exactly this graph standing under $\times$.

Proof of the $\subseteq$-inclusion. Let H be in $L(HGr)$; then there is a function $\Phi : E_H \to Tp(\mathrm{HL})$ such that $\Phi(e) = \wedge_E(\varphi_1(T_1(e)), , \dots, \varphi_k(T_k(e)))$ whenever $e \in E_H$, $lab(e) \triangleright_i T_i(e)$, and $\Phi(H) \to S$ is derivable in HL. We aim to decompose the derivation of this sequent into k ones in grammars $HGr_1, \dots, HGr_k$. In order to do this we transform the derivation in stages:

Stage 1. Using Proposition 1 we replace each edge in $\Phi(H)$ labeled by a type of the form $\times(M)$ by M. A new sequent (denote it by $H' \to S$) is derivable as well.

Stage 2. The sequent $H' \to S$ fits in Theorem 9; hence there exists its simple derivation. Let us fix some simple derivation of $H' \to S$ and call it Δ.

Furthermore we consider all derivations from bottom to top (in particular, if we state "X is after Y" regarding some places X and Y in a derivation, then we mean that X is above Y in the derivation tree).

Stage 3. Design of types $\varphi_i(T)$ differs in the case $i < k$ and $i = k$. Namely, if $\varphi_i(T)$ for $i < k$ participates in the rule $(\div \to)$ in Δ, this affects only primitive types from Pr_i; on the contrary, participating of $\varphi_k(T)$ in $(\div \to)$ affects types from Pr_k but also balloon types $b_1, \dots, b_{k-1}$, which appear after rule applications of $(\div \to)$ and $(\times \to)$ to several types of the form $\varphi_i(T)$, $i < k$. This allows us to come up with the following conclusion: if a rule $(\div \to)$ application to a type of the form $\varphi_k(T)$ preceeds (from bottom to top) a rule application of $(\div \to)$ to a type of the form $\varphi_i(T)$ for $i < k$, then we can change their order (note also that all nodes in the denominator of $\varphi_i(T)$ are external). Thus Δ can be remade in such a way that all rules affecting $\varphi_k(T)$ will occur upper than rules affecting $\varphi_i(T), i < k$ in a derivation (and it will remain simple). Let us call a resulting derivation Δ'.

Stage 4. A denominator of a type $\varphi_i(T)$ for $i < k$ contains edges labeled by elements of Pr_i only. Since Δ' is simple, applications of the rule $(\div \to)$ to types of the form $\varphi_i(T)$ and $\varphi_j(T')$ for $i \neq j$ are independent, and their order can be changed. This means that we can reorganize Δ' in the following way (from bottom to top):

1. Set $i = 1$.

2. Apply the rule $(\div \to)$ to a type of the form $\varphi_i(T)$ and right away the rule $(\times \to)$ to its numerator.
3. If there still are types of the form $\varphi_i(T)$, repeat step 2;
4. If $i = k-1$, go forward; otherwise, set $i = i+1$ and go back to step 2.
5. Apply the rule $(\div \to)$ to types of the form $\varphi_k(T)$.
6. Now an antecedent of the major sequent (denote this sequent as $G \to S$) does not include types with $\div$ or $\times$. S is of the form $\times(M_S)$, and Theorem 9 provides that the last rule applied has to be $(\to \times)$; therefore, $G = M_S$ and we reach the sequent $M_S \to S$. Consequently, $G = M_S$ consists of k edges labeled by $s_1, \ldots, s_k$ resp.

Let us call this derivation Δ_0. Observe that, after steps 1-4 in the above description, balloon edges with labels $b_1, \ldots, b_{k-1}$ may occur in the antecedent of a sequent (denote this sequent, which appears after step 4, as $G' \to S$). There is only one way for them to disappear: they have to participate in the rule $(\div \to)$ with a type of the form $\varphi_k(T)$ (since only for such types it is the case that their denominators may contain balloon edges). Note, however, that balloon edges within the denominator of $\varphi_k(T)$ are attached only to nonexternal nodes. Therefore, balloon edges in G' can be attached only to nonexternal nodes as well. Besides, if some balloon edge labeled by b_i is attached to a node $v \in V_{G'} \setminus [ext_{G'}]$, then the set of balloon edges attached to v has to consist of exactly $k-1$ edges labeled by $b_1, \ldots, b_{k-1}$ (because in the denominator of $\varphi_k(T)$ exactly such edges are attached to each nonexternal node). Finally, note that after step 5 all nonexternal nodes disappear since M_S contains exactly $rank(S)$ nodes, all of which are external. This allows us to conclude that balloon edges have to be present on all nonexternal nodes (otherwise, a nonexternal node cannot go away interacting with a type of the form $\varphi_k(T)$). Informally, a balloon edge labeled by b_i indicates that a node was used by a type from the i-th grammar HGr_i, and $\varphi_k(T)$ verifies that each nonexternal node is used by the i-th grammar exactly once.

Summarizing all the above observations, we conclude that, after steps 1-4, there is exactly one balloon edge labeled by b_i on each nonexternal node of G' for all $i = 1, \ldots, k-1$ (and no balloon edge is attached to some external node of G'). The only way for b_i to appear attached to a node (recall that we consider the derivation from bottom to top) is to participate in the rule $(\times \to)$ after the application of $(\div \to)$ to a type of the form $\varphi_i(T)$. Now we are ready to decompose Δ_0 into k ones:

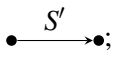
- For $1 \leq i < k$ we consider step 2 of Δ_0 with that only difference that we disregard balloon edges. Then the combination of rules $(\div \to)$ and $(\times \to)$ applied to a type $\varphi_i(T)$ turns into an application of the rule $(\div \to)$ to T in the HGr_i. Take into account that the only type that is built of elements of Pr_i and remains to step 6 is s_i attached to external nodes in the right order; therefore, if we remove from H' all edges not related to HGr_i and relabel each edge labeled by $\varphi_i(T)$ with T (call the resulting graph H'_i), then $H'_i \to s_i$ is derivable.
- For $i = k$ everything works similarly; however, instead of step 2 we have to look at step 5 and again not to consider balloon edges. Then an application of $(\div \to)$ to $\varphi_k(T)$ is transformed into a similar application of $(\div \to)$ to T in HGr_k. After the whole process, only s_k remains, so, if H'_k is a graph obtained from H' by removing edges not related to HGr_k and changing each label of the form $\varphi_k(T)$ by T, then $H'_k \to s_k$ is derivable.

Finally note that $H'_i = \Phi_i(H)$ where $\Phi_i(e) = T_i(e)$. The requirement $lab(e) \triangleright_i T_i(e)$ completes the proof, because thus $H \in L(HGr_i)$ for all $i = 1, \ldots, k$. □

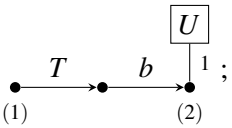
The balloon trick is used here to control that making all nodes in denominators of $\varphi_i(T)$ external ($i < k$) does not lead to using, e.g., a nonexternal isolated node in rules $(\div \to)$ more than once.

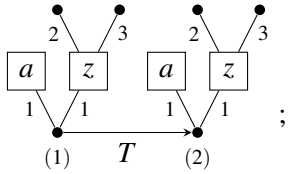
D Proof of an Auxiliary Fact From Theorem 5

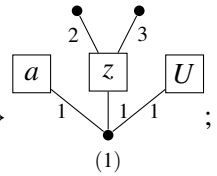
Our goal in this section is to prove that L_1 and L_2 introduced in the proof of Theorem 5 can be generated by some HRGs. We will present an HRG generating L_1; L_2 is generated similarly. Namely, we argue that $L_1 = L(HGr_1)$ where $HGr_1 = \langle \{S, S', T, U\}, \{a, z, b\}, P, S \rangle$ and productions from P are listed below:

- $S \to$ 

;
- $S \to$ 

;
- $S' \to (TbS')^{\bullet}$;
- $S' \to$ 

;
- $S' \to T^{\bullet}$;
- $T \to$ 

;
- $T \to b^{\bullet}$;
- $U \to$ 

;
- $U \to$ 

.

The fact that $L(HGr_1) = L_1$ directly follows from the construction of HGr_1.

From Linear Term Rewriting to Graph Rewriting with Preservation of Termination

Roy Overbeek Jörg Endrullis
Vrije Universiteit Amsterdam
Amsterdam
r.overbeek@vu.nl j.endrullis@vu.nl

Encodings of term rewriting systems (TRSs) into graph rewriting systems usually lose global termination, meaning the encodings do not terminate on all graphs. A typical encoding of the terminating TRS rule $a(b(x)) \to b(a(x))$, for example, may be indefinitely applicable along a cycle of a's and b's. Recently, we introduced PBPO$^+$, a graph rewriting formalism in which rules employ a type graph to specify transformations and control rule applicability. In the present paper, we show that PBPO$^+$ allows for a natural encoding of linear TRS rules that preserves termination globally. This result is a step towards modeling other rewriting formalisms, such as lambda calculus and higher order rewriting, using graph rewriting in a way that preserves properties like termination and confluence. We moreover expect that the encoding can serve as a guide for lifting TRS termination methods to PBPO$^+$ rewriting.

1 Introduction

A *rewriting framework* $\mathcal{F}$ consists of a set of *objects* O and a set of *rewriting systems* $\mathcal{R}$. Each system $R \in \mathcal{R}$ is a set of *rewrite rules*. Each rule $\rho \in R$ defines a particular *rewrite relation* $\to_\rho \subseteq O \times O$ on objects, and the rules of R collectively give rise to a general rewrite relation $\to_R = \bigcup_{\rho \in R} \to_\rho$. The usual definitions of string, cycle and term rewriting systems (TRSs), and the various definitions of term graph and graph rewriting formalisms, are instances of this abstract view.

Because terms can be viewed as generalizations of strings, term graphs as generalizations of terms, graphs as generalizations of terms graphs and cycles, etc., the question whether one framework can be encoded into another framework frequently arises naturally. The same is true when comparing the large variety of graph rewriting frameworks. Morever, the properties such an encoding is expected to satisfy may vary. Let us therefore fix some vocabulary.

Definition 1 (Encoding)**.** An *encoding* $\mathcal{E}$ of a framework $\mathcal{F}$ into a framework $\mathcal{G}$ consists of a function $\mathcal{E}_O : O^\mathcal{F} \to O^\mathcal{G}$ on objects and a function $\mathcal{E}_\mathcal{R} : \mathcal{R}^\mathcal{F} \to \mathcal{R}^\mathcal{G}$ on rewrite systems. The subscript is usually omitted, since it will be clear from context which of $\mathcal{E}_O$ and $\mathcal{E}_\mathcal{R}$ is meant.

Given an encoding $\mathcal{E}$, a variety of properties of interest may be distinguished. We will say $\mathcal{E}$ is

1. *step-preserving* if $x \to_R^\mathcal{F} y \implies \mathcal{E}(x) \to_{\mathcal{E}(R)}^\mathcal{G} \mathcal{E}(y)$;
2. *closed* if $x \to_{\mathcal{E}(R)}^\mathcal{G} y$ and $x \cong \mathcal{E}(x')$ for some $x' \in O^\mathcal{F} \implies y \cong \mathcal{E}(y')$ for some $y' \in O^\mathcal{F}$ with $x' \to_R y'$;
3. an *embedding* if $\mathcal{E}$ is step-preserving and closed;
4. *globally P-preserving* (for a property P, such as termination or confluence), if whenever $R \in \mathcal{R}^\mathcal{F}$ satisfies P, then so does the system $\mathcal{E}(R) \in \mathcal{R}^\mathcal{G}$ on all objects $O^\mathcal{G}$; and
5. *locally P-preserving* if whenever $R \in \mathcal{R}^\mathcal{F}$ satisfies P, then so does the system $\mathcal{E}(R) \in \mathcal{R}^\mathcal{G}$ on the restricted domain of objects $\mathcal{E}(O^\mathcal{F}) \subseteq O^\mathcal{G}$.

Consider the string rewrite rule $ab \to ba$ and its usual encoding $a(b(x)) \to b(a(x))$ as a term rewrite rule. This encoding is an embedding that preserves termination and confluence globally. The usual encoding as a cycle rewrite rule, by contrast, is step-preserving, but not closed, and neither termination- nor confluence-preserving.

Building on PBPO by Corradini et al. [4] and our own patch graph rewriting formalism [16], we recently proposed the PBPO$^+$ algebraic graph rewriting approach [17], in which rules employ a type graph to specify transformations and control rule applicability. In the present paper we give an embedding of linear term rewrite systems into PBPO$^+$ that preserves global termination, despite being applicable to graphs that are not encodings of terms. This result requires powerful features (unsupported by standard approaches such as DPO [8]), as two examples illustrate:

1. For the encoding of $f(x,y) \to f(a,y)$ to be step-preserving, it must be possible to delete an arbitrary subgraph x below f, while leaving the context above of f and the subgraph corresponding to y intact.
2. For the encoding of $a(b(x)) \to b(a(x))$ to be terminating, the rule must not be applicable on a cycle of a's and b's.

Apart from being an interesting expressiveness result for PBPO$^+$, our result enables reduction-style termination arguments for linear, 'term-like' PBPO$^+$ rewrite rules. Moreover, as we will elaborate in the discussion (Section 6), we believe our result has broader relevance for the development of termination techniques for graph rewriting, as well as the modeling of other rewrite formalisms.

The structure of the paper is as follows. In Section 2, we summarize the relevant categorical and TRS preliminaries. In Section 3, we give a self-contained introduction to PBPO$^+$ and to $\mathbf{Graph}^{(\mathcal{L},\leq)}$ [17], a special category that combines well with PBPO$^+$. In Section 4, we define an embedding of linear term rewriting into PBPO$^+$ rewriting over category $\mathbf{Graph}^{(\mathcal{L},\leq)}$. In Section 5, we prove that the embedding is globally termination-preserving, using a novel zoning proof. Finally, we discuss the significance of our results in Section 6.

2 Preliminaries

We assume familiarity with various basic categorical notions, notations and results, including morphisms $X \to Y$, pullbacks and pushouts, monomorphisms (monos) $X \rightarrowtail Y$ (note the different arrow notation) and identities $1_X : X \rightarrowtail X$ [1, 14].

Definition 2 (Graph Notions)**.** A (labeled) *graph* G consists of a set of vertices V, a set of edges E, source and target functions $s,t : E \to V$, and label functions $\ell^V : V \to \mathcal{L}$ and $\ell^E : E \to \mathcal{L}$ for some label set $\mathcal{L}$. A graph is *unlabeled* if $\mathcal{L}$ is a singleton.

A *premorphism* between graphs G and G' is a pair of maps $\phi = (\phi_V : V_G \to V_{G'}, \phi_E : E_G \to E_{G'})$ satisfying $(s_{G'}, t_{G'}) \circ \phi_E = \phi_V \circ (s_G, t_G)$.

A *homomorphism* is a label-preserving premorphism ϕ, i.e., a premorphism satisfying $\ell^V_{G'} \circ \phi_V = \ell^V_G$ and $\ell^E_{G'} \circ \phi_E = \ell^E_G$.

Definition 3 (Category **Graph** [7])**.** The category **Graph** has graphs as objects, parameterized over some global (and usually implicit) label set $\mathcal{L}$, and homomorphisms as arrows. **FinGraph** is the full subcategory of finite graphs.

The following TRS definitions are all standard [18].

Definition 4 (Signature)**.** A *signature* Σ consists of a non-empty set of *function symbols* $f, g, \ldots \in \Sigma$, equipped with an *arity function* $\# : \Sigma \to \mathbb{N}$. Nullary function symbols $a, b, \ldots$ are called *constants*.

Definition 5 (Terms)**.** The set of *terms* $\mathbf{l},\mathbf{r},\mathbf{s},\mathbf{t},\ldots \in \mathrm{Ter}(\Sigma,\mathcal{X})$ over a signature Σ and an infinite set of variables $x,y,\ldots \in \mathcal{X}$ is defined inductively by:

- $x \in \mathrm{Ter}(\Sigma,\mathcal{X})$ for every $x \in \mathcal{X}$;
- if $f \in \Sigma$ with $\#(f) = n$, and $\mathbf{t_1},\ldots,\mathbf{t_n} \in \mathrm{Ter}(\Sigma,\mathcal{X})$, then $f(\mathbf{t_1},\ldots,\mathbf{t_n}) \in \mathrm{Ter}(\Sigma,\mathcal{X})$. If $n = 0$, we write f instead of $f()$.

A term $\mathbf{t}$ is *linear* if every $x \in \mathcal{X}$ occurs at most once in $\mathbf{t}$. We write $\mathrm{Var}(\mathbf{t})$ to denote the set of variables occurring in $\mathbf{t}$.

Definition 6 (Position)**.** A *position* p is a sequence of integers, i.e., $p \in \mathbb{N}^*$. The empty sequence is denoted by ε. We write pn (and np) to denote the right (and left) concatenation of a positive integer n to a position p.

Every symbol occurrence in a term has a position associated with it. The position of the head symbol is ε, and the position of the i-th ($i \geq 1$) symbol below a symbol with position p is pi. For a term $\mathbf{s}$ and a position p in s, we write $\mathbf{s}(p)$ to denote the symbol at position p in $\mathbf{s}$.

Definition 7 (Substitutions)**.** A *substitution* is a function $\sigma : \mathcal{X} \to \mathrm{Ter}(\Sigma,\mathcal{X})$. For terms $\mathbf{s} \in \mathrm{Ter}(\Sigma,\mathcal{X})$ we define $\mathbf{s}\sigma \in \mathrm{Ter}(\Sigma,\mathcal{X})$ by $x\sigma = \sigma(x)$ for $x \in \mathcal{X}$, and $f(\mathbf{t_1},\ldots,\mathbf{t_n})\sigma = f(\mathbf{t_1}\sigma,\ldots,\mathbf{t_n}\sigma)$ for $f \in \Sigma$ and $\mathbf{t_1},\ldots,\mathbf{t_n} \in \mathrm{Ter}(\Sigma,\mathcal{X})$.

Definition 8 (Contexts)**.** A *context* $C[\,]$ is a term from $\mathrm{Ter}(\Sigma,\mathcal{X} \uplus \{\,\Box\,\})$ with exactly one occurrence of the *hole* $\Box$. We write $C[\mathbf{t}]$ for the term obtained by replacing the hole with $\mathbf{t}$.

Definition 9 (Term Rewriting Systems)**.** A *term rewrite rule* is a pair of terms $\mathbf{l} \to \mathbf{r}$ satisfying $\mathbf{l} \notin \mathcal{X}$ and $\mathrm{Var}(\mathbf{r}) \subseteq \mathrm{Var}(\mathbf{l})$. The rule is *linear* if both terms $\mathbf{l}$ and $\mathbf{r}$ are linear. A *term rewriting system* (*TRS*) $\mathcal{R}$ is a set of term rewrite rules. The system $\mathcal{R}$ is linear if all its rules are.

A TRS $\mathcal{R}$ induces a relation $\to$ on $\mathrm{Ter}(\Sigma,\mathcal{X})$, the *rewrite relation of* $\mathcal{R}$, as follows: $C[\mathbf{l}\sigma] \to C[\mathbf{r}\sigma]$ for every context C, substitution σ and rule $\mathbf{l} \to \mathbf{r} \in \mathcal{R}$. The *rewrite step* $C[\mathbf{l}\sigma] \to C[\mathbf{r}\sigma]$ is said to be an *application of the rule* $\rho = \mathbf{l} \to \mathbf{r}$ *at position* p, where p is the position of the hole in $C[\,]$.

3 PBPO$^+$ and Graph$^{(\mathcal{L},\leq)}$

We recently introduced PBPO$^+$ [17] (short for *PBPO with strong matching*), an algebraic rewriting formalism obtained by strengthening the matching mechanism of PBPO by Corradini et al. [4]. We believe PBPO$^+$ is of interest for at least three important reasons.

First, PBPO$^+$ is expressive: for **Graph** in particular, and assuming monic matching, we conjecture [17] that PBPO$^+$ is able to faithfully model DPO, SPO [13], SqPO [5], AGREE [3] and PBPO. More precisely, for any rule in such a formalism, there exists a PBPO$^+$ rule that generates exactly the same rewrite relation.

Second, PBPO$^+$ makes relatively weak assumptions on the underlying category: it is sufficient to require the existence of pushouts along monomorphisms and the existence of pullbacks. In particular, adhesivity [12], assumed for DPO rewriting to ensure the uniqueness of pushout complements, is not required.

Third, we have defined a non-adhesive category called **Graph**$^{(\mathcal{L},\leq)}$ [17] that combines very nicely with PBPO$^+$, allowing graph rewrite rules to easily model notions of relabeling, type systems, wildcards and variables. These notions have been significantly more challenging to define for DPO.

In this section we provide the necessary background on PBPO$^+$ and **Graph**$^{(\mathcal{L},\leq)}$.

Definition 10 (PBPO$^+$ Rewriting [17])**.** A *PBPO$^+$ rewrite rule* ρ (left) and *adherence morphism* $\alpha : G_L \to L'$ induce a *rewrite step* $G_L \Rightarrow^{\alpha}_{\rho} G_R$ on arbitrary G_L and G_R if the properties indicated by the commuting diagram on the right hold

$$\rho \;=\; \begin{array}{ccccc} & & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ & & t_L\big\downarrow & \text{PB} & \big\downarrow t_K & & \\ & & L' & \xleftarrow[l']{} & K' & & \end{array}$$

$$\begin{array}{ccccccc} & & & & K & \xrightarrow{r} & R \\ & & & & !u\big\downarrow & \text{PO} & \big\downarrow w \\ L & \xrightarrow{m} & G_L & \xleftarrow{g_L} & G_K & \xrightarrow[g_R]{} & G_R \\ 1_L\big\downarrow & \text{PB} & \alpha\big\downarrow & & \text{PB} & \big\downarrow u' & \swarrow t_K \\ L & \xrightarrow[t_L]{} & L' & \xleftarrow[l']{} & K' & & \end{array}$$

where $u : K \to G_K$ is the unique mono satisfying $t_K = u' \circ u$ [17, Lemma 11]. We write $G_L \Rightarrow_\rho G_R$ if $G_L \Rightarrow^{\alpha}_{\rho} G_R$ for some α.

In the rewrite rule diagram, L is the *lhs pattern* of the rule, L' its *type graph* and t_L the *typing* of L. Similarly for the *interface* K. R is the *rhs pattern* or *replacement for* L. The rewrite step diagram can be thought of as consisting of a match square (modeling an application condition), a pullback square for extracting (and possibly duplicating) parts of G_L, and finally a pushout square for gluing these parts along pattern R. The inclusion of the match square is the main aspect which differentiates PBPO$^+$ from PBPO: intuitively, it prevents α from collapsing context elements of G_L onto the pattern $t_L(L) \subseteq L'$.

For the present paper, it suffices to restrict attention to rules in which l' does not duplicate subgraphs.

Definition 11 (Linear PBPO$^+$ Rule)**.** A PBPO$^+$ rule is *linear* if the morphism $l' : K' \to L'$ is monic.

Remark 12. For linear PBPO$^+$ rewriting, it is enough to assume the existence of pushouts and pullbacks along monomorphisms. An interesting question is whether these weakened requirements enable new use cases.

The category $\mathbf{Graph}^{(\mathcal{L},\leq)}$ is similar to **Graph**. The difference is that it is assumed that the label set forms a complete lattice, and that morphisms do not decrease labels. The complete lattice requirement ensures that pushouts and pullbacks are well-defined.

Definition 13 (Complete Lattice)**.** A *complete lattice* $(\mathcal{L},\leq)$ is a poset such that all subsets S of $\mathcal{L}$ have a supremum (join) $\bigvee S$ and an infimum (meet) $\bigwedge S$.

Definition 14 (Category $\mathbf{Graph}^{(\mathcal{L},\leq)}$ [17])**.** For a complete lattice $(\mathcal{L},\leq)$, the category $\mathbf{Graph}^{(\mathcal{L},\leq)}$ is the category in which objects are graphs are labeled from $\mathcal{L}$, and arrows are graph premorphisms $\phi : G \to G'$ that satisfy $\ell_G(x) \leq \ell_{G'}(\phi(x))$ for all $x \in V_G \cup E_G$. We let $\mathbf{FinGraph}^{(\mathcal{L},\leq)}$ denote the full subcategory of finite graphs.

Proposition 15. *In* $\mathbf{Graph}^{(\mathcal{L},\leq)}$*, monomorphisms are stable under pushout.*

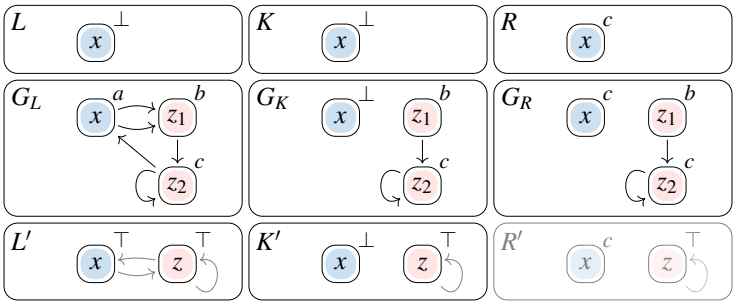
Proof. Assume given a span $B \overset{b}{\leftarrow} A \overset{c}{\rightarrowtail} C$ in $\mathbf{Graph}^{(\mathcal{L},\leq)}$. Overloading names, consider the unlabeled version in **Graph**, and construct the pushout $B \overset{m}{\to} D \overset{n}{\leftarrow} C$. Morphism $m : B \to D$ is monic, because monos are stable in the category of unlabeled graphs, by virtue of it being an adhesive category. Now for each $x \in V_D \cup E_D$, define the label function $\ell(x)$ to be the supremum of all labels in the labeled preimages $m^{-1}(x)$ and $n^{-1}(x)$, and define the $\mathbf{Graph}^{(\mathcal{L},\leq)}$ object $D_\ell = (V_D, E_D, s_D, t_D, \ell)$. Then it is easy to verify that $B \overset{m}{\rightarrowtail} D_\ell \overset{n}{\leftarrow} C$ is the pushout of $B \overset{b}{\leftarrow} A \overset{c}{\rightarrowtail} C$ in $\mathbf{Graph}^{(\mathcal{L},\leq)}$. □

In this paper we will use the following simple complete lattice only.

Definition 16 (Flat Lattice [17])**.** Let $\mathcal{L}^{\bot,\top} = \mathcal{L} \uplus \{\bot, \top\}$. We define the *flat lattice* induced by $\mathcal{L}$ as the poset $(\mathcal{L}^{\bot,\top}, \leq)$, which has $\bot$ as a global minimum and $\top$ as a global maximum, and where all elements of $\mathcal{L}$ are incomparable. In this context, we refer to $\mathcal{L}$ as the *base label set*.

The following example is a variation of an example found in our previous paper [17, Example 40]. It exemplifies all relevant features of linear PBPO$^+$ rewriting in category $\mathbf{Graph}^{(\mathcal{L},\leq)}$.

Example 17 (Rewrite Example)**.** As vertex labels we employ the flat lattice induced by the base label set $\{a,b,c,\dots\}$, and we assume edges are unlabeled for notational simplicity. The diagram



displays a rule (L,L',K,K',R) which

- matches an arbitrarily labeled, loopless node x, in an arbitrary context;
- "hard overwrites" the label of x to label c;
- disconnects x from its component by deleting its incident edges; and
- leaves all other nodes, edges and labels unchanged.

The pushout $K' \xrightarrow{r'} R' \xleftarrow{\iota_R} R$ for span $K' \xleftarrow{\iota_K} K \xrightarrow{r} R$ is depicted as well (in lower opacity), because it shows the schematic effect of applying the rewrite rule. An application to a host graph G_L is included in the middle row.

With respect to the labeling, the example demonstrates how (i) labels in L serve as lower bounds for matching, (ii) labels in L' serve as upper bounds for matching, (iii) labels in K' can be used to decrease matched labels (so in particular, $\bot$ "instructs" to "erase" the label and overwrite it with $\bot$, and $\top$ "instructs" to preserve labels), and (iv) labels in R can be used to increase labels.

4 Embedding Linear Term Rewriting Systems

We are now ready to define an encoding (Definition 1) of linear term rewrite systems into PBPO$^+$. We also show that the encoding is an embedding (Theorem 35). In the next section, we prove that the embedding is globally termination-preserving.

For defining the encoding of terms as graphs, the auxiliary notion of a rooted graph is convenient.

Definition 18 (Rooted Graph)**.** A *rooted graph* (G,r) consists of a graph G and a distinguished *root* $r \in V_G$. We let $\mathrm{graph}((G,r)) = G$ and $\mathrm{root}((G,r)) = r$.

We usually omit $\mathrm{graph}(\dots)$ in places where a non-rooted graph is expected, since confusion is unlikely to occur. In visual depictions, the root r is highlighted in a circle r.

Definition 19 (Term Encoding)**.** Define the flat lattice Σ° for signatures Σ by $\Sigma^\circ = (\Sigma \uplus \mathbb{N}^+)^{\bot,\top}$.

For linear terms $\mathbf{t} \in \mathrm{Ter}(\Sigma,\mathcal{X})$, we define the *term encoding* $\mathbf{t}^\circ$ *of* $\mathbf{t}$ as the Σ°-labeled rooted graph $\mathbf{t}^\circ = \mathcal{E}(\mathbf{t},\varepsilon)$, where $\mathcal{E}(\mathbf{t},p)$ is defined by clauses

$$\mathcal{E}(f(t_1,\dots,t_n),p) = \quad \mathcal{E}(\mathbf{t_1},p1) \;\; {}_1\!\leftarrow\; p^f \;\rightarrow_n \;\; \mathcal{E}(\mathbf{t_n},pn)$$

and $\mathcal{E}(x,p) = (x^{\perp}, x)$ for $f \in \Sigma$, $\mathbf{t_1}, \ldots, \mathbf{t_n} \in \mathrm{Ter}(\Sigma, \mathcal{X})$, $x \in \mathcal{X}$ and $p \in \mathbb{N}^*$. The target of an edge pointing towards a rooted graph (G, p') is p'. In these graphs, the identity of an edge with source p and target p' is (p, p').

Note that the term encoding always results in a tree, because the terms it operates on are linear.

Definition 20 (Positions in Term Encodings)**.** Analogous to positions in terms $\mathbf{t}$ (Definition 6), we assign positions to the nodes of $\mathbf{t}^\circ$: $\mathrm{root}(\mathbf{t}^\circ)$ is assigned position ε; and if $v \xrightarrow{i} w$ is an edge in $E_{\mathbf{t}^\circ}$ (for $i \geq 1$) and v has position p, then w is assigned position pi.

A translated rule ρ° is said to be *applied at position p in* $\mathbf{t}^\circ$ if the match morphism $m : L \rightarrowtail \mathbf{t}^\circ$ maps the root of L onto the vertex with position p in $\mathbf{t}^\circ$, and establishes a match.

The following definition is used in the setting of rule encodings.

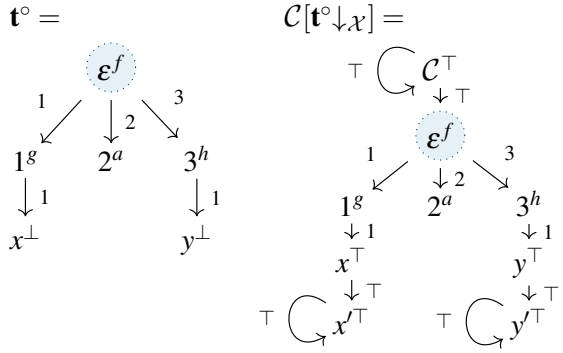
Definition 21 (Context Closures)**.** Let $G_r = (G, r)$ be a rooted graph.

Assume $\mathcal{C} \notin V_G$. The *upper context closure* of G_r, denoted $\mathcal{C}[G_r]$, is the r-rooted graph obtained by adding a $\top$-labeled vertex $\mathcal{C}$ and two $\top$-labeled edges with identities $(\mathcal{C}, r)$ and $(\mathcal{C}, \mathcal{C})$ to G. Sources and targets are given by the first and second projections, respectively.

For $x \in \mathcal{X}$, let x' be fresh for V_G. The *lower context closure* of G w.r.t. a subset $\mathcal{X} \subseteq V_G$, denoted $G{\downarrow_{\mathcal{X}}}$, is the r-rooted graph obtained as follows: for every $x \in V_G \cap \mathcal{X}$, (i) relabel x to $\top$, and (ii) add a $\top$-labeled vertex x' and two $\top$-labeled edges (x, x') and (x', x') to G.

The *context closure* of G_r is defined as $\mathcal{C}[G_r{\downarrow_{\mathcal{X}}}]$.

Example 22. The term encoding $\mathbf{t}^\circ$ of $\mathbf{t} = f(g(x), a, h(y))$ and its context closure $\mathcal{C}[\mathbf{t}^\circ{\downarrow_{\mathcal{X}}}]$ are shown on the right. Both graphs are rooted in ε. (The edge identities are left implicit.)



Definition 23 (Variable Heads and Symbol Vertices)**.** For term encodings $\mathbf{t}^\circ$, the vertices in $x \in V_{\mathbf{t}^\circ} \cap \mathcal{X}$ with $\ell(x) = \perp$ are called *variable heads*, and the remaining vertices labeled from Σ are called *symbol vertices*.

Definition 24 (Interface Graph)**.** The *interface graph* $\mathcal{I}(\mathbf{t})$ for a term $\mathbf{t}$ is the rooted graph (G', ε), where G' is the discrete graph induced by $V_{G'} = \mathrm{Var}(\mathbf{t}) \cup \{\varepsilon\}$ and $\ell^V_{G'}(v) = \perp$ for all $v \in V_{G'}$.

Definition 25 (Rule Encoding)**.** The *rule encoding* ρ° of a linear term rewrite rule $\rho : \mathbf{l} \to \mathbf{r}$ over Σ into a (linear) PBPO$^+$ rewrite rule over Σ°-labeled graphs is defined as follows:

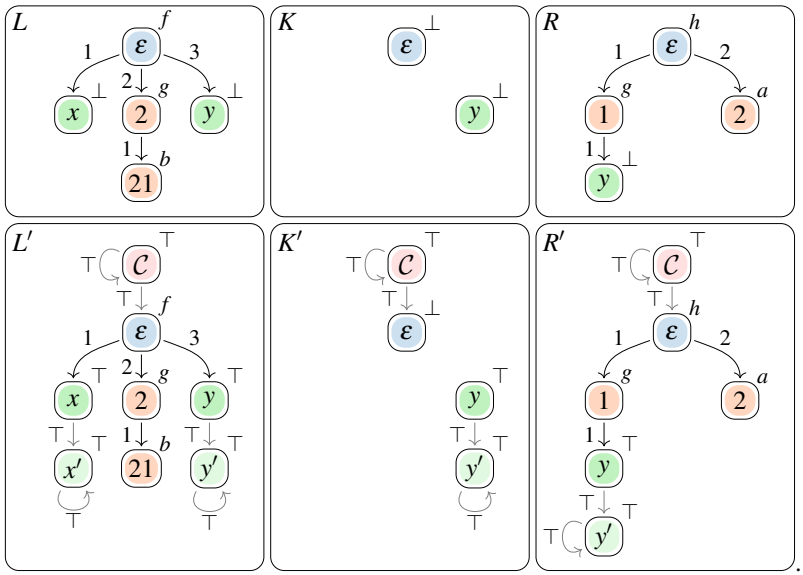
$$L = \mathbf{l}^\circ \qquad K = \mathcal{I}(\mathbf{r}) \qquad R = \mathbf{r}^\circ$$
$$L' = \mathcal{C}[\mathbf{l}^\circ{\downarrow_{\mathcal{X}}}] \qquad K' = \mathcal{C}[\mathcal{I}(\mathbf{r}){\downarrow_{\mathcal{X}}}]$$

Here we implicitly consider the rooted graphs as graphs by forgetting their roots. Each of the morphisms l, r, l', t_L, and t_K map roots to roots and behave as inclusions otherwise.

Observe that the rule encoding accounts for the special case where the right-hand side $\mathbf{r}$ of the TRS rule is a variable x, in which case $r : K \to R$ is the morphism determined by $r(\varepsilon) = r(x) = x$. (The case where the left-hand side $\mathbf{l}$ is a variable is excluded by definition.)

Proposition 26. *In Definition 25, all of the morphisms are well-defined and uniquely determined, and the pullback property is satisified. Moreover, morphisms l, l', t_L and t_K are monic, and r is monic iff* $\mathbf{r}$ *is not a variable.* □

Example 27 (Rule Encoding)**.** The TRS rule $\rho = f(x, g(b), y) \to h(g(y), a)$ is encoded as the PBPO$^+$ rewrite rule ρ° given by



An application of this rule can be thought of as binding head variable x and y of L to the roots of two subterms. These subterms and the context are then uniquely captured by L' (by virtue of the strong match property), and correctly rearranged around R by the rewrite step.

Rule encodings extend to rewrite system encodings in the obvious way.

Definition 28 (Rewrite System Encoding)**.** The *rewrite system encoding* R° of a linear TRS R is $\{\rho^\circ \mid \rho \in R\}$.

All the encodings we have introduced have obvious inverses.

Definition 29 (Decoding)**.** For term/rule/system encodings x°, we define the inverse $\mathit{decode}(x^\circ) = x$.

Proposition 30 (Root Mapping Determines Adherence)**.** *Let $\rho = \mathbf{l} \to \mathbf{r}$ be a linear term rewrite rule. If ρ° is applied at position p in $\mathbf{s}^\circ$, then a unique $\alpha : \mathbf{s}^\circ \to \mathcal{C}[\mathbf{l}^\circ{\downarrow_\mathcal{X}}]$ exists that establishes a strong match, i.e., that makes*

$$\begin{array}{ccc} \mathbf{l}^\circ & \xrightarrow{m} & \mathbf{s}^\circ \\ {\scriptstyle 1_{\mathbf{l}^\circ}}\downarrow & PB & \downarrow{\scriptstyle \alpha} \\ \mathbf{l}^\circ & \xrightarrow{t_L} & \mathcal{C}[\mathbf{l}^\circ{\downarrow_\mathcal{X}}] \end{array}$$

a pullback square.

Proof. By definition of applying at a position p, m maps root$(\mathbf{l}^\circ)$ onto position p of $\mathbf{s}^\circ$, fully determining m to map nodes with position q in $\mathbf{l}^\circ$ onto nodes with position pq in $\mathbf{s}^\circ$. A node in $\mathbf{l}^\circ$ is either a symbol vertex or a variable head. For symbol vertices, any m must preserve labels. Variable heads (labeled with $\bot$) are mapped by m onto either (i) vertex $\sigma(x)$ labeled with $\bot$ if $\sigma(x) \in \mathcal{X}$ is a variable, or (ii) vertex pq labeled with $f \in \Sigma$ if x is substituted for some non-variable term $\sigma(x) = f(\mathbf{t_1}, \ldots, \mathbf{t_n})$ $(n \geq 0)$.

On the image $m(\mathbf{l}^\circ)$, define α such that $t_L = \alpha \circ m$. The labels of symbol vertices are thereby preserved, and the labels in the head variables of $\mathbf{l}^\circ$ are increased to $\top$. The elements not in $m(\mathbf{l}^\circ)$ can be mapped onto the appropriate elements added by the context closure, and only in one way as to not overlap with t_L. Because t_L does not map onto these closures, pulling α along t_L gives the required pullback square. □

Lemma 31 (Match Determinism)**.** *Let $\rho = \mathbf{l} \to \mathbf{r}$ be a linear term rewrite rule. If ρ° is applied at position p in $\mathbf{s}^\circ$ and gives rise to a step $\mathbf{s}^\circ \to G$, then G is uniquely determined up to isomorphism.*

Proof. By Proposition 30, adherence α is completely determined, and by general categorial properties, the pullback of α along l' gives a unique result up to isomorphism, and so does the final pushout. □

Proposition 32. *If $m : \mathbf{l}^\circ \rightarrowtail \mathbf{s}^\circ$ is a mono, then $\mathbf{s}^\circ = (C[\mathbf{l}\sigma])^\circ$ for some context C and substitution σ. Moreover, the position of $m(\mathrm{root}(\mathbf{l}^\circ))$ in $\mathbf{s}^\circ$ equals the position of $\Box$ in $C[\,]$.*

Proof. By monicity of m, the tree structure of $\mathbf{l}^\circ$ is preserved into $\mathbf{s}^\circ$. The labels of symbol vertices and edges are also preserved, since $\mathbf{s}^\circ$ has no occurrences of $\top$. This also means that, for every vertex v of $\mathbf{l}^\circ$, v and $m(v)$ have the same number of outgoing edges, since encodings preserve arities.

A variable head $x \in V_{\mathbf{l}^\circ}$ is mapped onto a vertex $m(x)$, which is either a variable head with label $\bot$, or a symbol vertex labeled with some $f \in \Sigma$ and a subtree underneath.

Let p be the position of $m(\mathrm{root}(\mathbf{l}^\circ))$ in $\mathbf{s}^\circ$. Define C as the context obtained from $\mathbf{s}$ by replacing the subterm at position p by $\Box$. Define the substitution σ, for every $x \in \mathrm{Var}(\mathbf{l})$, by $\sigma(x) = \mathbf{s}|_{pq_x}$ where q_x is the position of x in $\mathbf{l}$. Then the claim follows since m maps x in $\mathbf{l}^\circ$ to the position pq_x in $\mathbf{s}^\circ$, and the subtree rooted at this position is $(\mathbf{s}|_{pq_x})^\circ$. □

Lemma 33 ($(\cdot)^\circ$ Is Step-Preserving)**.** *Let $\rho = \mathbf{l} \to \mathbf{r}$ be a linear term rewrite rule. If $\mathbf{s} \to \mathbf{t}$ via ρ at position p, then $\mathbf{s}^\circ \to \mathbf{t}^\circ$ via ρ° at position p.*

Proof. By the definition of a term rewrite step, $\mathbf{s} = C[\mathbf{l}\sigma]$ and $\mathbf{t} = C[\mathbf{r}\sigma]$ for some context C and substitution σ, and $\mathbf{l}\sigma$ is at position p in $C[\mathbf{l}\sigma]$.

By the definitions of encodings and a PBPO$^+$ rewrite step, we must show that the diagram

$$\begin{array}{ccccc}
 & & \mathcal{I}(\mathbf{r}) & \xrightarrow{r} & \mathbf{r}^\circ \\
 & & \vdots\, !u & & \mathrm{PO}\ \downarrow w \\
\mathbf{l}^\circ & \xrightarrow{m} (C[\mathbf{l}\sigma])^\circ & \xleftarrow{g_L} G_K & \xrightarrow{g_R} & (C[\mathbf{r}\sigma])^\circ \\
1_{\mathbf{l}^\circ}\downarrow\ \mathrm{PB} & \alpha\downarrow & \mathrm{PB}\ \downarrow u' & & \\
\mathbf{l}^\circ & \xrightarrow{t_L} \mathcal{C}[\mathbf{l}^\circ{\downarrow_{\mathcal{X}}}] & \xleftarrow{l'} \mathcal{C}[\mathcal{I}(\mathbf{r}){\downarrow_{\mathcal{X}}}] &
\end{array} \tag{1}$$

holds for some G_K and the various morphisms that are not fixed by ρ° (including α), and where m maps $\mathrm{root}(\mathbf{l}^\circ)$ onto position p of $(C[\mathbf{l}\sigma])^\circ$. Note that g_L is a mono by Proposition 26 and stability of monos under pullbacks, and w is a mono by Proposition 15.

By Proposition 30, m and α exist and they exist uniquely. It is then straightforward to check that the middle pullback extracts the subgraphs corresponding to the context C and to every subterm bound to a variable $x \in \mathrm{Var}(\mathbf{l}) \cap \mathrm{Var}(\mathbf{r})$, and that the pushout performs the appropriate gluing around pattern $\mathbf{r}^\circ$, with $(C[\mathbf{r}\sigma])^\circ$ as the result. □

Lemma 34 ($(\cdot)^\circ$ Is Closed)**.** *Let $\rho = \mathbf{l} \to \mathbf{r}$ be a linear term rewrite rule. If $\mathbf{s}^\circ \to G$ via ρ° then $G \cong \mathbf{t}^\circ$ for some term $\mathbf{t}$ with $\mathbf{s} \to \mathbf{t}$.*

Proof. Assume $\mathbf{s}^\circ \to G$ via ρ° at position p. Then by Proposition 32 we have $\mathbf{s}^\circ = (C[\mathbf{l}\sigma])^\circ$ for some context C and substitution σ such that $\mathbf{s}(p) = \Box$. Then $\mathbf{s} = C[\mathbf{l}\sigma] \to C[\mathbf{r}\sigma] = \mathbf{t}$ via ρ at position p. Thus $\mathbf{s}^\circ \to \mathbf{t}^\circ$ via ρ° at position p by Lemma 33. Then we have $G \cong \mathbf{t}^\circ$ by Lemma 31. □

Theorem 35. *The encoding $(\cdot)^\circ$ is an embedding.*

Proof. From Lemma 33 and Lemma 34. □

5 The Embedding Preserves Termination Globally

From the fact that the encoding is step-preserving (Lemma 33), the following is almost immediate.

Lemma 36. *Let R be a linear TRS. If R° is terminating on $\mathbf{FinGraph}^{\Sigma^\circ}$, then R is terminating.* □

It is obvious that the other direction holds if the category $\mathbf{FinGraph}^{\Sigma^\circ}$ is restricted to graphs that are term encodings; so we have *local termination* [9–11]. However, in this subsection we will show that the direction holds *globally*. Thus, in particular, the finite graphs may be disconnected, cyclic, and labeled arbitrarily from Σ°.

Our overall proof strategy is as follows. First, we show that it suffices to restrict to cycle-free graphs G (Corollary 45). Then, we show that an infinite rewrite sequence on cycle-free G contains (in some sense) an infinite rewrite sequence on term encodings, and therefore on terms (Theorem 62).

Definition 37 (Undirected Path)**.** Let $n \in \mathbb{N}$. An *undirected path* of length n from node v_1 to v_{n+1} in a graph G is a sequence $v_1\, e_1\, v_2\, v_2 \cdots v_n\, e_n\, v_{n+1}$ where $v_1, v_2, \ldots, v_{n+1}$ are nodes of G and $e_1, e_2, e_3, \ldots, e_n$ are edges of G such that $(v_i, v_{i+1}) \in \{\, (s(e_i), t(e_i)),\ (t(e_i), s(e_i)) \,\}$ for every $1 \le i \le n$.

The path is an *undirected cycle* if moreover $n > 0$, $v_1 = v_{n+1}$ and $e_i \neq e_j$ for all $0 < i < j \le n$. A *cycle edge* (*cycle node*) is an edge (node) that is part of an undirected cycle. A graph is *cycle-free* if it does not contain undirected cycles.

Example 38. A path of length 1 is an undirected cycle iff its only edge e is a loop, that is, $s(e) = t(e)$. Two edges between two nodes always constitute an undirected cycle of length 2 (irrespective of the direction of the edges).

Proposition 39. *Edge e is a cycle edge iff there exists an undirected path from $s(e)$ to $t(e)$ that does not include e.*

Proof. If $s(e) = t(e)$, one path is the empty path. Obvious otherwise. □

Proposition 40. *If e is a cycle edge in G and $\phi : G \rightarrowtail H$ a mono, then $\phi(e)$ is a cycle edge in H.* □

Although monos preserve the cycle edge property, morphisms do not generally do so (consider a morphism that identifies two parallel edges). However, for adherence morphisms α we have the following result.

Lemma 41. *Consider the PBPO$^+$ match square (the leftmost square of the rewrite step diagram) with a host graph G_L. Suppose that e is a cycle edge in G_L and $\alpha(e) = t_L(e')$ for some $e' \in E_L$. Then $\alpha(e)$ is a cycle edge in L'.*

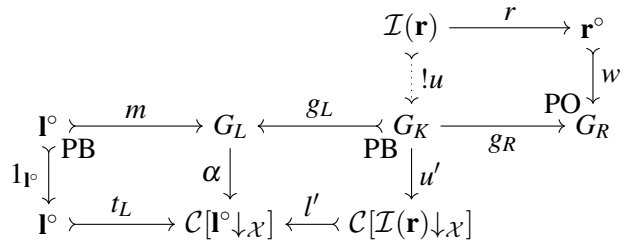
Proof. Let σ_1 be the path just consisting of e. By Proposition 39 there also exists an undirected path from $s(e)$ to $t(e)$ in G_L that does not include e. Since premorphisms preserve undirected paths, $\alpha(\sigma_1)$ and $\alpha(\sigma_2)$ are undirected paths from $\alpha(s(e))$ to $\alpha(t(e))$ in L'. If $\alpha(e)$ is not a cycle edge, then paths $\alpha(\sigma_1)$ and $\alpha(\sigma_2)$ both include $\alpha(e)$ by Proposition 39. Thus α maps two distinct edges in G_L onto $\alpha(e) = t_L(e')$. Since L is the α-preimage of $t_L(L)$, $\alpha \circ m = t_L$ also maps two distinct edges onto $t_L(e)$. This contradicts that t_L is monic. So $\alpha(e)$ is a cycle edge. □

Lemma 42 (Cycle-Preserving Pullback)**.** *If for $\tau = G \xrightarrow{g} X \xleftarrow{h} H$, (i) σ is an undirected cycle in G, (ii) $g(\sigma)$ lies in the image of h, and (iii) the pullback for τ is $G \xleftarrow{g'} Y \xrightarrow{h'} H$, then every edge $e \in g'^{-1}(\sigma)$ is a cycle edge in Y.* □

Definition 43 (Cycle Edge Removal)**.** For a graph G, we let $[G]$ denote the graph obtained by deleting all cycle edges from G.

Lemma 44. *Let $\rho : \mathbf{l} \to \mathbf{r}$ be a linear term rewrite rule over Σ. If there is a rewrite step $G_L \xrightarrow{\rho^\circ} G_R$ on graphs over Σ°, then also $[G_L] \xrightarrow{\rho^\circ} [G_R]$.*

Proof. By the definition of a rewrite step and substituting for the translation of ρ, we have the following arrangement of objects and morphisms



for some G_K. Many of the morphisms are fixed by the rule ρ°. Note that g_L is a mono by Proposition 26 and stability of monos under pullbacks.

Observe that $t_L(\mathbf{l}^\circ)$ does not contain cycle edges (Definition 25). Hence by Lemma 41, α must map every cycle edge of G_L into one of the edges created by constructing the context closure $\mathcal{C}[\mathbf{l}^\circ\downarrow_\mathcal{X}]$ of $\mathbf{l}^\circ$.

Now suppose that we replace G_L by $[G_L]$ in the diagram. Then the middle pullback object G'_K is obtained by removing from G_K the set of edges $C \subseteq E_{G_K}$ that mono g_L maps into a cycle edge of G_L. Since monos preserve cycle edges, every cycle edge of G_K is in C. Moreover, using Lemma 42, C contains only cycle edges. Hence $G'_K = [G_K]$.

Similarly, the pushout object replacement G'_R for G_R is obtained by removing from G_R the set of edges $C \subseteq E_{G_R}$ that have a cycle edge g_R-preimage in G_K. Since an undirected path ρ in G_R is an undirected cycle iff ρ is in the range of g_R and $g_R^{-1}(\rho)$ is an undirected cycle, $G'_R = [G_R]$. □

As a direct consequence of Lemma 44 we obtain the following.

Corollary 45. *Let R be a linear TRS over Σ. R° admits an infinite rewrite sequence on all graphs iff R° admits an infinite rewrite sequence on cycle-free graphs.* □

Thus, in order to prove that termination of R implies termination of R° in $\mathbf{FinGraph}^{\Sigma^\circ}$, it suffices to restrict attention to finite, cycle-free graphs. However, not all such graphs are term-like: graphs may be arbitrarily labeled from Σ°, non-rooted and disconnected. So a further argument is needed.

Definition 46 (Well-Formedness)**.** Let Σ be a signature, and G a graph with labels from Σ°. A node $v \in V_G$ with label $l \in \Sigma \cup \{\bot, \top\} \cup \mathbb{N}^+$ is *in-well-formed* (I) if it has at most one incoming edge; and it is *out-well-formed* (O) if $l \in \Sigma$, and v has precisely $\#l$ outgoing edges, labeled with $1, 2, \ldots, \#l$.

Definition 47 (Good and Bad Nodes)**.** A node $v \in V_G$ is called *good* if v is O and all of v's children in G are I. Nodes that are not good are *bad*.

We will use the distinction between good and bad nodes to define a kind of partitioning on graphs G, which we call a *zoning*. For cycle-free graphs, each zone will be seen to correspond to a term encoding in a qualified sense. (Some edges of G will not be part of any zone of G.) Since most results related to zoning hold not only for non-cycle-free graphs, we will use minimal assumptions where possible (in particular, note that (directed) acyclicity is a weaker condition than cycle-freeness). We do assume finiteness globally.

Definition 48 (Zoning)**.** A *zoning* of G divides up G into *zones*, which are subgraphs of G. The zoning is iteratively constructed as follows:

- Initially, every node of G forms its own zone.
- At each subsequent iteration, if an edge e is not included in a zone and $s(e)$ is good, join the zones of $s(e)$ and $t(e)$ along e. (If $s(e)$ and $t(e)$ are in the same zone Z, this is the same as adding e to zone Z.)
- The algorithm terminates if the previous step can no longer be applied.

Definition 49 (Bridge)**.** A *bridge* is an edge $e \in E_G$ not included in any zone of G.

Proposition 50. *The zoning of a graph G is unique, and any zone is a connected subgraph.* □

Proposition 51. *If $e \in E_Z$ is included in zone Z, then $t(e)$ is* I.

Proof. Since e was joined along, $s(e)$ is good, and hence $t(e)$ is I. □

Definition 52 (Root)**.** A node $v \in V_Z$ without a parent inside zone Z is called a *root* for Z.

Proposition 53. *Within a zone Z, for any two nodes $u, v \in V_Z$, there is a node $x \in V_Z$ such that $u \leftarrow^* x \rightarrow^* v$ (using edges included in Z).*

Proof. Because any zone is connected, there is an undirected path between u and v within Z. This path cannot contain a segment of the form $a \rightarrow c \leftarrow b$, for then c would not be I, contradicting Proposition 51. Hence the path must be of the form $u \leftarrow^* x \rightarrow^* v$ for some $x \in V_Z$. □

Corollary 54. *If a zone has a root, it is unique.*

Proposition 55. *If a zone Z is acyclic, it has a root.*

Proof. If not, following the edges in Z backwards would reveal a directed cycle in Z. □

Proposition 56. *If a zone Z is acyclic, then Z is a directed tree.*

Proof. As follows from the preceding propositions, Z is connected and each zone has a unique root u.

By Proposition 53, u has a path to every node v in Z. Such an (acyclic) path is moreover unique, for otherwise the first point at which these paths join is not I, contradicting Proposition 51. Thus Z is a directed tree. □

We also have the following general characterization of bridges.

Proposition 57 (On Bridges)**.** *The source of a bridge is a bad leaf of a zone, and the target of a bridge is a root of a zone.*

Proof. If e is a bridge, $s(e)$ must be bad. If $s(e)$ is bad, none of its outgoing edges have been joined along. Hence $s(e)$ is a leaf in Z.

If a bridge e targets a non-root $t(e)$ of a zone Z, then $t(e)$ is not I, since it has at least two incoming edges. Thus the parent p of $t(e)$ inside Z is bad. But this contradicts that p must be good since it has an edge to $t(e)$ inside Z. Hence $t(e)$ must be a root. □

Although acyclic zones are directed trees, not every zone corresponds directly to a term encoding $\mathbf{t}^\circ$ for some term $\mathbf{t}$. For instance, for the 3-zone graph $f \xrightarrow{1} a \xleftarrow{1} f$, with $\#(f) = 1$ and $\#(a) = 0$, only the zone containing the node labeled with a corresponds to a term encoding. But we have the following result.

Proposition 58 (Zones as Term Encodings)**.** *If every bad node of an acyclic zone Z is relabeled with $\bot$, then Z is isomorphic to a term encoding $\mathbf{t}^\circ$.*

Proof. Every acyclic zone is structurally a directed tree. All inner nodes (and some leaves labeled with constants $a \in \Sigma$) are good, meaning they are labeled with Σ and out-well-formed; and all of their children are in-well formed and included into the zone by the zoning algorithm. Since bad nodes are leaves, relabeling them with $\bot$ essentially makes them represent variables. To establish an isomorphism between a zone and a term encoding, one simply has to rename the identity of every good node to its position in this tree, and the identity of every bad node to some unique $x \in \mathcal{X}$. □

We will now show that relabeling bad nodes with $\bot$ does not meaningfully affect the rewriting behavior in a graph G. Intuitively, this is because matches cannot cross zones, as shown by the following results. Recall the terminology of Definition 23.

Lemma 59. *A match morphism $m : \mathbf{l}^\circ \to G$ (for a rule encoding ρ°) maps symbol vertices $v \in V_{\mathbf{l}^\circ}$ onto good nodes.*

Proof. We must show that $m(v)$ is O and that all of $m(v)$'s children are I.

First, we show that $m(v)$ *is* O. Because v is a symbol vertex, $\ell(v) \in \Sigma$. Since morphisms do not decrease labels, either (a) $\ell(v) = \ell(m(v))$, or (b) $\ell(v) < \ell(m(v))$.

In case (a), we must show that $m(v)$ has precisely $\#(\ell(m(v))) = \#(\ell(v))$ outgoing edges labeled with $1, 2, \ldots, \#(\ell(v))$. By monicity of m and the definition of rule encodings, we know that it has these edges at least once. Moreover, $m(v)$ cannot have additional outgoing edges, since these cannot be suitably mapped by α into L' without violating the strong match property.

In case (b), we obtain a contradiction. For note that $t_L : \mathbf{l}^\circ \to \mathcal{C}[\mathbf{l}^\circ{\downarrow_{\mathcal{X}}}]$ preserves labels for nodes labeled from Σ, so that $\ell(t_L(v)) = \ell(v)$. Furthermore, since m enables a rewrite step, $t_L = \alpha \circ m$ and hence $\ell((\alpha \circ m)(v)) = \ell(v)$. This implies that α decreases the label on $m(v)$, which is not allowed by the $\leq$ requirement on morphisms.

Second, we establish that *all of $m(v)$'s children are* I. Observe that for symbol vertices v, all incoming edges of children of $t_L(v)$ (i) have their source in $t_L(v)$ and (ii) are in the image of t_L. For a contradiction, assume a child u of $m(v)$ has multiple incoming edges e, e'. Then using that $\alpha(m(v)) = t_L(v)$ (by the strong match property) and that $\alpha(u)$ is a child of $t_L(v)$, by observation (i) $\alpha(s(e)) = \alpha(s(e'))$. Since there are no parallel edges in L', $\alpha(e) = \alpha(e')$. By (ii) $\alpha(e)$ is in the image of t_L. Thus multiple elements are mapped onto the same element in L'. This violates the strong match property. Contradiction. □

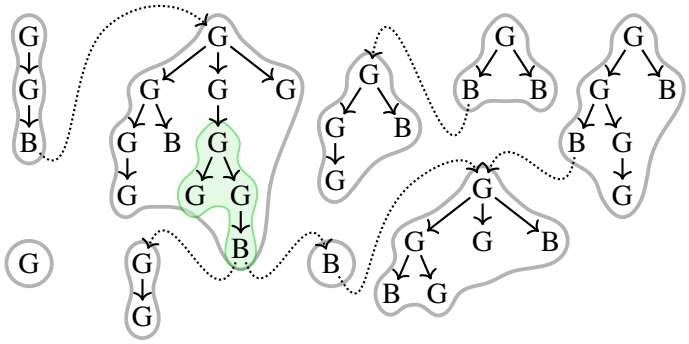


Figure 1: A zoning of a cycle-free graph with three components. Zone borders (gray), good (G) and bad (B) nodes, bridges (dotted) and a match (green) are indicated.

Lemma 60 (Matches Respect Boundaries)**.** *Let $\rho = \mathbf{l} \to \mathbf{r}$ be a TRS rule, and consider the translation ρ°. Then for any match morphism $m : \mathbf{l}^\circ \rightarrowtail G$, the image $m(\mathbf{l}^\circ)$ lies in precisely one zone.*

Proof. Because $\mathbf{l}^\circ$ is connected, so is $m(\mathbf{l}^\circ)$. So if a counterexample to the lemma exists, it involves a bridge. Let $m(e)$, the image of an $e \in E_{\mathbf{l}^\circ}$, be such a bridge. By Proposition 57, $s(m(e)) = m(s(e))$ is a bad leaf of a zone Z. Hence $s(e) \in V_{\mathbf{l}^\circ}$ is a variable head by the contrapositive of Proposition 59. Since variable heads are leaves in $\mathbf{l}^\circ$, this contradicts that $s(e)$ has $e \in E_{\mathbf{l}^\circ}$ for an outgoing edge. □

Figure 1 is an abstract depiction of a zoning, and exemplifies the properties established thus far.

Proposition 61 (Bad Node Labels Are Irrelevant)**.** *Let $G^{[\ell(v):=l]}$ denote the graph obtained by changing the label of $v \in V_G$ to $l \in \mathcal{L}$. If $v \notin V_G$, $G^{[\ell(v):=l]} = G$.*

For bad $v \in V_G$ and any $l \in \mathcal{L}$, if $G \to H$ is a rewrite step via a translated TRS rule ρ° and adherence morphism α, then $G^{[\ell(v):=l]} \to H^{[\ell(v):=l]}$ is a rewrite step via ρ° and α.

Proof. In a rewrite step, bad nodes are either matched by variable heads, or lie outside the image of t_L. In both cases, the label does not influence the application condition, since any label l with $\bot \leq l \leq \top$ is allowed. Moreover, the node is either preserved (and its label unchanged), or deleted. In either case the statement holds. □

Theorem 62. *Let R be a linear TRS. R is terminating on* $\mathrm{Ter}(\Sigma, \mathcal{X})$ *iff R° is terminating on* $\mathbf{FinGraph}^{\Sigma^\circ}$.

Proof. Direction $\Longleftarrow$ is Lemma 36.

For direction $\Longrightarrow$, we prove the contrapositive. By Corollary 45, we may assume G is cycle-free, and thus acyclic. So suppose R° admits an infinite rewrite sequence $\tau_G = G \to G' \to \cdots$ rooted in a cycle-free, finite graph G.

Because matches respect zone boundaries, the number of zones is finite, and zones are never created by rewrite steps, there exists a zone Z of G in which a match is fixed and rewritten infinitely often. This zone is at no point affected by matches in other zones, since zones can only affect other zones by completely deleting them. Similarly, due to cycle-freeness, it is easy to see that the bridges and zones connected to Z do not affect rule applicability in Z. Hence we can restrict G to Z, and construct an infinite rewrite sequence $\tau_Z = Z \to Z' \to Z'' \to \cdots$.

By relabeling every bad node of starting term Z with $\bot$, the existence of an infinite rewrite sequence is not disturbed using Proposition 61. Furthermore, Z is now isomorphic to a term encoding $\mathbf{t}^\circ$ for some term $\mathbf{t}$ (Proposition 58). Using the fact that the encoding is closed (Lemma 34) and that rewriting is defined modulo isomorphism, we can obtain an infinite rewrite sequence on terms. Thus R is also not terminating. □

Remark 63. Our result may be compared to one due to Nolte [15, Chapter 6]. Nolte first defines two encodings of TRSs into term graph rewriting systems, a basic encoding and an extended encoding. These encodings preserve neither termination nor confluence, and are not embeddings. He then shows that for term graph systems obtained by the basic encoding, there exists a globally termination-preserving encoding into graph rewriting systems (DPO) [15, Theorem 6.3]. So although Nolte's approach is similar to ours in spirit, it does not constitute a globally termination-preserving embedding of TRSs into graph rewriting systems.

Remark 64 (Confluence)**.** Although $\Longleftarrow$ of Theorem 62 holds for confluence as well, $\Longrightarrow$ does not, even if graphs are assumed to be connected, cycle-free and well-labeled. Namely, consider $\Sigma = \{f,g,h,a,b\}$ with $\#(f) = \#(g) = \#(h) = 1$ and $\#(a) = \#(b) = 0$, and the confluent TRS $R = \{g(x) \to a, h(x) \to b\}$. Then for the graph $g \overset{1}{\to} f \overset{1}{\to} a \overset{1}{\leftarrow} f \overset{1}{\leftarrow} h$ both a and b are R°-normal forms.

If graphs may be disconnected, rule $g(x) \to a$ even constitutes a counter-example by itself. For the type graph of its rule encoding, a disjoint component H can either be mapped onto the upper context closure (preserving H) or the lower context closure (deleting H).

6 Discussion

We have defined an encoding of linear term rewriting into PBPO$^+$ rewriting that is both an embedding and globally termination-preserving. These properties are achievable because a PBPO$^+$ rule allows (i) specifying where parts of a context may occur around a pattern, (ii) ensuring that these parts are disjoint, and (iii) deleting such parts (in our case study, such parts correspond to variable substitutions).

We submit that a rewriting framework $\mathcal{F}$ can be said to be a proper generalization of some other framework $\mathcal{G}$ if there exists an embedding $\mathcal{E}$ from $\mathcal{G}$ to $\mathcal{F}$. In this sense, PBPO$^+$ is a proper generalization of linear term rewriting (and DPO is not). Often we want the encoding $\mathcal{E}$ to have additional properties such as the global preservation of certain properties (e.g., termination). For instance, the embedding that interprets the TRS rule $\rho = a(b(x)) \to b(a(x))$ as a mere swap of symbols, and thus as applicable in any context, is an embedding that does not preserve termination globally. (Note that such an alternative embedding is also expressible in PBPO$^+$.)

The fact that a certain property-preserving embedding is possible is an interesting expressiveness result for the embedding formalism. Moreover, it opens up a path to reduction arguments, as was also considered by Nolte [15] in a different setting (Remark 63). In our case, if a PBPO$^+$ rewrite system is (isomorphic to) the encoding of a TRS (as defined in Definition 25), termination can be decided by considering the decoded TRS and forgetting about the complexities of graphs. Our proof technique extends to more general PBPO$^+$ rewrite systems as long as the following conditions are met: the pattern of the rules is tree-like (possibly with loops on the nodes of the pattern), the outgoing edges of nodes in the pattern have distinct labels, and the 'context' and 'variable graphs' are disconnected (except through the pattern) and are not duplicated by the rule.

Our provided embedding into PBPO$^+$ does not preserve confluence globally. As shown in Remark 64, the key problem is that an assumption true for terms, namely connectedness, does not hold

for graphs. For the same reason it is currently impossible to define a termination-preserving embedding of non-right-linear term rewriting into PBPO$^+$: whenever a variable is duplicated, it may also lead to the duplication of any number of disjoint components in the graph that are mapped onto the corresponding variable closure. For future work, we intend to investigate extensions of our encoding that do preserve confluence and termination globally even when variables are duplicated.

Adopting a broader perspective, we hope that our encoding contributes to the development of termination techniques for graph rewriting. There have been recent advances in proving termination of graph rewriting; see for instance work by Bruggink et al. [2] and Dershowitz et al. [6]. In [6], recursive path orders are generalized from term rewriting to graph transformation by decomposing the graph into strongly connected components and a well-founded structure between them. A difficulty in this approach is that all possible cycles around the pattern of a rule must be considered. We hope that the technique can be extended to PBPO$^+$ and strengthened by making use of the application conditions that exclude certain cycles around and through the pattern.

Finally, we believe that our result is a step towards modeling other rewriting formalisms such as lambda calculus and higher order rewriting using graph rewriting. These formalisms also rewrite tree structures, and we expect that extensions of our zoning construction will be instrumental for this purpose. Our goal in this respect is to model these systems in such a way that important properties like termination and confluence are preserved globally, while at the same time keeping the modeling overhead minimal (e.g., avoiding auxiliary rules and rewrite steps that increase the length of rewrite sequences).

Acknowledgments

We thank anonymous reviewers for useful suggestions and corrections. Both authors received funding from the Netherlands Organization for Scientific Research (NWO) under the Innovational Research Incentives Scheme Vidi (project. No. VI.Vidi.192.004).

References

[1] S. Awodey (2006): *Category Theory*. Oxford University Press.

[2] H. J. S. Bruggink, B. König, D. Nolte & H. Zantema (2015): *Proving Termination of Graph Transformation Systems Using Weighted Type Graphs over Semirings*. In: *Proc. Conf. on Graph Transformation (ICGT)*, *LNCS* 9151, Springer, pp. 52–68, doi:10.1007/978-3-319-21145-9_4.

[3] A. Corradini, D. Duval, R. Echahed, F. Prost & L. Ribeiro (2015): *AGREE – Algebraic Graph Rewriting with Controlled Embedding*. In: *Proc. Conf. on Graph Transformation (ICGT)*, *LNCS* 9151, Springer, pp. 35–51, doi:10.1007/978-3-319-21145-9_3.

[4] A. Corradini, D. Duval, R. Echahed, F. Prost & L. Ribeiro (2019): *The PBPO Graph Transformation Approach*. *J. Log. Algebraic Methods Program.* 103, pp. 213–231, doi:10.1016/j.jlamp.2018.12.003.

[5] A. Corradini, T. Heindel, F. Hermann & B. König (2006): *Sesqui-Pushout Rewriting*. In: *Proc. Conf. on Graph Transformation (ICGT)*, *LNCS* 4178, Springer, pp. 30–45, doi:10.1007/11841883_4.

[6] N. Dershowitz & J.-P. Jouannaud (2018): *Graph Path Orderings*. In: *Proc. Conf. on Logic for Programming, Artificial Intelligence and Reasoning, (LPAR)*, *EPiC Series in Computing* 57, EasyChair, pp. 307–325, doi:10.29007/6hkk.

[7] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer, doi:10.1007/3-540-31188-2_1.

[8] H. Ehrig, M. Pfender & H. J. Schneider (1973): *Graph-Grammars: An Algebraic Approach*. In: *Proc. Symp. on on Switching and Automata Theory (SWAT)*, IEEE Computer Society, p. 167–180, doi:10.1109/SWAT.1973.11.

[9] J. Endrullis, R.C. de Vrijer & J. Waldmann (2009): *Local Termination*. In: *Proc. Conf. on Rewriting Techniques and Applications (RTA 2009)*, *LNCS* 5595, Springer, pp. 270–284, doi:10.1007/978-3-642-02348-4_-19.

[10] J. Endrullis, R.C. de Vrijer & J. Waldmann (2010): *Local Termination: Theory and Practice*. *Logical Methods in Computer Science* 6(3), doi:10.2168/LMCS-6(3:20)2010.

[11] J. Endrullis & H. Zantema (2015): *Proving Non-termination by Finite Automata*. In: *Proc. Conf. on Rewriting Techniques and Applications (RTA 2015)*, *LIPIcs* 36, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 160–176, doi:10.4230/LIPIcs.RTA.2015.160.

[12] S. Lack & P. Sobociński (2004): *Adhesive Categories*. In: *Proc. Conf. on Foundations of Software Science and Computation Structures (FOSSACS)*, *LNCS* 2987, Springer, pp. 273–288, doi:10.1007/978-3-540-24727-2_20.

[13] M. Löwe (1993): *Algebraic Approach to Single-Pushout Graph Transformation*. *Theor. Comput. Sci.* 109(1&2), pp. 181–224, doi:10.1016/0304-3975(93)90068-5.

[14] S. Mac Lane (1971): *Categories for the Working Mathematician*. 5, Springer Science & Business Media.

[15] D. Nolte (2019): *Analysis and Abstraction of Graph Transformation Systems via Type Graphs*. Ph.D. thesis, University of Duisburg-Essen, Germany. Available at `https://duepublico2.uni-due.de/receive/duepublico_mods_00070359`.

[16] R. Overbeek & J. Endrullis (2020): *Patch Graph Rewriting*. In: *Proc. Conf. on Graph Transformation (ICGT)*, *LNCS* 12150, Springer, pp. 128–145, doi:10.1007/978-3-030-51372-6_8.

[17] R. Overbeek, J. Endrullis & A. Rosset (2021): *Graph Rewriting and Relabeling with PBPO$^+$*. In: *Proc. Conf. on Graph Transformation (ICGT)*, *LNCS* 12741, Springer, doi:10.1007/978-3-030-78946-6_4.

[18] Terese, editor (2003): *Term Rewriting Systems*. *Cambridge Tracts in Theoretical Computer Science* 55, Cambridge University Press.

Stochastic Graph Rewriting For Social Network Modeling

N. Behr
Univ. de Paris, CNRS, IRIF
Paris, France

B.S. Bello
Dept. of Computer Science
Bayero Univ., Kano, Nigeria

S. Ehmes
Real-Time Systems Lab
TU Darmstadt, Germany

R. Heckel*
School of Informatics
Univ. of Leicester, UK

Adaptive networks model social, physical, technical, or biological systems as attributed graphs evolving at the level of both their topology and data. They are naturally described by graph rewriting, but the majority of authors take an approach inspired by the physical sciences, combining an informal description of the operations with programmed simulations, and systems of ODEs as the only abstract mathematical description. We show that we can capture a range of social network models, the so-called voter models, as stochastic attributed graph rewrite systems, demonstrate the benefits of this representation and establish its relation to the non-standard probabilistic view adopted in the literature. We use the theory and tools of graph rewriting to analyze and simulate the models and propose a new variant of a standard stochastic simulation algorithm to recreate the results observed.

1 Introduction

Modeling and analyzing the dynamics of social networks allows scientists to understand the impact of social interactions on areas as diverse as politics (opinion formation, spread of (dis)information), economic development, and health (spread of diseases, update of vaccines) [22, 1, 9]. Much of the more foundational literature approach social network analysis from a perspective informed by statistical physics [21] using a combination of mathematical models (differential equations) and programmed simulation, both derived from an intuitive understanding of the operation of the network. This works well for static networks, where the structure is fixed and changes are to node or link attributes only, but in complex adaptive networks [15] the interconnectedness of structure and data evolution poses additional challenges.

Stochastic typed attributed graph rewriting [16] is an obvious choice to formalize and analyze complex adaptive networks. The formalism provides both tool support for simulation and analysis and an established theory to derive mathematical models from the same rule-based descriptions, thus replacing the informal, and sometimes vague, descriptions in natural language. A case in point are the various voter models [11, 25, 17] which describe opinion formation in a network of agents. The operations, as described in [11] seem clear enough.

> ... we consider two opinions (called 0 and 1) ...; and on each step, we pick a discordant edge (x,y) at random .... With probability $1-\alpha$, the voter at x adopts the opinion of the voter at y. Otherwise (i.e., with probability α), x breaks its connection to y and makes a new connection to a voter chosen at random from those that share its opinion. The process continues until there are no edges connecting voters that disagree.

Intuitively, this is a graph rewrite system of four rules over undirected graphs whose nodes are attributed by 0 or 1, shown below using $\circ$ and $\bullet$, respectively. We adopt the right-to-left notation of rules in line with the theory of rule algebras [8, 2, 7, 6].

*Corresponding author email address: rh122@leicester.ac.uk

The first pair of rules models *rewiring* where an agents disconnects from another one with a different opinion to form a connection with a third agent of the same opinion. In the second pair of rules, an agent connected to one with a different opinion *adopts* the opinion of the other. We could now associate rates with these rules, resulting in a stochastic graph rewrite system that allows simulation using the SimSG tool [12] and the derivation of differential equations using the rule algebra approach [8, 2, 7, 6].

However, on closer inspection we discover a number of discrepancies. First, the model in [11] and related papers is probabilistic, but without time. This is an abstraction of real-world behavior in social networks, where time is not discrete and actions not round-based. Arguably, a continuous-time model is a better representation of this behavior. Semantically, a stochastic graph transformation system induces a continuous-time Markov Chain (CTMC) — an established model with clear links to logics, model checking and simulation techniques, while the operational model behind [11] and others is left informal, but could be formalized as a discrete-time Markov chain (DTMC) or decision process (MDP) [4, 5, 8, 6].

More specifically, looking at the description of the operation, this is a two- or three-step process, where first a conflict edge is selected at random, then a decision made based on the fixed probability α between adopting another opinion, and rewiring which requires another random selection of a node with the same opinion. That means, in the rewiring case, the combined behavior of the operation is not reflected directly by the rewiring rules above, which choose all three nodes first. Also, the race between the rewiring and adopt rules in a stochastic graph rewrite system depends on the number of available matches (the candidates for the 3rd agent to link) while the probability is fixed in the original formulation. Stochastic graph rewriting realizes a mass action semantics which reflects the behavior of physical, chemical and biological processes where the frequency of actions (formally the jump rate of the CTMC) depends on both a rate constant and the concentration or amount available of the input materials required for the action.

In our view, the informal description of operations, lack of continuous time, and non-standard selection and execution procedure all represent weaknesses in the formulation which inhibit a natural mathematical interpretation of the operational behavior of the model and consequently a formal and systematic connection of this operational description with the simulation and equational model.

In this paper, we present an approach addressing those weaknesses by starting from a formal graph rewrite model as operational description. We investigate the links to the existing voter models both analytically, using the rule algebra framework to establish how to translate operations and parameters to stochastic graph rewriting, and experimentally trying to recreate the observed emergent behavior in the literature. Apart from making a convincing claim for the superiority of our methodological approach, this will allow us in the future to both compare new analysis results to existing ones in the literature.

We start by introducing the model as a stochastic graph transformation system in the format accepted by our simulation tool, then study the theory of relating the different CTMC and DTMC semantics to support different possible conversions between models before reporting on the experimental validation of the resulting models through simulation.

2 Voter Models as Stochastic Typed Attributed Graph Rewrite Systems

We model voter networks as instances of the type graph below (Figure 1), where *Group* nodes represent undirected links connecting *Agent* nodes referred to by their *member* edges. For now, the cardinality

of each group is exactly 2, so they are indeed a model of undirected edges (and we are planning to generalize to groups with several members later). We see the voter network as an undirected multi graph,

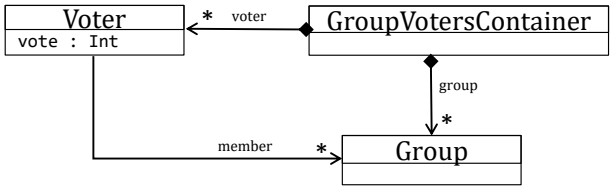


Figure 1: Type graph of the basic voter model

i.e., parallel links between two voters $v1$ and $v2$ are permitted. That means, in our representation, $v1, v2$ can jointly make up one group $g1$ as well as another group $g2$. It is worth noting that the multi-graph interpretation is never explicitly stated in [11] nor the literature building on it. However it is the only interpretation consistent with the usual assumptions made, in particular that rewiring is always possible after the selection of the 3rd node, which may or may not be connected to the first node $v1$ already, and that the total number of edges in the graph is constant, so we have to create a new edge on rewiring even if $v1$ and $v2$ are already connected.

In this paper we use SimSG [12] to simulate rule-based models, such as the basic voter model, for a given start graph and rates according to the semantics of stochastic graph transformations [16]. SimSG implements its own version of Gillespie's well-known algorithm [14] which describes the behavior of stochastic systems over time using a continuous-time Markov Chain with exponentially distributed transition delays. SimSG is build upon the graph transformation rule interpreter eMoflon [20], which provides the means to define and execute rules. For example, there are four rules in the basic voter model, two dual variants each of the *adopt* and *rewire* operations. The graphs in Figure 2 show a visual representation of these four rules as defined in the eMoflon::IBeX-GT[1] syntax. In these rules, black and red elements (--) specify context elements that need to be present in an instance graph, while green elements (++) will be created. A pattern matcher will search for matches in an instance graph that fit these requirements. As shown in Figure 2, in addition to structural constraints, eMoflon allows the specification of attribute conditions. If a match is found, the rule can be applied by deleting all graph elements matching red elements in the rule, and creating new instances of all green elements. In addition to the structural constraints and attribute conditions shown in the example, eMoflon-GT also allows the definition of more complex conditions, such as negative application conditions that filter out matches connected to prohibited graph structures. Beyond the specification of rules, SimSG allows for the annotation of rules with rates, either through value literals or statically evaluated arithmetic expressions. Most importantly, eMoflon provides SimSG with an interface to its underlying incremental graph pattern matching engines, such as Viatra[23], Democles[24] or the recently developed HiPE[2], all of which can be used to find matches for rules and track observable patterns during simulations. During each simulation, SimSG tracks occurrence counts of these patterns and provide the user with the option to plot these counts over the simulation time or save them to a file.

[1] https://emoflon.org/#emoflonIbex
[2] https://github.com/HiPE-DevOps/HiPE-Updatesite

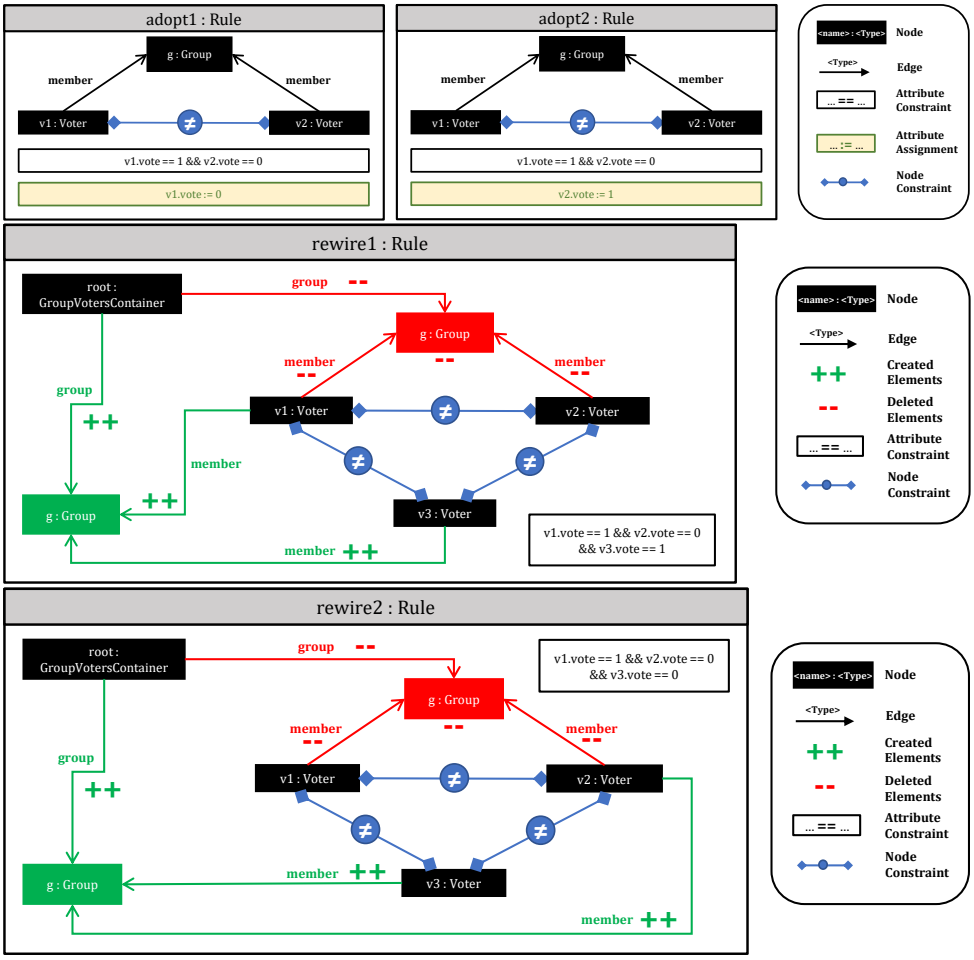


Figure 2: Rules of the basic voter model

3 Theory: CTMCs and DTMCs via Rule-algebraic Methods

Approaches to social network modeling can be broadly classified by their underlying semantics. In this paper, we consider the two major classes consisting of *discrete-time Markov chains (DTMCs)* and *continuous-time Markov Chains (CTMCs)* (leaving for future work the class of Markov decision processes). We will utilize the *rule algebra formalism* [8, 2, 7, 6] as our central technical tool, with CTMCs in so-called mass-action semantics implemented via the *stochastic mechanics framework* [4, 5, 8, 6]. Here, the firing rate of a given rule for a system state at some time $t \geq 0$ is proportional to a *base rate* (i.e., a positive real parameter) times the number of admissible matches of the rule into the system state. Note however that even in the CTMC setting, this semantics is only one of several conceivable variants, and indeed we will study in this paper also a variation of rule-based CTMC semantics wherein rule activities in infinitesimal jumps are not weighted by their numbers of matches. In order to faithfully implement rule-based DTMCs, we will adopt the approach of [3].

3.1 Preliminaries: "compositional" rewriting theory

We will focus throughout this paper on the case[3] of Sesqui-Pushout (SqPO) semantics [10], which under certain conditions on the underlying base categories furnishes a "compositional" rewriting semantics [2], i.e., supports the requisite construction of rule algebras. Since the main theme of this paper is to advocate rewriting-based methods for implementing models, we discuss here the recent extension of the SqPO-formalism to include rewriting rules with conditions and under the influence of constraints [7, 6]. We refer the interested readers to [6] for the analogous formalism in the Double Pushout (DPO) setting.

Consider thus a *base category* $\mathbf{C}$ that is $\mathscr{M}$*-adhesive* (for some class $\mathscr{M}$ of monomorphisms), *finitary*, possesses an $\mathscr{M}$-initial object, $\mathscr{M}$-effective unions and an epi-$\mathscr{M}$-factorization, and such that all final pullback complements (FPCs) along composable pairs of $\mathscr{M}$-morphisms exist (compare [6]). Throughout this paper, we will consider the particular example of the category $\mathbf{uGraph}/T$ (for some $T \in \mathsf{obj}(\mathbf{uGraph})$) of *typed undirected multigraphs*, which according to [6] satisfies all of the aforementioned requirements.

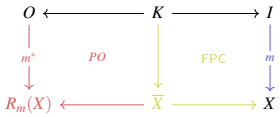
Let us briefly recall the salient points of SqPO-type "compositional" rewriting theory:

Definition 1 (Rewriting rules)**.** Denote by $\overline{\mathsf{Lin}}(\mathbf{C})$ the set of equivalence classes of linear rewriting rules with conditions,

$$\overline{\mathsf{Lin}}(\mathbf{C}) := \{R = (O \xleftarrow{o} K \xrightarrow{i} I, \mathsf{c}_I) | i,o \in \mathscr{M}, \mathsf{c}_I \in \mathsf{cond}(\mathbf{C})\}/\sim, \tag{1}$$

where $R \sim R'$ if and only if $\mathsf{c}_I \doteq \mathsf{c}'_I$ (i.e., if the conditions are equivalent for all matches of the rules) and the rules are isomorphic. The latter entails the existence of isomorphisms $\omega : O \xrightarrow{\cong} O'$, $\kappa : K \xrightarrow{\cong} K'$ and $\iota : I \xrightarrow{\cong} I'$ such that the obvious diagram commutes. We will adopt the convention to speak of "a" rule $R \in \overline{\mathsf{Lin}}(\mathbf{C})$ to mean an equivalence class of rules.

Definition 2. Given a rule $R \in \overline{\mathsf{Lin}}(\mathbf{C})$ and an object $X \in \mathsf{obj}(\mathbf{C})$, an *SqPO-admissible match* $m \in \mathsf{M}_R(X)$ is an $\mathscr{M}$-morphism $m : I \to X$ that satisfies the application condition c_I. Then an *SqPO-type direct derivation* of X along R with a match $m \in \mathsf{M}_R(X)$ is defined via a commutative diagram of the form



(2)

where the object $R_m(X)$ is defined up to the universal isomorphisms of FPCs and pushouts (POs).

Note that in all of our applications, we will only consider objects up to isomorphisms, so we will in a slight abuse of notations speak of taking direct derivations of iso-classes $X \in \mathsf{obj}(\mathbf{C})_{\cong}$ along (equivalence classes of) rules, understood as a mapping from iso-classes to iso-classes. With the focus of the present paper upon implementation strategies, we forgo here a complete review of the concepts of compositional rewriting theory (see e.g. [7, 6]), and limit ourselves to the following salient points. From hereon, $\mathbf{C}$ will always denote a category suitable for SqPO-type rewriting.

Definition 3. Let $\mathscr{R}_{\mathbf{C}}$ denote the $\mathbb{R}$-*vector space "over rules"* spanned by basis vectors $\delta(R)$, where $\delta : \overline{\mathsf{Lin}}(\mathbf{C}) \xrightarrow{\cong} \mathsf{basis}(\mathscr{R}_{\mathbf{C}})$ is an isomorphism. Let $\hat{\mathbf{C}}$ denote the $\mathbb{R}$-*vector space "over states"*, which is defined via the isomorphism $|.\rangle : \mathsf{obj}(\mathbf{C})_{\cong} \to \mathsf{basis}(\hat{\mathbf{C}})$ from isomorphism-classes of objects to basis

[3] Since the rewriting rules in our concrete models are *linear* and will not involve vertex deletions nor creations, we could have equivalently opted for DPO- or SPO-semantics, which are well-known to coincide with SqPO-semantics in this special situation.

vectors of $\hat{\mathbf{C}}$. We introduce the notation $End_{\mathbb{R}}(\hat{\mathbf{C}})$ to denote the space of *endomorphisms* on $\hat{\mathbf{C}}$ (i.e., of linear operators from $\hat{\mathbf{C}}$ to itself). Then we denote by $\rho_{\mathbf{C}} : \mathscr{R}_{\mathbf{C}} \to End_{\mathbb{R}}(\hat{\mathbf{C}})$ the morphism defined via

$$\forall R \in \overline{\mathsf{Lin}}(\mathbf{C}), X \in \mathsf{obj}(\mathbf{C})_{\cong} : \quad \rho_{\mathbf{C}}(\delta(R))\,|X\rangle := \sum_{m \in \mathsf{M}_R(X)} |R_m(X)\rangle\,. \tag{3}$$

The above definition extends to arbitrary elements of $\mathscr{R}_{\mathbf{C}}$ and $\hat{\mathbf{C}}$ by *linearity*, i.e., for $A = \sum_{j=1}^{M} \alpha_j \delta(R_j)$ and $|\Psi\rangle := \sum_X \psi_X\,|X\rangle$, we define $\rho_{\mathbf{C}}(A)\,|X\rangle := \sum_{j=1}^{M} \alpha_j \sum_X \psi_X\,\rho_{\mathbf{C}}(\delta(R_j))\,|X\rangle$.

Intuitively, $\rho_{\mathbf{C}}$ takes a rule vector in $\mathscr{R}_{\mathbf{C}}$ and delivers a transformation between state vectors, applying the rules to the input states, and "weighing" the resulting states according to the coefficients in the rule vector. This concept is useful in particular in order to formalize the jumps of a CTMC (see Section 3.2).

Theorem 1 (Cf. [6]). *(i) The* trivial rule $R_{\varnothing} := [(\varnothing \leftarrow \varnothing \rightarrow \varnothing, \mathsf{true})]_{\sim} \in \overline{\mathsf{Lin}}(\mathbf{C})$ *satisfies*

$$\rho_{\mathbf{C}}(\delta(R_{\varnothing})) = Id_{End_{\mathbb{R}}(\hat{\mathbf{C}})}\,. \tag{4}$$

(ii) Denote by $\langle| : \hat{\mathbf{C}} \to \mathbb{R}$ *the so-called* (dual) projection vector*, defined via* $\langle\,|X\rangle := 1_{\mathbb{R}}$ *for all* $X \in \mathsf{obj}(\mathbf{C})_{\cong}$*. Extending this definition by linearity,* $\langle|$ *thus implements the operation of* summing coefficients*, i.e., for* $|\Psi\rangle = \sum_X \psi_X\,|X\rangle$*, we define* $\langle\,|\Psi\rangle := \sum_X \psi_X\,\langle\,|X\rangle = \sum_X \psi_X$*. Then* $\rho_{\mathbf{C}}$ *satisfies the so-called* ***SqPO-type jump-closure property****:*

$$\forall R \in \overline{\mathsf{Lin}}(\mathbf{C}) : \quad \langle|\,\rho_{\mathbf{C}}(\delta(R)) = \langle|\,\hat{\mathbb{O}}(\delta(R))\,, \quad \hat{\mathbb{O}}(\delta(R)) := \rho_{\mathbf{C}}(\delta([I \leftarrow I \rightarrow I, \mathsf{c}_I]_{\sim}))\,. \tag{5}$$

The last point of the above theorem alludes to the important special case of rule-based linear operators that are *diagonal* in the basis of $\hat{\mathbf{C}}$, since these implement operations of *counting patterns*. In the particular case at hand, we may combine the definition of $\rho_{\mathbf{C}}$ in (3) with (5) in order to obtain

$$\langle|\,\hat{\mathbb{O}}(\delta(R))\,|X\rangle = \langle|\,\rho_{\mathbf{C}}(\delta(R))\,|X\rangle = \sum_{m \in \mathsf{M}_R(X)} \langle\,|R_m(X)\rangle = \sum_{m \in \mathsf{M}_R(X)} 1_{\mathbb{R}} = |\mathsf{M}_R(X)|\,. \tag{6}$$

In this sense, $\hat{\mathbb{O}}(\delta(R))$ as defined in (5) "counts" the numbers of admissible matches of R into objects.

3.2 Rule-based CTMCs

Traditionally, CTMCs based upon rewriting rules have been defined in so-called *mass-action semantics*. In the rule algebra formalism, such types of CTMCs are compactly expressed as follows (with $\rho \equiv \rho_{\mathbf{C}}$):

Theorem 2 (Mass-action semantics for CTMCs). *Let* $\mathscr{T} := \{(\kappa_j, R_j)\}_{j=1}^{N}$ *denote a (finite) set of pairs of* base rates $\kappa_j \in \mathbb{R}_{>0}$ *and rules* $R_j \in \overline{\mathsf{Lin}}(\mathbf{C})$ *(for* $j = 1, \ldots, N$*) over some category* $\mathbf{C}$*. Denote by* $\mathsf{Prob}(\hat{\mathbf{C}})$ *the space of (sub-)probability distributions over the state-space* $\hat{\mathbf{C}}$*, and choose an* initial state $|\Psi_0\rangle := \sum_X \psi_X^{(0)}\,|X\rangle \in \mathsf{Prob}(\hat{\mathbf{C}})$*. Then the data* $(\mathscr{T}, |\Psi_0\rangle)$ *defines a* ***SqPO-type rule-based CTMC*** *via the following* ***evolution equation*** *for the* ***system state*** $|\Psi(t)\rangle := \sum_X \psi_X(t)\,|X\rangle$ *at time* $t \geq 0$*:*

$$\tfrac{d}{dt}\,|\Psi(t)\rangle = H\,|\Psi(t)\rangle\,, \quad |\Psi(0)\rangle = |\Psi_0\rangle\,, \quad H := \rho(h) - \hat{\mathbb{O}}(h)\,, \quad h := \sum_{j=1}^{N} \kappa_j \delta(R_j) \tag{7}$$

Here, we used linearity to extend the definition of the jump-closure operator $\hat{\mathbb{O}}(.)$ *of* (5) *to arbitrary elements of* $\mathscr{R}_{\mathbf{C}}$*, i.e.,* $\hat{\mathbb{O}}(h) := \sum_{j=1}^{N} \kappa_j \hat{\mathbb{O}}(\delta(R_j))$*.*

However, while mass-action semantics is of key importance in the modeling of chemical reaction systems, it is of course not the only conceivable semantics. In particular, as we shall demonstrate in the later part of this paper, for certain applications it will prove useful to utilize alternative adjustments of the "firing rates" of individual rules other than the one fixed in mass-action semantics.

Definition 4 (Generalized rule-based CTMC semantics)**.** For a suitable category $\mathbf{C}$, let $\mathscr{T} := \{(\gamma_j, R_j, W_j)\}_{j=1}^N$ be a set of triples of *base rates* $\gamma_j \in \mathbb{R}_{>0}$, *rewriting rules* $R_j \in \overline{\mathsf{Lin}}(\mathbf{C})$ and *(inverse) weight functions* $W_j \in End(\hat{\mathbf{C}})_{diag}$ $(j = 1,\ldots,N)$. Here, $End(\hat{\mathbf{C}})_{diag}$ denotes the space of *diagonal operators* (with respect to the basis of $\hat{\mathbf{C}}$). Together with a choice of *initial state* $|\Psi_0\rangle \in \mathsf{Prob}(\hat{\mathbf{C}})$, this data defines a *rule-based SqPO-type CTMC* via the *evolution equation*

$$\begin{aligned} &\tfrac{d}{dt}|\Psi(t)\rangle = H_{\underline{W}}|\Psi(t)\rangle\,, \quad |\Psi(0)\rangle = |\Psi_0\rangle\,, \quad H_{\underline{W}} := \sum_{j=1}^N \gamma_j \left(\rho(\delta(R)_j) - \hat{\mathbb{O}}(\delta(R_j))\right)\frac{1}{W_j^*} \\ &\forall F \in End(\hat{\mathbf{C}})_{diag}, |X\rangle \in \hat{\mathbf{C}}: \quad \frac{1}{F^*}|X\rangle := \begin{cases} |X\rangle & \text{if } F|X\rangle = 0_{\mathbb{R}} \\ \frac{1}{\langle |F|X\rangle}|X\rangle & \text{otherwise.} \end{cases} \end{aligned} \tag{8}$$

Example 1. Given a set of pairs of base rates and rules $\mathscr{T} = \{(\kappa_j, R_j)\}_{j=1}^N$ for a rule-based CTMC with mass-action semantics as in Definition 2, one may define from this transition set in particular a *uniformed* CTMC with infinitesimal generator H_U defined as follows (with $\rho \equiv \rho_{\mathbf{C}}$):

$$H_U := -Id_{End(\hat{\mathbf{C}})} + \sum_{j=1}^N p_j \cdot \rho(\delta(R_j))\frac{1}{\hat{\mathbb{O}}(\delta(R_j))^*}\,, \quad p_j := \frac{\kappa_j}{\sum_{j=1}^N \kappa_j}\,. \tag{9}$$

Note in particular that since all base rates κ_j are positive real numbers, the parameters p_j are *probabilities*. Moreover, applying H_U to an arbitrary basis state $|X\rangle$, one finds that the overall jump rate (i.e., minus the coefficient of the diagonal part of H_U) is precisely equal to 1, so that the p_j in the non-diagonal part of H_U encodes in fact the probability for the "firing" of rule R_j, regardless of the numbers of admissible matches of R_j into the given state $|X\rangle$. Finally, under the assumption that both the mass-action and the uniformed CTMC admit a *steady state*, we find by construction that

$$\lim_{t\to\infty} \tfrac{d}{dt}|\Psi(t)\rangle = 0 \quad \Leftrightarrow \quad \lim_{t\to\infty} H|\Psi(t)\rangle = 0 \quad \Leftrightarrow \quad \lim_{t\to\infty} H_U\left(\hat{\mathbb{O}}(h)\cdot|\Psi(t)\rangle\right) = 0\,. \tag{10}$$

Letting $|\Psi_U(t)\rangle$ denote the system state at time t of the "uniformed" CTMC with generator H_U, and assuming $|\Psi(0)\rangle = |\Psi_U(0)\rangle$, the above entails in particular that $|\Psi_U(t\to\infty)\rangle = (\hat{\mathbb{O}}(h)^*)^{-1}\cdot|\Psi(t\to\infty)\rangle$, i.e., the steady state of the CTMC generated by H_U and the one of the CTMC generated by H are related by an operator-valued rescaling (via the operator $(\hat{\mathbb{O}}(h)^*)^{-1}$, and thus in general in non-constant fashion). A similar construction will play a key role when studying rule-based discrete-time Markov chains.

Another interesting class of examples is motivated via the "Potsdam approach" to probabilistic graph transformation systems as advocated in [19].

Example 2 (LCM-construction)**.** Given a generalized CTMC with infinitesimal generator $H_{\underline{W}}$ specified as in Definition 4, construct the least common multiple (LCM) L of the inverse weight functions:

$$L := LCM(W_1,\ldots,W_N)\,. \tag{11}$$

Then we define the *LCM-variant* of the CTMC as

$$H_{LCM} := H_{\underline{W}}\cdot L\,. \tag{12}$$

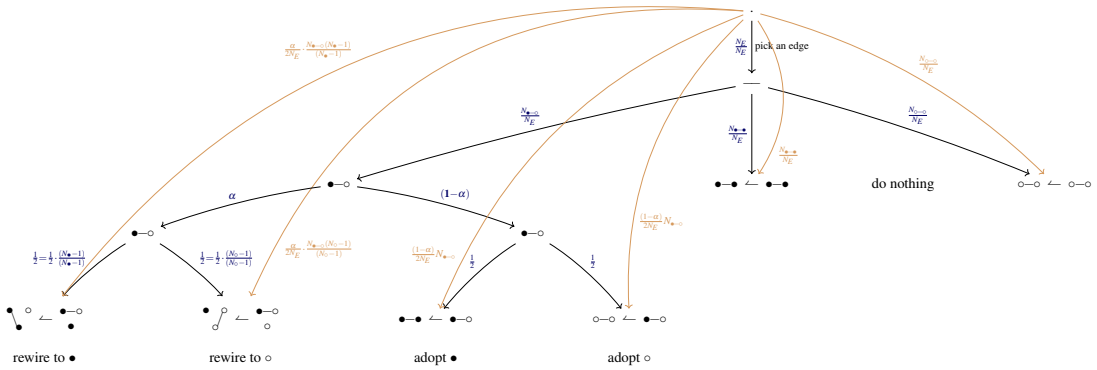


Figure 3: Specification of the adaptive voter model (AVM) according to [11] via a probabilistic decision tree (black arrows), and as combined one-step probabilistic transitions (orange arrows).

By construction, H_{LCM} does not contain any operator-valued inverse weights (in contrast to the original $H_{\underline{W}}$). Moreover, for certain choices of (inverse) weight functions $W_1,\ldots,W_N$, the LCM-construction results in an infinitesimal generator H_{LCM} in which all contributing rules have the same input motif I, thus making contact with the methodology of [19].

3.3 Rule-based DTMCs

Despite their numerous applications in many different research fields, to date discrete-time Markov chains (DTMCs) that are based upon notions of probabilistic rewriting systems have not been considered in quite a comparable detail as the CTMC constructions. Following [3], we present here a possible general construction of rule-based DTMC via a rule-algebraic approach:

Definition 5. For a suitable category $\mathbf{C}$, let $\mathscr{T} = \{(\gamma_j, R_j, W_j)\}_{j=1}^{N}$ be a (finite) set of triples of positive coefficients $\gamma_j \in \mathbb{R}_{>0}$, rewriting rules $R_j \in \overline{\mathsf{Lin}}(\mathbf{C})$ and (inverse) weight functions $W_j \in End(\hat{\mathbf{C}})_{diag}$ ($j = 1,\ldots,N$), with the additional constraint that

$$\sum_{j=1}^{N} \gamma_j \hat{\mathbb{O}}(\delta(R_j)) \cdot \frac{1}{W_j^*} = Id_{End(\hat{\mathbf{C}})} \,. \tag{13}$$

Then together with an initial state $|\Phi_0\rangle \in \mathsf{Prob}(\hat{\mathbf{C}})$, this data defines a *SqPO-type rule-based discrete-time Markov chain (DTMC)*, whose n-th state (for non-negative integer values of n) is given by

$$|\Phi_n\rangle := D^n |\Phi_0\rangle \,, \quad D := \sum_{j=1}^{N} \gamma_j \rho(\delta(R_j)) \cdot \frac{1}{W_j^*} \qquad (\rho \equiv \rho_{\mathbf{C}}) \,. \tag{14}$$

Example 3 (Adaptive Voter Model)**.** As a typical example of a social network model, consider the specification of the *adaptive voter model (AVM)* in the variant according to Durrett et al. [ref], which has as its *input parameters* an *initial graph state* $|\Phi_0\rangle = |X_0\rangle$ (with $N_\bullet^{(0)}$ and $N_\circ^{(0)}$ vertices of types $\bullet$ and $\circ$, respectively, and with N_E undirected edges) and a *probability* $0 \leq \alpha \leq 1$, and whose transitions as depicted in Figure 3 are given via a form of a probabilistic decision procedure in several steps (black

arrows): in each round of the AVM, an edge is chosen at random; if the edge is linking two nodes of different kind, with probability p, one of the rewiring rules is "fired", or with probability $(1-p)$ one of the adopt rules, respectively; otherwise, i.e., if the chosen edge is linking two vertices of the same kind, no action is performed. As annotated in dark blue, each phase of this probabilistic decision procedure is dressed with a probability (and so that all probabilities for a given phase sum to 1). These probabilities in turn depend upon constant parameters (here, p and N_E), as well as on *pattern counts* $N_\bullet$, $N_\circ$, $N_{\bullet\text{–}\bullet}$, $N_{\bullet\text{–}\circ}$ and $N_{\circ\text{–}\circ}$ that dynamically depend on the current system state. Note that $N_V := N_\bullet + N_\circ$ and $N_E = N_{\bullet\text{–}\bullet} + N_{\bullet\text{–}\circ} + N_{\circ\text{–}\circ}$ are *conserved quantities* in this model, since none of the transitions create or delete vertices, but at most exchange vertex types[4], and since the transitions manifestly preserve the overall number of edges.

Upon closer inspection, the transition probabilities may be written in a form that permits to compare the "firing" semantics to the mass-action and to the generalized semantics used in rule-based CTMCs. According to (6), we may implement the operation of counting patterns via linear operators based upon "identity rules", since for arbitrary graph patterns P and graph states $|X\rangle$,

$$N_P(X) = \langle | \, \hat{O}_P \, | X \rangle \,, \quad \hat{O}_P := \rho(\delta([P \leftarrow P \rightarrow P, \mathsf{true}]_\sim)) \,. \tag{15}$$

Since rule algebra theory is not the main topic of the present paper, we provide below some operator relations without derivation[5], noting that they may be computed utilizing the fact that ρ is a so-called representation of the SqPO-type rule algebra (cf. [6, Thm. 3]):

$$\hat{O}_{\bullet\text{–}\circ}\hat{O}_x = \hat{O}_{\bullet\text{–}\circ\, x} + \hat{O}_{\bullet\text{–}\circ} \quad \Leftrightarrow \quad \hat{O}_{\bullet\text{–}\circ\, x} = \hat{O}_{\bullet\text{–}\circ}(\hat{O}_x - 1) \qquad (x \in \{\bullet, \circ\}) \,, \tag{16}$$

which permit us to interpret the AVM model in the following way as a rule-based DTMC: starting by computing the overall one-step transitions and their respective probabilities (orange arrows in Figure 3), and using the above operator relations, we find the rule-based DTMC generator

$$D_{AVM} = \frac{\alpha}{2N_E}\rho\left(\delta\left(\begin{smallmatrix}\bullet & & \circ\\ & \searrow & \\ & \bullet & \end{smallmatrix} \leftarrow \begin{smallmatrix}\bullet\text{–}\circ\\ \bullet\end{smallmatrix}\right)\right)\frac{1}{(\hat{O}_\bullet - 1)^*} + \frac{\alpha}{2N_E}\rho\left(\delta\left(\begin{smallmatrix}\bullet & & \\ & \swarrow & \\ & \circ & \end{smallmatrix} \leftarrow \begin{smallmatrix}\bullet\text{–}\circ\\ \circ\end{smallmatrix}\right)\right)\frac{1}{(\hat{O}_\circ - 1)^*} \tag{17a}$$

$$+ \frac{(1-\alpha)}{2N_E}\rho\left(\delta\left(\bullet\text{–}\bullet \leftarrow \bullet\text{–}\circ\right)\right) + \frac{(1-\alpha)}{2N_E}\rho\left(\delta\left(\circ\text{–}\circ \leftarrow \bullet\text{–}\circ\right)\right) \tag{17b}$$

$$+ \frac{1}{N_E}\rho\left(\delta\left(\bullet\text{–}\bullet \leftarrow \bullet\text{–}\bullet\right)\right) + \frac{1}{N_E}\rho\left(\delta\left(\circ\text{–}\circ \leftarrow \circ\text{–}\circ\right)\right) . \tag{17c}$$

Interestingly, the particular semantics for the "firing" of transitions as encoding in this DTMC generator is neither a variant of mass-action semantics (wherein each rule would fire with a rate dependent on its number of matches, and so that the overall firing rate of all rules would be utilized to normalize all state-dependent firing rates to probabilities as described in Definition 5), nor do all transitions have state-independent, i.e., constant rates. Instead, the *rewiring rules* (17a) have rates proportional to $N_{\bullet\text{–}\circ}$; according to (16), we have that $N_{\bullet\text{–}\circ} = N_{\bullet\text{–}\circ\, x}/(N_X - 1)^*$ (for $x \in \{\bullet, \circ\}$), whence the firing rates of the

[4]Strictly speaking, we implement the two vertex types $\bullet$ and $\circ$ as attributes in our `SimSG` implementation, which in the theoretical setting can be emulated via using self-loops of two different types; evidently, this amounts merely to a slight modification of the type graph plus the enforcement of some structural constraints (compare [6]), thus for notational simplicity, we do not make this technical detail explicit in our diagrams.

[5]In fact, for the simple case at hand, one may derive the relations heuristically, noting that applying $\hat{O}_{\bullet\text{–}\circ}\hat{O}_x$ to some graph state amounts to first counting all vertices of type x (for $x \in \{\bullet, \circ\}$), followed by counting all $\bullet$–$\circ$ edges; performing this operation in one step, this amounts to either counting the x-type vertices separately (i.e., counting the pattern $\bullet$–$\circ$ x), or counting the x-type vertices on the same location as counting the $\bullet$–$\circ$ patterns, thus explaining the two contributions in (16).

rewiring transitions of the AVM model are found to be proportional to a state-dependent fraction of their mass-action rates! In contrast, the *adoption* transitions (17b) as well as the *inactive* transitions (17c) are found to follow standard mass-action semantics, which might raise the question of whether the overall "mixed" semantics chosen for the AVM model indeed reflect the intended intuitions and empirical findings.

Finally, in view of simulation experiments, we would like to advocate the use of dedicated and high-performance Gillespie-style stochastic graph transformation software such as in particular `SimSG` [12] (cf. Section 4). More concretely, one may take advantage of the rule-algebraic formulation of DTMCs and CTMCs to identify for a given DTMC with generator D a particular CTMC based upon the *uniform* infinitesimal generator $H_U := D - Id_{End(\hat{\mathbf{C}})}$, so that (if both limits exist) the two alternative probabilistic systems have the same limit distribution (i.e., $|\Phi_\infty\rangle = \lim_{t\to\infty} |\Psi(t)\rangle$):

$$D\,|\Phi_\infty\rangle = |\Phi_\infty\rangle \quad \Leftrightarrow \quad (D - Id_{End(\hat{\mathbf{C}})})\,|\Phi_\infty\rangle = 0 \quad \leftrightarrow \quad \left(\tfrac{d}{dt}\,|\Psi(t)\rangle\right)\big|_{t\to\infty} = (H_U\,|\Psi(t)\rangle)\big|_{t\to\infty} = 0 \tag{18}$$

However, CTMC simulations for rules that are not following mass-action semantics are non-standard to date, which is why it is also of interest to consider two particular mass-action type alternative CTMC models that are motivated by the original AVM model:

- A "standard" *mass-action semantics (MAS)* CTMC, where the rewriting rules in D_{AVM} are defined to act in mass-action semantics (with the base rates $\kappa_\bullet, \kappa_\circ, \alpha_\bullet, \alpha_\circ \in \mathbb{R}_{>0}$ however not fixed in any evident way from the original AVM model interpretation):

$$H_{MAS} = \kappa_\bullet \cdot \rho\left(\delta\left(\begin{array}{c}\bullet\\ \bullet\end{array}\!\!\begin{array}{c}\diagdown\\ \diagup\end{array}\!\!\begin{array}{c}\circ\\ \bullet\!-\!\circ\end{array} \leftarrow \begin{array}{c}\bullet\!-\!\circ\\ \bullet\end{array}\right)\right) + \kappa_\circ \cdot \rho\left(\delta\left(\begin{array}{c}\bullet\\ \circ\end{array}\!\!\begin{array}{c}\diagup\\ \diagdown\end{array}\!\!\begin{array}{c}\circ\\ \bullet\!-\!\circ\end{array} \leftarrow \begin{array}{c}\bullet\!-\!\circ\\ \circ\end{array}\right)\right) \tag{19a}$$

$$+ \alpha_\bullet \cdot \rho\left(\delta\left(\bullet\!-\!\bullet \leftarrow \bullet\!-\!\circ\right)\right) + \alpha_\circ \cdot \rho\left(\delta\left(\circ\!-\!\circ \leftarrow \bullet\!-\!\circ\right)\right) \tag{19b}$$

$$- \hat{O}_{\bullet-\circ}(\kappa_\bullet(\hat{O}_\bullet - 1) + \kappa_\circ(\hat{O}_\circ - 1) + \alpha_\bullet + \alpha_\circ)\,. \tag{19c}$$

  Note that if we let $\rho(h_{MAS}) := (19a) + (19b)$, then the terms in (19c) are indeed found to be equal to $-\hat{\mathbb{O}}(h_{AVM})$ (utilizing (16) to simplify terms) as required by Theorem 2.

- A *least-common-multiple (LCM)* variant which is computable from D_{AVM} via the method presented in Example 2 (with constant and operator-valued contributions to firing rates in orange):

$$H_{LCM} = \frac{\alpha}{2N_E}\rho\left(\delta\left(\begin{array}{c}\bullet\\ \bullet\end{array}\!\!\begin{array}{c}\diagdown\\ \diagup\end{array}\!\!\begin{array}{c}\circ\\ \bullet\!-\!\circ\end{array} \leftarrow \begin{array}{c}\bullet\!-\!\circ\\ \bullet\end{array}\right)\right)(\hat{O}_\circ - 1) + \frac{\alpha}{2N_E}\rho\left(\delta\left(\begin{array}{c}\bullet\\ \circ\end{array}\!\!\begin{array}{c}\diagup\\ \diagdown\end{array}\!\!\begin{array}{c}\circ\\ \bullet\!-\!\circ\end{array} \leftarrow \begin{array}{c}\bullet\!-\!\circ\\ \circ\end{array}\right)\right)(\hat{O}_\bullet - 1) \tag{20a}$$

$$+ \frac{(1-\alpha)}{2N_E}\rho\left(\delta\left(\bullet\!-\!\bullet \leftarrow \bullet\!-\!\circ\right)\right)(\hat{O}_\bullet - 1)(\hat{O}_\circ - 1) \tag{20b}$$

$$+ \frac{(1-\alpha)}{2N_E}\rho\left(\delta\left(\circ\!-\!\circ \leftarrow \bullet\!-\!\circ\right)\right)(\hat{O}_\bullet - 1)(\hat{O}_\circ - 1)$$

$$- \frac{1}{N_E}\hat{O}_{\bullet-\circ}(\hat{O}_\bullet - 1)(\hat{O}_\circ - 1)\,. \tag{20c}$$

  Expanding terms in H_{LCM} via utilizing once again the representation property of ρ (not presented here for brevity; cf. [6, Thm. 3] for the details), we may write H_{LCM} in the equivalent form below, which exhibits this CTMC model as a particular instance of a model for which all rules have the

same input motif (and thus as a model in the spirit of the "Potsdam approach" as in [19]):

$$H_{LCM} = \frac{\alpha}{2N_E}\rho\left(\delta\left(\right)\right) + \frac{\alpha}{2N_E}\rho\left(\delta\left(\right)\right) \tag{21a}$$

$$+ \frac{1-\alpha}{2N_E}\rho\left(\delta\left(\right)\right) + \frac{1-\alpha}{2N_E}\rho\left(\delta\left(\right)\right) \tag{21b}$$

$$- \frac{1}{N_E}\hat{O}_{\bullet\!-\!\circ\ \bullet\ \circ} \tag{21c}$$

4 Simulating the Models

In Section 3.3 we presented several ways of converting a DTMC-based model into a CTMC-based one. The first (M1) is based on introducing weight factors correcting for the deviation of the probabilistic view of the DTMC from the CTMC mass action semantics. This allows us to use a modified stochastic simulation algorithm to recover the behavior of the DTMC. Then we have the mass action stochastic graph transformation system (M2) as introduced in Section 2, in our view the most natural and closest to reality due to its use of continuous time, but in order to relate to the original probabilistic formulation we have to sample rate constants experimentally until the behavior matches that observed in the given model. Finally we consider a model (M3) obtained by mutually extending the rules of the system to the same left-hand side. Then all rules have the same number of matches and hence rates directly reflect probabilities.

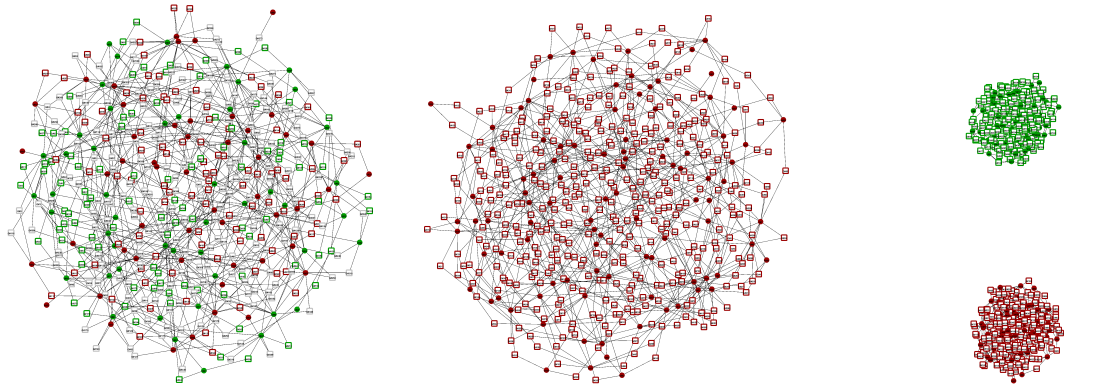
In this section we simulate these models with the CTMC-based SimSG tool [12] to see how, with suitably chosen parameters, their behavior matches that of the original formulation [11]. In particular, we want to answer the following questions.

RQ1: Can we reproduce the results of [11] by simulating (M1) in SimSG?

RQ2: What rates do we have to use to reproduce the behavior of [11] in (M2)?

RQ3: Does (M3) with rates reflecting the probabilities of [11] lead to the same overall behavior?

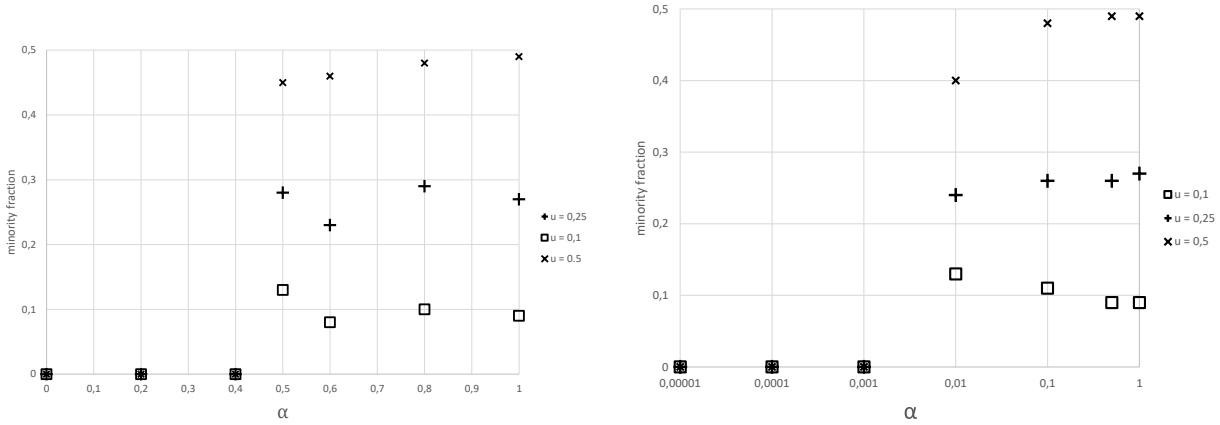
Note that in (M1) and (M2) rates are derived analytically based on the theory in Section 3.3 while they are determined experimentally for (M3). As in [11] initial graphs are generated randomly based on a fraction



(a) Initial graph (u=0.5) (b) Giant component (c) Fragmented graph

Figure 4: Graphs before and after simulation

u of agents voting 0 (where $u = 0.5$ gives a 50/50 split between opinions) by deciding for each pair of nodes with probability v if they should be linked. This means that, with 100 voters and $v = 0.04$, we create an Erdös-Rényi graph with 400 links and average degree 8. According to [11] we expect to see one of two behaviors, depending on the probability α of rewiring (vs. $1-\alpha$ for adoption) for a given discordant link. With $\alpha > 0.43$ rewiring causes a fragmentation of the graph into homogeneous connected components; otherwise adoption leads to the dominance of a single opinion (usually the initial majority) in a giant component. Visually we indicate opinion 1 in green and opinion 0 in red. Homogeneous components are highlighted in their respective color, whereas heterogeneous components are shown in gray. For example, Figure 4(a) shows the initial graph. In the limit, if the rates favor the adopt rules we obtain a graph as in Figure 4(b), while Figure 4(c) shows a possible result of dominant rewiring where the graph has split split into two components, one for each opinion.



(a) DTMC Version of the Voter Model

(b) CTMC Version of the Voter Model

Figure 5: Simulation results

To answer RQ1 we use the rules as presented in Section 2, but modify the simulation algorithm by the weight factors derived in Section 3.3. This produces a simulation matching that of a discrete time Markov chain. While fixing the model size, we perform a nested parameter sweep over α, the static probability of rewiring, and over the fraction u of agents voting 0. The results are shown in Figure 5(a). On the y-axis we plot the fraction of voters with the minority opinion in the final graph, while the x-axis shows the α value for each simulation. Furthermore, each set of symbols in the figure represents a different opinion ratio u in the initial graph. As we can see, independently of the u there is a clear point after which an increase in α leads to a separation of voters by opinion, which implies a segmentation of the graph. In turn, if α is low, the minority opinion disappears. Thus, Figure 5(a) reflects very closely the findings of [11], with only a minor deviation of the value of α separating the two outcomes.

For RQ2, we execute the same rules with our standard implementation simulation algorithm reflecting mass action CTMC semantics. While fixing the rate of the adopt rules as 1, we perform a nested parameter sweep over α, in this case denoting the rate of the rewiring rules, and the minority fraction u. The results are shown in Figure 5(b), again with the y-axis showing the fractions of voters in the minority in each final graph and the x-axis tracking the α values. As before, each set of symbols represents a different initial fraction u. In contrast to Figure 5(a), here the threshold of α at which the final graph becomes segmented is orders of magnitudes smaller. This is a result of the CTMC-based simulation algorithm, that obtains the rule application probability by multiplying a rule's match count with its static

rate. Intuitively, the set of matches for each of the adopt rules consists of all connected pairs of voters $v1, v2$ of different opinion. Instead, the set of matches of the rewire rules is made up by the Cartesian product of this first set with the set of voters sharing the opinion of $v1$. To find the same balancing point as in the probabilistic model, this "unfair advantage" of the rewire rules is compensated for by a very low rate.

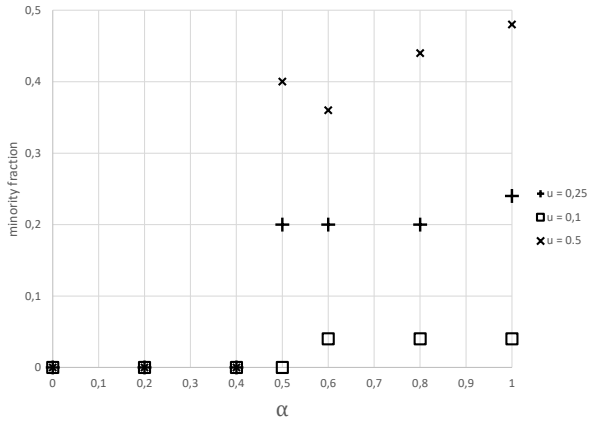


Figure 6: Simulation results of the alternative CTMC Version described by the rules in equation (21)

Finally, to address RQ3 we perform simulations using adopt and rewire rules extended to a common left-hand side consisting of two connected voters $v1, v2$ of conflicting opinions plus one extra voter each of opinion 0 and 1. By the same argument as above this leads to a large increase in the number of matches and hence poses scalability challenges to the simulation. To address this we used a smaller initial graph of 50 Voters and 200 groups and executed only 10 simulation runs per parameter configuration rather than 40 as in the other two models. (In each case, configurations range across 3 values for u and 7 values for α totaling 21 parameter configurations for each model.) The results are shown in Figure 6 using the same layout as before. It is interesting that, despite the high degree of activity in this model due to the very large match count, it displays almost the same behavior as the DTMC version (M1), even down to the α threshold.

The limited scalability of this approach is in part due to the global nature of the requirement that all rules share the same left-hand side. In the MDP-based approach of [19] this applies only to the subsets of rules defining the same action. One could then consider the four alternatives of rewire and adopt as cases of the same action of resolving a conflict edge, leading to a probabilistic graph transformation system with one rule. An analysis of the relation of mass-action CTMC with MDP-based semantics, however, is beyond the scope of this paper.

Social network analysis of the voter model and its variants is limited to a theoretical level, exploring mechanisms resulting in certain emergent phenomena that are observable in real-life networks, but are not quantified using real data. For a deeper quantitative analysis of phenomena such as the spread of opinions or the fragmentation of networks, the parameters of our rule-based models need to be matched to real social network data where such is available. This means, in particular, to determine the rates of the rules, e.g., from observations of pattern frequencies.

Such approaches are well established in chemical and biological modeling where statistical methods are used to derive rates of reaction rules from concentrations, i.e., relative pattern frequencies describing the ratios of the different types molecules [18, 13]. Instead, the derivation of an equational model from the operational rule-based description based on the rule algebra approach described in this paper could

allow for an analytical approach where rates are determined by solving a system of equations. The precise conditions under which this is possible are a subject of future research.

5 Conclusion

In this paper we analyzed the non-standard semantics behind the formulation of adaptive system models in the literature. For a simple but prototypical example, we focused specifically on the voter models of opinion formation in social networks. We analyzed the non-standard semantics behind these models and established that they can be seen as Discrete-Time Markov Chains (DTMCs). In order to start the study of such models using the concepts, theories and tools of stochastic graph transformations (SGTS), we formalized the semantic relation between the rule-based specification by SGTS of mass-action Continuous-Time Markov Chains (CTMCs) with the DTMC-based semantics, identifying two systematic ways by which an SGTS can be derived from a DTMC-based model while preserving the behavior in the limit.

In the first derivation, this leads to a generalized notion of SGTS with weight factors correcting for the mass-action component represented by the dependency of the jump rate on the number of matches for each rule. This new type of SGTS is supported by a generalized simulation algorithm supporting the analysis of DTMC-based probabilistic graph transformation systems which, to our knowledge, is original. The second derivation converts the rules of the system by extending them to the same left-hand sides, producing a model resembling Markov-Decision Process (MDP)-based probabilistic graph transformation rules.

Apart from the theoretical analysis, we validate both resulting systems through simulations, establishing that they reproduce the expected behavior of the model. We also create a direct model of the same system as an SGTS with standard mass-action CTMC semantics and succeed in determining its parameters experimentally to match the expected behavior. We conclude that we can use standard mass-action stochastic GTS to model the phenomena expressed by the voter models in the literature, providing a starting point for further more elaborate modeling and analyses.

In future work we want to study more complex adaptive networks, including variations on the voter model with groups of more than two members, opinion profiles for a set of topics instead of a single opinion per voter, and including concepts such as influencers (who actively try to link to and persuade others), or zealots (who do not change their opinions). We are also planning to study how to match models to social network data. On the foundational side, we are planning optimizations to the simulation algorithm and a study of the relation between mass-action CTMC and MDP semantics of probabilistic graph transformations. We can also derive and study differential equations (ODEs) from our SGTS using the rule-algebra formalism. This provides a more scalable solution to analyzing their behavior complementing the simulation-based approach.

References

[1] Fabian Baumann, Philipp Lorenz-Spreen, Igor M. Sokolov & Michele Starnini (2020): *Modeling Echo Chambers and Polarization Dynamics in Social Networks*. *Phys. Rev. Lett.* 124, p. 048301, doi:10.1103/PhysRevLett.124.048301.

[2] Nicolas Behr (2019): *Sesqui-Pushout Rewriting: Concurrency, Associativity and Rule Algebra Framework*. In Rachid Echahed & Detlef Plump, editors: *Proceedings of theTenth International Workshop on Graph*

Computation Models (GCM 2019) in Eindhoven, The Netherlands, *Electronic Proceedings in Theoretical Computer Science* 309, Open Publishing Association, pp. 23–52, doi:10.4204/eptcs.309.2.

[3] Nicolas Behr (2021): *On Stochastic Rewriting and Combinatorics via Rule-Algebraic Methods*. In Patrick Bahr, editor: *Proceedings 11th International Workshop on Computing with Terms and Graphs (TERM-GRAPH 2020)*, 334, Open Publishing Association, pp. 11–28, doi:10.4204/eptcs.334.2.

[4] Nicolas Behr, Vincent Danos & Ilias Garnier (2016): *Stochastic mechanics of graph rewriting*. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '16*, ACM Press, pp. 46–55, doi:10.1145/2933575.2934537.

[5] Nicolas Behr, Vincent Danos & Ilias Garnier (2020): *Combinatorial Conversion and Moment Bisimulation for Stochastic Rewriting Systems*. *Logical Methods in Computer Science* Volume 16, Issue 3. Available at `https://lmcs.episciences.org/6628`.

[6] Nicolas Behr & Jean Krivine (2020): *Rewriting theory for the life sciences: A unifying framework for CTMC semantics*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation, 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25–26, 2020, Proceedings, Theoretical Computer Science and General Issues* 12150, Springer International Publishing, pp. 185–202, doi:10.1007/978-3-030-51372-6.

[7] Nicolas Behr & Jean Krivine (2021): *Compositionality of Rewriting Rules with Conditions*. *Compositionality* 3, doi:10.32408/compositionality-3-2.

[8] Nicolas Behr & Pawel Sobocinski (2020): *Rule Algebras for Adhesive Categories (extended journal version)*. *Logical Methods in Computer Science* Volume 16, Issue 3. Available at `https://lmcs.episciences.org/6615`.

[9] Paolo Bolzern, Patrizio Colaneri & Giuseppe De Nicolao (2020): *Opinion Dynamics in Social Networks: The Effect of Centralized Interaction Tuning on Emerging Behaviors*. *IEEE Transactions on Computational Social Systems* 7(2), pp. 362–372, doi:10.1109/TCSS.2019.2962273.

[10] Andrea Corradini, Tobias Heindel, Frank Hermann & Barbara König (2006): *Sesqui-Pushout Rewriting*. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro & G. Rozenberg, editors: *Graph Transformations (ICGT 2006)*, *Lecture Notes in Computer Science* 4178, Springer Berlin Heidelberg, pp. 30–45, doi:10.1007/11841883_4.

[11] Richard Durrett, James P. Gleeson, Alun L. Lloyd, Peter J. Mucha, Feng Shi, David Sivakoff, Joshua E. S. Socolar & Chris Varghese (2012): *Graph fission in an evolving voter model*. *Proceedings of the National Academy of Sciences* 109(10), pp. 3682–3687, doi:10.1073/pnas.1200709109.

[12] Sebastian Ehmes, Lars Fritsche & Andy Schürr (2019): *SimSG: Rule-based Simulation using Stochastic Graph Transformation*. *J. Object Technol.* 18(3), pp. 1:1–17, doi:10.5381/jot.2019.18.3.a1.

[13] Hoda Eydgahi, William W Chen, Jeremy L Muhlich, Dennis Vitkup, John N Tsitsiklis & Peter K Sorger (2013): *Properties of cell death models calibrated and compared using Bayesian approaches*. *Molecular Systems Biology* 9(1), p. 644, doi:10.1038/msb.2012.69.

[14] Daniel T. Gillespie (1977): *Exact stochastic simulation of coupled chemical reactions*. *The Journal of Physical Chemistry* 81(25), pp. 2340–2361, doi:10.1021/j100540a008.

[15] Thilo Gross & Bernd Blasius (2008): *Adaptive coevolutionary networks: a review*. *J. R. Soc. Interface* 5(20), pp. 259–271, doi:10.1098/rsif.2007.1229.

[16] Reiko Heckel, Georgios Lajios & Sebastian Menge (2004): *Stochastic Graph Transformation Systems*. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce & Grzegorz Rozenberg, editors: *Graph Transformations*, Springer Berlin Heidelberg, pp. 210–225, doi:10.1007/978-3-540-30203-2_16.

[17] Pascal P. Klamser, Marc Wiedermann, Jonathan F. Donges & Reik V. Donner (2017): *Zealotry effects on opinion dynamics in the adaptive voter model*. *Phys. Rev. E* 96, p. 052315, doi:10.1103/PhysRevE.96.052315.

[18] David J. Klinke (2014): *In silico model-based inference: A contemporary approach for hypothesis testing in network biology*. *Biotechnology Progress* 30, doi:doi.org/10.1002/btpr.1982.

[19] Christian Krause & Holger Giese (2012): *Probabilistic Graph Transformation Systems*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Graph Transformations*, *Lecture Notes in Computer Science* 7562, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 311–325, doi:10.1007/978-3-642-33654-6_21.

[20] Erhan Leblebici, Anthony Anjorin & Andy Schürr (2014): *Developing eMoflon with eMoflon*. In Davide Di Ruscio & Dániel Varró, editors: *Theory and Practice of Model Transformations*, *Lecture Notes in Computer Science* 8568, Springer International Publishing, Cham, pp. 138–145, doi:10.1007/978-3-319-08789-4_10.

[21] Mark Newman (2008): *The physics of networks*. *Physics Today* 61(11), pp. 33–38, doi:10.1063/1.3027989.

[22] Jonathan M Read, Ken T.D Eames & W. John Edmunds (2008): *Dynamic social networks and the implications for the spread of infectious disease*. *Journal of The Royal Society Interface* 5(26), pp. 1001–1007, doi:10.1098/rsif.2008.0013.

[23] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth & Zoltán Ujhelyi (2016): *Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework*. *Software & Systems Modeling* 15(3), pp. 609–629, doi:10.1007/s10270-016-0530-4.

[24] Gergely Varró & Frederik Deckwerth (2013): *A Rete Network Construction Algorithm for Incremental Pattern Matching*. In Keith Duddy & Gerti Kappel, editors: *Theory and Practice of Model Transformations (ICMT 2013)*, *Lecture Notes in Computer Science* 7909, Springer Berlin Heidelberg, pp. 125–140, doi:10.1007/978-3-642-38883-5_13.

[25] Gerd Zschaler, Gesa A. Böhme, Michael Seißinger, Cristián Huepe & Thilo Gross (2012): *Early fragmentation in the adaptive voter model on directed networks*. *Phys. Rev. E* 85, p. 046107, doi:10.1103/PhysRevE.85.046107.

A Categorical Semantics for Hierarchical Petri Nets

Fabrizio Genovese
0000-0001-7792-1375
University of Pisa / Statebox
fabrizio.romano.genovese@gmail.com

Jelle Herold
0000-0002-1966-2536
Statebox
research@statebox.io

Fosco Loregian
0000-0003-3052-465X
Tallinn University of Technology
fosco.loregian@gmail.com

Daniele Palombi
0000-0002-8107-5439
Sapienza University of Rome
danielepalombi@protonmail.com

We show how a particular variety of hierarchical nets, where the firing of a transition in the parent net must correspond to an execution in some child net, can be modelled utilizing a functorial semantics from a free category – representing the parent net – to the category of sets and spans between them. This semantics can be internalized via Grothendieck construction, resulting in the category of executions of a Petri net representing the semantics of the overall hierarchical net. We conclude the paper by giving an engineering-oriented overview of how our model of hiearchic nets can be implemented in a transaction-based smart contract environment.

1 Introduction

This paper is the fourth instalment in a series of works [22, 21, 20] devoted to describing the semantics of extensions of Petri nets using categorical tools.

Category theory has been applied to Petri nets starting in the nineties [29]; see also [6, 11, 8, 7, 3, 10, 5, 9, 12, 4]. The main idea is that we can use different varieties of free monoidal categories to describe the executions (or runs) of a net [28, 18]. These works have been influential since they opened up an avenue of applying high-level methods to studying Petri nets and their properties. For instance, in [2] the categorical approach allowed to describe glueing of nets leveraging on colimits and double categories, while category-theory libraries such as [17] can be leveraged to implement nets in a formally verified way. These libraries implement category theory *directly*, so that one could translate the categorical definitions defining some model object directly and obtain an implementation.

In [22], we started another line of research, where we were able to define a categorical semantics for coloured nets employing monoidal functors. The Grothendieck construction was then used to internalize this semantics, obtaining the well-known result that coloured nets can be "compiled back" to Petri nets.

In [21, 20], we extended these ideas further, and we were able to characterize bounded nets and mana-nets – a new kind of nets useful to model chemical reactions – in terms of generalized functorial semantics.

This approach, based on the correspondence between slice categories and lax monoidal functors to the category of spans [31], has still a lot to give. In this paper, we show how it can be used to model hierarchical nets.

There are a lot of different ways to define hierarchical nets [24, 16, 30, 23, 14], which can be seen as a graph-based model. It means that we have one "parent" Petri net and a bunch of "child" nets. A transition

firing in the parent net corresponds to some sort of run happening in a corresponding child net. The main net serves to orchestrate and coordinate the executions of many child nets in the underlayer.

This paper will contain very little new mathematics. Instead, we will reinterpret results obtained in [22] to show how they can be used to model hierarchical nets, moreover, in a way that makes sense from an implementation perspective.

It is worth noting that category theory in this paper is used in a way that is slightly different than the usage in graph transformations research: We won't be using category theory to generalize definitions and proofs to different classes of graph(-related) objects. Instead, we will employ categorical concepts to actually build a semantics for hierarchical Petri nets.

2 Nets and their executions

Definition 1 (Monad, comonad)**.** *Let $\mathcal{C}$ be a category; a* monad *on $\mathcal{C}$ consists of an endofunctor $T : \mathcal{C} \to \mathcal{C}$ endowed with two natural transformations*

- $\mu : T \circ T \Rightarrow T$*, the* multiplication *of the monad, and*
- $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$*, the* unit *of the monad,*

such that the following axioms are satisfied:

- *the multiplication is associative, i.e. the diagram*

$$\begin{array}{ccc} T \circ T \circ T & \xrightarrow{T*\mu} & T \circ T \\ {\scriptstyle \mu * T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T \circ T & \xrightarrow[\mu]{} & T \end{array}$$

  is commutative, i.e. the equality of natural transformations $\mu \circ (\mu * T) = \mu \circ (T * \mu)$ holds;

- *the multiplication has the transformation η as unit, i.e. the diagram*

$$\begin{array}{ccccc} T & \xrightarrow{\eta * T} & T \circ T & \xleftarrow{T*\eta} & T \\ & \searrow & \downarrow{\scriptstyle \mu} & \swarrow & \\ & & T & & \end{array}$$

  is commutative, i.e. the equality of natural transformations $\mu \circ (\eta * T) = \mu \circ (T * \eta) = \mathrm{id}_T$ holds.

Dually, let $\mathcal{C}$ be a category; a comonad *on $\mathcal{C}$ consists of an endofunctor $T : \mathcal{C} \to \mathcal{C}$ endowed with two natural transformations*

- $\sigma : T \Rightarrow T \circ T$*, the* comultiplication *of the comonad, and*
- $\varepsilon : T \Rightarrow \mathrm{id}_{\mathcal{C}}$*, the* counit *of the comonad,*

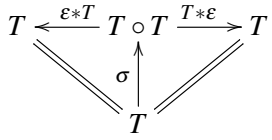
such that the following axioms are satisfied:

- *the comultiplication is coassociative, i.e. the diagram*

$$\begin{array}{ccc} T \circ T \circ T & \xleftarrow{T*\sigma} & T \circ T \\ {\scriptstyle \sigma * T}\uparrow & & \uparrow{\scriptstyle \sigma} \\ T \circ T & \xleftarrow[\sigma]{} & T \end{array}$$

  is commutative.

- *the comultiplication has the transformation ε as counit, i.e. the diagram*



is commutative.

Definition 2 (Bicategory)**.** *A* (locally small) bicategory $\mathcal{B}$ *consists of the following data.*

1. *A class $\mathcal{B}_o$ of* objects*, denoted with Latin letters like $A, B, \ldots$.*
2. *A collection of (small) categories $\mathcal{B}(A,B)$, one for each $A, B \in \mathcal{B}_o$, whose objects are called* 1-cells *or* arrows *with* domain *A and* codomain *B, and whose morphisms $\alpha : f \Rightarrow g$ are called* 2-cells *or* transformations *with domain f and codomain g; the composition law $\circ$ in $\mathcal{B}(A,B)$ is called* vertical composition *of 2-cells.*
3. *A* horizontal composition *of 2-cells*

$$\boxminus_{\mathcal{B},ABC} : \mathcal{B}(B,C) \times \mathcal{B}(A,B) \to \mathcal{B}(A,C) : (g,f) \mapsto g \boxminus f$$

defined for any triple of objects A, B, C. This is a family of functors between hom-categories.

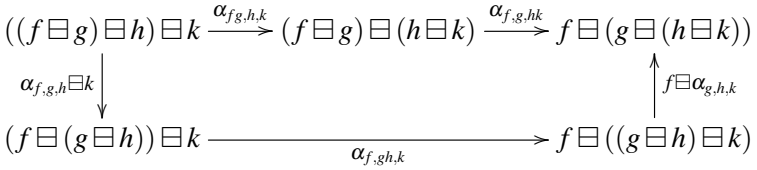
4. *For every object $A \in \mathcal{B}_o$ there is an arrow $\mathrm{id}_A \in \mathcal{B}(A,A)$.*

To this basic structure we add

1. *a family of invertible maps $\alpha_{fgh} : (f \boxminus g) \boxminus h \cong f \boxminus (g \boxminus h)$ natural in all its arguments f, g, h, which taken together form the* associator *isomorphisms;*
2. *a family of invertible maps $\lambda_f : \mathrm{id}_B \boxminus f \cong f$ and $\rho_f : f \boxminus \mathrm{id}_A \cong f$ natural in its component $f : A \to B$, which taken together form the* left unitor *and* right unitor *isomorphisms.*

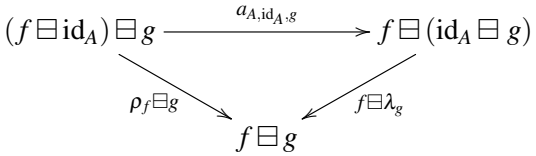
Finally, these data are subject to the following axioms.

1. *For every quadruple of 1-cells f, g, h, k we have that the diagram*



commutes.

2. *For every pair of composable 1-cells f, g,*



commutes.

Definition 3 (Bimodule)**.** *Given a bicategory $\mathcal{B}$ having finite colimits (in the 2-categorical sense of [25]), define the 2-category* **Mod**$(\mathcal{B})$ *of* bimodules *as in [36, p. 2.19]:*

- *0-cells are the monads in $\mathcal{B}$;*
- *1-cells $T \to S$ are* bimodules*, i.e. 1-cells $H : C \to D$ (assuming T is a monad on C, and S a monad on D) equipped with suitable action maps: $\rho : HT \to H$ and $\lambda : SH \to H$ satisfying suitable axioms expressing the fact that T acts on the right over H, via ρ (resp., S acts on the left on H, va λ);*
- *2-cells are natural transformations $\alpha : H \Rightarrow K : T \to S$ compatible with the action maps.*

Definition 4 (Pseudofunctor, (co)lax functor)**.** *Let $\mathcal{B}, \mathcal{C}$ be two bicategories; a* pseudofunctor *consists of*

1. *a function $F_o : \mathcal{B}_o \to \mathcal{C}_o$,*
2. *a family of functors $F_{AB} : \mathcal{B}(A,B) \to \mathcal{C}(FA, FB)$,*
3. *an invertible 2-cell $\mu_{fg} : Ff \circ Fg \Rightarrow F(fg)$ for each $A \xrightarrow{g} B \xrightarrow{f} C$, natural in f (with respect to vertical composition) and an invertible 2-cell $\eta : \eta_f : \mathrm{id}_{FA} \Rightarrow F(\mathrm{id}_A)$, also natural in f.*

These data are subject to the following commutativity conditions for every 1-cell $A \to B$:

$$\begin{array}{ccc} Ff \circ \mathrm{id}_A & \xrightarrow{\rho_{Ff}} & Ff \\ {\scriptstyle Ff*\eta}\downarrow & & \downarrow{\scriptstyle F(\rho_f)} \\ Ff \circ F(\mathrm{id}_A) & \xrightarrow[\mu_{f,\mathrm{id}_A}]{} & F(f \circ \mathrm{id}_A) \end{array} \qquad \begin{array}{ccc} \mathrm{id}_B \circ Ff & \xrightarrow{\lambda_{Ff}} & Ff \\ {\scriptstyle \eta*Ff}\downarrow & & \downarrow{\scriptstyle F(\lambda_f)} \\ F(\mathrm{id}_B) \circ Ff & \xrightarrow[\mu_{\mathrm{id}_B,f}]{} & F(\mathrm{id}_B \circ f) \end{array}$$

$$\begin{array}{ccc} (Ff \circ Fg) \circ Fh & \xrightarrow{\alpha_{Ff,Fg,Fh}} & Ff \circ (Fg \circ Fh) \\ {\scriptstyle \mu_{fg}*Fh}\downarrow & & \downarrow{\scriptstyle Ff*\mu_{gh}} \\ F(fg) \circ Fh & & Ff \circ F(gh) \\ {\scriptstyle \mu_{fg}*Fh}\downarrow & & \downarrow{\scriptstyle \mu_{f,gh}} \\ F((fg)h) & \xrightarrow[F\alpha_{fgh}]{} & F(f(gh)) \end{array}$$

(we denote invariably α, λ, ρ the associator and unitor of $\mathcal{B}, \mathcal{C}$).

A lax *functor is defined by the same data, but both the 2-cells $\mu : Ff \circ Fg \Rightarrow F(fg)$ and $\eta : \mathrm{id}_{FA} \Rightarrow F(\mathrm{id}_A)$ can be non-invertible; the same coherence diagrams in Definition 4 hold. A* colax *functor reverses the direction of the cells μ, η, and the commutativity of the diagrams in Definition 4 changes accordingly.*

2.1 Categorical Petri nets

Notation 1. Let S be a set; a multiset is a function $S \to \mathbb{N}$. Denote with $S^{\oplus}$ the set of multisets over S. Multiset sum and difference (only partially defined) are defined pointwise and will be denoted with $\oplus$ and $\ominus$, respectively. The set $S^{\oplus}$ together with $\oplus$ and the empty multiset is isomorphic to the free commutative monoid on S.

Definition 5 (Petri net)**.** *A* Petri net *is a pair of functions $T \xrightarrow{s,t} S^{\oplus}$ for some sets T and S, called the set of places and transitions of the net, respectively. s,t are called* input and output *functions, respectively, or equivalently* source and target.

A morphism of nets *is a pair of functions $f : T \to T'$ and $g : S \to S'$ such that the following square commutes, with $g^{\oplus} : S^{\oplus} \to S'^{\oplus}$ the obvious lifting of g to multisets:*

$$\begin{array}{ccccc} S^{\oplus} & \xleftarrow{s} & T & \xrightarrow{t} & S^{\oplus} \\ {\scriptstyle g^{\oplus}}\downarrow & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g^{\oplus}} \\ S'^{\oplus} & \xleftarrow[s']{} & T' & \xrightarrow[t']{} & S'^{\oplus} \end{array}$$

Petri nets and their morphisms form a category, denoted **Petri**. *Details can be found in [29].*

Definition 6 (Markings and firings)**.** *A* marking *for a net $T \xrightarrow{s,t} S^{\oplus}$ is an element of $S^{\oplus}$, representing a distribution of tokens in the net places. A transition u is* enabled *in a marking M if $M \ominus s(u)$ is defined. An enabled transition can* fire*, moving tokens in the net. Firing is considered an atomic event, and the marking resulting from firing u in M is $M \ominus s(u) \oplus t(u)$. Sequences of firings are called* executions*.*

The main insight of categorical semantics for Petri nets is that the information contained in a given net is enough to generate a free symmetric strict monoidal category representing all the possible ways to run the net. There are multiple ways to do this [32, 18, 19, 28, 1]. In this work, we embrace the *individual-token philosophy*, where tokens are considered distinct and distinguishable and thus require the category in Definition 7 to have non-trivial symmetries.

Definition 7 (Category of executions – individual-token philosophy)**.** *Let $N : T \xrightarrow{s,t} S^{\oplus}$ be a Petri net. We can generate a* free symmetric strict monoidal category (FSSMC)*, $\mathfrak{F}(N)$, as follows:*

- *The monoid of objects is the free monoid generated by S. Monoidal product of objects A, B is denoted with $A \otimes B$.*
- *Morphisms are generated by T: each $u \in T$ corresponds to a morphism generator (u, su, tu), pictorially represented as an arrow $su \xrightarrow{u} tu$; morphisms are obtained by considering all the formal (monoidal) compositions of generators and identities.*

A detailed description of this construction can be found in [28].

In this definition, objects represent markings of a net. For instance, the object $A \oplus A \oplus B$ means "two tokens in A and one token in B". Morphisms represent executions of a net, mapping markings to markings. A marking is reachable from another one if and only if there is a morphism between them. An example is provided in Fig. 1.

3 Hierarchical nets

Now we introduce the main object of study of the paper, *hierarchical nets*. As we pointed out in Section 1, there are many different ways to model hierarchy in Petri nets [24], often incompatible with each other. We approach the problem from a developer's perspective, wanting to model the idea that "firing a transition" amounts to call another process and waiting for it to finish. This is akin to calling subroutines in a piece of code. Moreover, we do not want to destroy the decidability of the reachability relation for our nets [15], as it happens for other hierarchical models such as the net-within-nets framework [26]. We consider this to be an essential requirement for practical reasons.

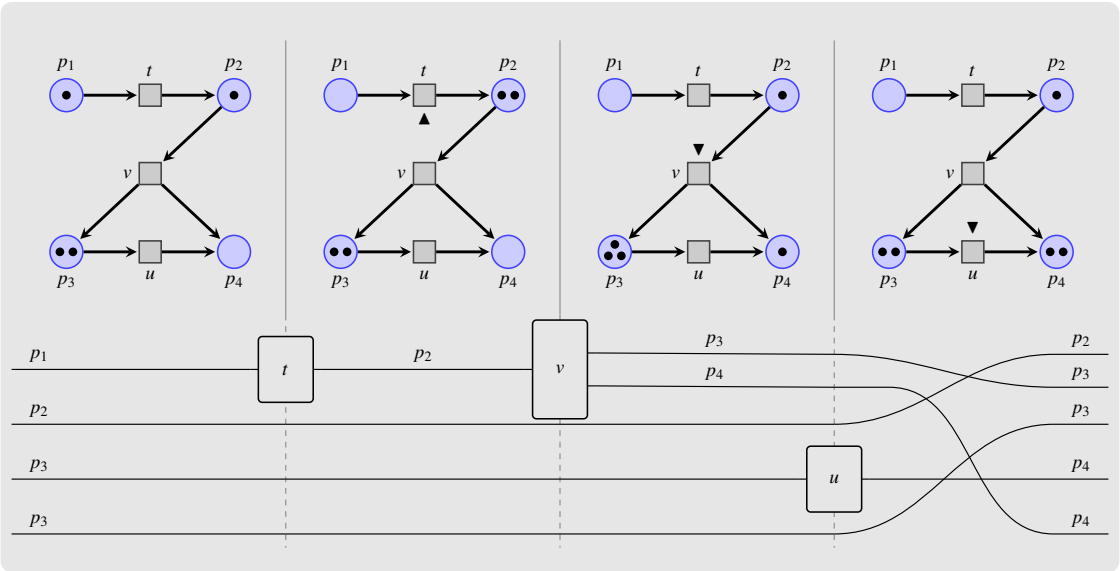


Figure 1: Graphical representation of a net's execution.

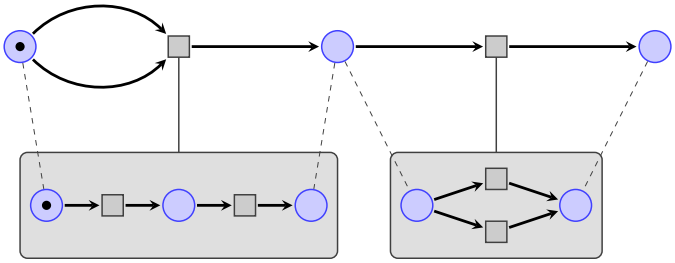


Figure 2: A hierarchical net.

We will postpone any formal definition to Section 5. In the present work, we focus on giving an intuitive explanation of what our requirements are.

Looking at the net in Fig. 2, we see a net on the top, which we call *parent*. To each transition of the parent net is attached another net, which we call *child*. Transitions can only have one child, but the parent net may have multiple transitions, and hence multiple children overall. Connecting input and output places of a transition in the parent net with certain places in the corresponding child, we can represent the orchestration by saying that each time a transition in the parent net fires, its input tokens are transferred to the corresponding child net, that takes them across until they reach a place connected with the output place in the parent net. This way, the atomic act of firing a transition in the parent net results in an execution of the corresponding child.

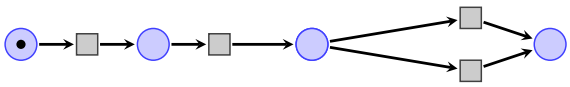


Figure 3: Replacing transitions in the parent net of Fig. 2 with its children.

Notice that we are not interested in considering the semantics of such hierarchical net to be akin to the one in Fig. 3, where we replaced transitions in the parent net with their corresponding children. There are two reasons for this: First, we want to consider transition firings in the parent net as atomic events, and replacing nets as above destroys this property. Secondly, such replacement is not so conceptually easy given that we do not impose any relationship between the parent net's topologies and its children. Indeed, the leftmost transition of the parent net in Fig. 2 consumes two inputs, while the corresponding leftmost transition in its child only takes one. How do we account for this in specifying rewriting-based semantics for hierarchical nets?

4 Local semantics for Petri nets

We concluded the last section pointing out reasons that make defining a semantics for hierarchical nets less intuitive than one would initially expect. Embracing an engineering perspective, we could get away with some ad-hoc solution to conciliate that parent and child net topologies are unrelated. One possible way, for instance, would be imposing constraints between the shapes of the parent net and its children. However, in defining things ad-hoc, the possibility for unforeseen corner cases and situations we do not know how to deal with becomes high. To avoid this, we embrace a categorical perspective and define things up to some degree of canonicity.

Making good use of the categorical work already carried out on Petri nets, our goal is to leverage it and get to a plausible definition of categorical semantics for hierarchical nets. Our strategy is to consider a hierarchical net as an extension of a Petri net: The parent net will be the Petri net we extend, whereas the children nets will be encoded in the extension.

This is precisely the main idea contained in [22], that is, the idea of describing net extensions with different varieties of monoidal functors. Indeed, we intend to show how the theory presented in [22], and initially serving a wholly different purpose, can be reworked to represent hierarchical nets with minimal effort.

As for semantics, we will use strict monoidal functors and name it *local* because the strict-monoidality requirement amounts to endow tokens with properties that cannot be shared with other tokens. To understand this choice of naming a little bit better, it may be worth comparing it with the notion of *non-local semantics*, defined in terms of lax-monoidal-lax functors, that we gave in [21].

Definition 8 (Local semantics for Petri nets)**.** *Given a strict monoidal category $\mathcal{S}$, a* Petri net with a local $\mathcal{S}$-semantics *is a pair $\left(N, N^\sharp\right)$, consisting of a Petri net N and a strict monoidal functor*

$$N^\sharp : \mathfrak{F}(N) \to \mathcal{S}.$$

A morphism $F : \left(M, M^\sharp\right) \to \left(N, N^\sharp\right)$ is just a strict monoidal functor $F : \mathfrak{F}(M) \to \mathfrak{F}(N)$ such that $M^\sharp = F \fatsemi N^\sharp$, where we denote composition in diagrammatic order; i.e. given $f : c \to d$ and $g : d \to e$, we denote their composite by $(f \fatsemi g) : c \to e$.

Nets equipped with $\mathcal{S}$-semantics and their morphisms form a monoidal category denoted $\mathbf{Petri}^{\mathcal{S}}$*, with the monoidal structure arising from the product in* **Cat**.

In [22], we used local semantics to describe guarded Petri nets, using **Span** as our category of choice. We briefly summarize this, as it will become useful later.

Definition 9 (The category **Span**)**.** *We denote by* **Span** *the 1-category of sets and spans, where isomorphic spans are identified. This category is symmetric monoidal. From now on, we will work with the* strictified *version of* **Span***, respectively.*

Notation 2. Recall that a morphism $A \to B$ in **Span** consists of a set S and a pair of functions $A \leftarrow S \to B$. When we need to extract this data from f, we write

$$A \xleftarrow{f_1} S_f \xrightarrow{f_2} B$$

We sometimes consider the span as a function $f\colon S_f \to A \times B$, thus we may write $f(s) = (a,b)$ for $s \in S_f$ with $f_1(s) = a$ and $f_2(s) = b$.

Definition 10 (Guarded nets with side effects)**.** *A* guarded net with side effects *is an object of* $\mathbf{Petri}^{\mathbf{Span}}$. *A morphism of guarded nets with side effects is a morphism in* $\mathbf{Petri}^{\mathbf{Span}}$.

Example 1. Let us provide some intuition behind the definition of $\mathbf{Petri}^{\mathbf{Span}}$.

Given a net N, its places (generating objects of $\mathfrak{F}(N)$) are sent to sets. Transitions (generating morphisms of $\mathfrak{F}(N)$) are mapped to spans. Spans can be understood as *relations with witnesses*, provided by elements in the apex of the span: Each path from the span domain to its codomain is indexed by some element of the span apex, as it is shown in Fig. 4. Witnesses allow considering different paths between the same elements. These paths represent the actions of processing the property a token is enowed with according to some side effect. Indeed, an element in the domain can be sent to different elements in the codomain via different paths. We interpret this as *non-determinism*: the firing of the transition is not only a matter of the tokens input and output; it also includes the chosen path, which we interpret as having side-effects interpreted outside of our model.

In Fig. 4 the composition of paths is the empty span: Seeing things from a reachability point of view, the

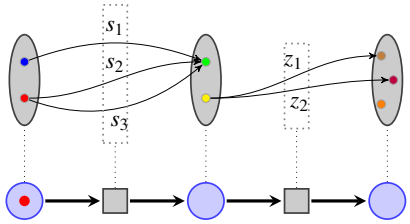


Figure 4: Semantics in **Span**

process given by firing the left transition and then the right will never occur. This is because the rightmost transition has a guard that only accepts yellow tokens, so that a green token can never be processed by it. This is witnessed by the fact that there is no path connecting the green dot with any dot on its right. The relation with reachability can be made precise by recasting Definition 7.

Definition 11 (Markings for guarded nets)**.** *Given a guarded Petri net with side effects* $\left(N, N^\sharp\right)$*, a* marking *for* $\left(N, N^\sharp\right)$ *is a pair* (X,x) *where* X *is an object of* $\mathfrak{F}(N)$ *and* $x \in N^\sharp X$*. We say that a marking* (Y,y) *is* reachable *from* (X,x) *if there is a morphism* $f : X \to Y$ *in* $\mathfrak{F}(N)$ *and an element* $s \in S_f$ *such that* $N^\sharp f(s) = (x,y)$.

5 Semantics for hierarchical nets

In the span semantics we can encode externalities in the tips of the spans to which we send transitions. That is, given a bunch of tokens endowed with some properties, to fire a transition, we need to provide a

witness that testifies how these properties have to be handled. The central intuition of this paper is that we can use side effects to encode the runs of some other net: To fire a transition in the parent net, we need to provide a *trace* of the corresponding child net. This can be made precise relying on the following high-level result:

Theorem 1 ([36, Section 2.4.3])**.** *Given a category A with finite limits, a* category internal in A *is a monad in* **Span**(A)*. Categories are monads in* **Span***, whereas strict monoidal categories are monads in* **Span**(**Mon**)*, with* **Mon** *being the category of monoids and monoid homomorphisms. A symmetric monoidal category is a* bimodule *in* **Span**(**Mon**)*, that is, a monad in* **Span**(**Mon**) *with extra structure.*

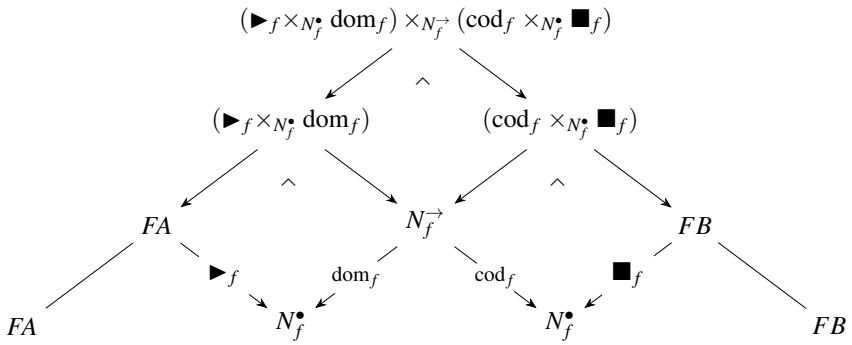
In particular, it follows that any free symmetric strict monoidal category can be represented as a span of monoids

$$N^\bullet \xleftarrow{\text{dom}} N^\to \xrightarrow{\text{cod}} N^\bullet$$

underlying a bimodule, with $N^\bullet$ and $N^\to$, representing the objects and arrows of the category, respectively, both free. Here, dom and cod map each morphism in $N^\to$ to its domain and codomain objects in $N^\bullet$. We will refer to such a span as *the* FSSMC N *(in* **Span**(**Mon**)*)*.

Definition 12 (Hierarchical nets – External definition)**.** *A* hierarchical net *is a functor* $\mathfrak{F}(N) \to$ **Span**(**Mon**) *defined as follows:*

- *Each generating object A of $\mathfrak{F}(N)$ is sent to a set FA, called* the set of accepting states for the place A.
- *Each generating morphism $A \xrightarrow{f} B$ is sent to a span with the following shape:*



  The FSSMC N_f *at the center of the span is called the* child net associated to f*; the morphisms $\blacktriangleright_f$ and $\blacksquare_f$ are called* play N_f *and* stop N_f*, respectively.*

Unrolling the definition, we are associating to each generating morphism of f of $\mathfrak{F}(N)$ – the parent net – a FSSMC N_f– the child net. As the feet of the spans corresponding to the child nets will, in general, be varying with the net themselves, we need to pre and post-compose them with other spans to ensure composability: $\blacktriangleright_f$ and $\blacksquare_f$ represent morphisms that select the *initial and accepting states* of N_f, that is, markings of N_f in which the computation starts, and markings of N_f in which the computation is considered as concluded. Notice how this also solves the problems highlighted in Section 3, as $\blacktriangleright_f$ and $\blacksquare_f$ mediate between the shape of inputs/outputs of the transition f and the shape of N_f itself.

Remark 1. Interpreting markings as in Definition 11, We see that to fire f in the parent net we need to provide a triple (a,x,b), where:

- a is an element of FA, witnessing that the tokens in the domain of f provide a valid initial state for N_f.

- x is an element of $N_f^{\rightarrow}$, that is, a morphism of N_f, and hence an execution of the child net.
- b is an element of FB, witnessing that the resulting state of the execution x is *accepting*, and can be lifted back to tokens in the codomain of f.

Definition 13 (Category of hierarchical Petri nets)**.** *Nets* $\left(N, N^\sharp\right)$ *in the category* $\mathbf{Petri}^{\mathbf{Span}}$ *with* $N^\sharp$ *having the shape of Definition 12 form a subcategory, denoted with* $\mathbf{Petri}^{\triangle}$*, and called* the category of hierarchical Petri nets.

Remark 2. Using the obvious forgetful functor $\mathbf{Mon} \to \mathbf{Set}$ we obtain a functor $\mathbf{Span}(\mathbf{Mon}) \to \mathbf{Span}$, which allows to recast our non-local semantics in a more liberal setting. In particular, we could send a transition to spans whose components are *subsets* of the monoids heretofore considered. We could select only a subset of the executions/states of the child net as valid witnesses to fire a transition in the parent.

Everything we do in this work will go through smoothly, but we consider this approach less elegant; thus, we will not mention it anymore.

6 Internalization

In Section 5 we defined hierarchical nets as nets endowed with a specific kind of functorial semantics to **Span**. As things stand now, Petri nets correspond to categories, while hierarchical nets correspond to functors. This difference makes it difficult to say what a Petri net with multiple levels of hierarchy is: intuitively, it is easy to imagine that the children of a parent net N can be themselves parents of other nets, which are thus "grandchildren" of N, and so on and so forth.

In realizing this, we are blocked by having to map N to hierarchical nets, which are functors and not categories. To make such an intuition viable, we need a way to *internalize* the semantics in Definition 12 to obtain a category representing the executions of the hierarchical net.

Luckily, there is a way to turn functors into categories, which relies on an equivalence between the slice 2-category over a given category $\mathcal{C}$, denoted $\mathbf{Cat}/\mathcal{C}$, and the 2-category of lax-functors $\mathcal{C} \to \mathbf{Span}$ [31]. This is itself the "1-truncated" version of a more general equivalence between the slice of **Cat** over $\mathcal{C}$, and the 2-category of lax *normal* functors to the bicategory **Prof** of profunctors (this has been discovered by Bénabou [13]; a fully worked out exposition, conducted in painstaking detail, is in [27]).

Here, we gloss over these abstract motivations and just give a very explicit definition of what this means, as what we need is just a particular case of the construction we worked out for guarded nets in [22].

Definition 14 (Internalization)**.** *Let* $\left(M, M^\sharp\right) \in \mathbf{Petri}^{\triangle}$ *be a hierarchical net. We define its* internalization*, denoted* $\int M^\sharp$*, as the following category:*

- *The objects of* $\int M^\sharp$ *are pairs* (X,x)*, where* X *is an object of* $\mathfrak{F}(M)$ *and* x *is an element of* $M^\sharp X$*. Concisely:*
$$\mathrm{Obj}\left(\int M^\sharp\right) := \left\{(X,x) \,\middle|\, (X \in \mathrm{Obj}\,\mathfrak{F}(M)) \wedge (x \in M^\sharp X)\right\}.$$
- *A morphism from* (X,x) *to* (Y,y) *in* $\int M^\sharp$ *is a pair* (f,s) *where* $f\colon X \to Y$ *in* $\mathfrak{F}(M)$ *and* $s \in S_{M^\sharp f}$ *in the apex of the corresponding span that connects* x *to* y*. Concisely:*
$$\begin{aligned}&\mathrm{Hom}_{\int M^\sharp}[(X,x),(Y,y)] := \\ &:= \left\{(f,s) \,\middle|\, (f \in \mathrm{Hom}_{\mathfrak{F}(M)}[X,Y]) \wedge (s \in S_{M^\sharp f}) \wedge (M^\sharp f(s) = (x,y))\right\}.\end{aligned}$$

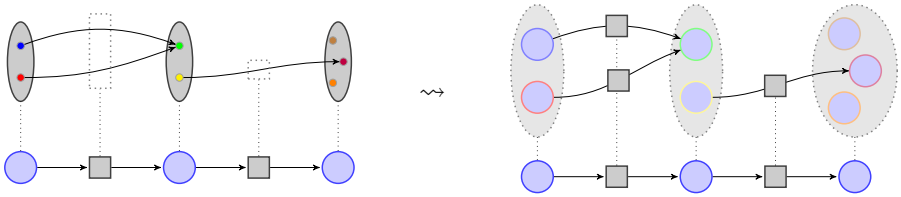


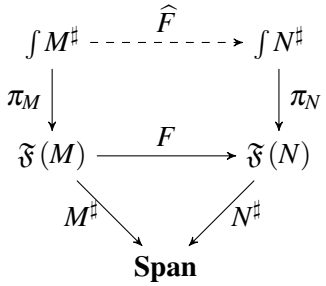
Figure 5: The Grothendieck construction applied to $N^\sharp$.

The category $\int N^\sharp$, called *the Grothendieck construction applied to* $N^\sharp$, produces a place for each element of the set we send a place to, and makes a transition for each path between these elements, as shown in Figure 5.

Notice that in Fig. 5, on the left, each path between coloured dots is a triple (a,x,b) as in Remark 1. This amounts to promote every possible trace of the child net – together with a selection of initial and accepting states – to a transition in the parent net. This interpretation is justified by the following theorem, which we again proved in [22]:

Theorem 2. *Given any strict monoidal functor* $\mathfrak{F}(N) \xrightarrow{N^\sharp}$ **Span**, *the category* $\int N^\sharp$ *is symmetric strict monoidal, and free. Thus* $\int N^\sharp$ *can be written as* $\mathfrak{F}(M)$ *for some net* M.

Moreover, we obtain a projection functor $\int N^\sharp \to \mathfrak{F}(N)$ *which turns* $\int$ *into a functor, in that for each functor* $F : (M, M^\sharp) \to (N, N^\sharp)$ *there exists a functor* $\widehat{F}$ *making the following diagram commute:*



Theorem 2 defines a functor $\mathbf{Petri}^{\mathbf{Span}} \to \mathbf{FSSMC}$, the category of FSSMCs and strict monoidal functors between them. As $\mathbf{Petri}^{\triangle}$ is a subcategory of $\mathbf{Petri}^{\mathbf{Span}}$, we can immediately restrict Theorem 2 to hierarchical nets. A net in the form $\int N^\sharp$ for some hiearchical net $(N, N^\sharp)$ is called the *internal categorical semantics for* N (compare this with Definition 12, which we called *external*).

Remark 3. Notice how internalization is *very* different from just copy-pasting a child net in place of a transition in the parent net as we discussed in Section 3. Here, each *execution* of the child net is promoted to a transition, preserving the atomicity requirement of transitions in the parent net.

Clearly, now we can define hierarchical nets with a level of hierarchy higher than two by just mapping a generator f of the parent net to a span where N_f is in the form $\int N^\sharp$ for some other hierarchical nets N, and the process can be recursively applied any finite number of times for each transition.

7 Engineering perspective

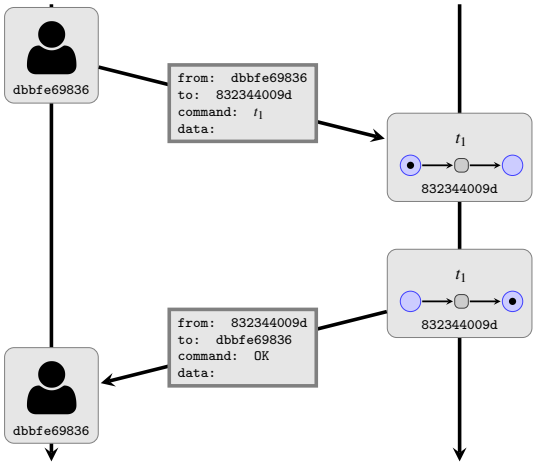
We deem it wise to spend a few words on why we consider this way of doing things advantageous from an applicative perspective. Petri nets have been considered as a possible way of producing software for a long

time, with some startups even using them as a central tool in their product offer [34]. Providing some form of hierarchical calling is needed to make the idea of "Petri nets as a programming language/general-purpose design tool" practical.

Our definition of hierarchy has the great advantage of not making hierarchical nets more expressive than Petri nets. If this seems like a downside, notice that a consequence of this is that decidability of any reachability-related question is exactly as for Petri nets, which is a great advantage from the point of view of model checking. Internalization, moreover, allows us to compile hierarchical nets back to Petri nets so that we can use already widespread tools for reachability checking [35] without having necessarily to focus on producing new ones.

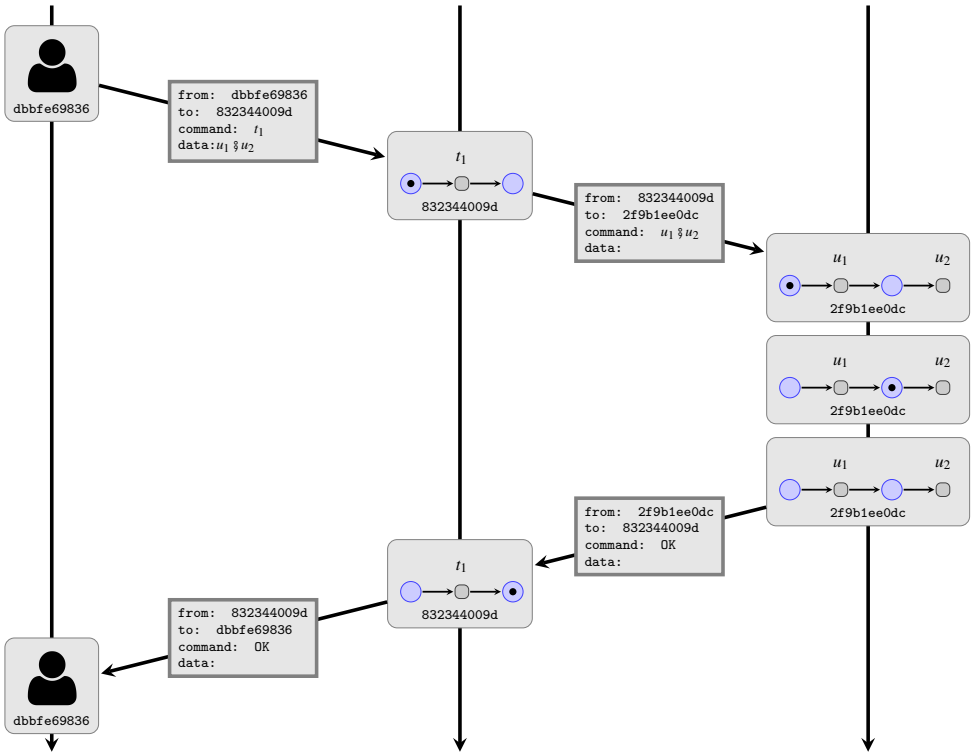
Moreover, and more importantly, our span formalism works really well in modelling net behaviour in a distributed setting. The parent and children nets may exist on different machines and are not required to be engineered in a monolithic fashion.

To better understand this, imagine an infrastructure where each Petri net is considered as a piece of data with its address (as it would be, for instance, if we were to implement nets as smart contracts on a blockchain). The way of operating Petri nets in this formalism is *transactional*: A user sends a message consisting of a net address, the transaction the user intends to fire, and some transaction data. The infrastructure replies affirmatively or negatively if the transaction can be fired, which amounts to accept or reject the transaction. Clearly, this is particularly suitable for blockchain-related contexts and is how applications such as [33] implement Petri nets in their services.



From this point of view, a hierarchical net would work exactly as a standard Petri net, with the exception that in sending a transaction to the parent net, the user also has to specify, in the transaction data, a proper

execution of the child net corresponding to the firing transition.



Again, from a smart contract standpoint, this means that the smart contract corresponding to the parent net will call the contract corresponding to the child net with some execution data, and will respond affirmatively to the user only if the generated call resolves positively.

All the possible ways of executing the contracts above form a category, which is obtaining by internalizing the hierarchical net corresponding to them via Theorem 2.

Internalized categories being free, they are presented by Petri nets, which we can feed to any mainstream model checker. Now, all sorts of questions about liveness and interaction of the contracts above can be analyzed by model-checking the corresponding internalized net.

This provides an easy way to analyze complex contract interaction, relying on tools that have been debugged and computationally optimized for decades.

8 Conclusion and future work

In this work, we showed how a formalism for guarded nets already worked out in [22] can be used to define the categorical semantics of some particular variety of hierarchical nets, which works particularly well from a model-checking and distributed-implementation point of view. Our effort is again part of a more ample project focusing on characterizing the categorical semantics of extensions of Petri nets by studying functors from FSSMCs to spans [21, 20].

As a direction of future work, we would like to obtain a cleaner way of describing recursively hierarchical nets. In this work, we relied on the Grothendieck construction to internalize a hierarchical net, so that we could use hierarchical nets as children of some other another parent net, recursively. This feels a bit like throwing all the carefully-typed information that the external semantics gives into the same

bucket, and as such it is a bit unsatisfactory. Ideally, we would like to get a fully external semantics for recursively hierarchical nets, and generalize the internalization result to this case.

Another obvious direction of future work is implementing the findings hereby presented, maybe relying on some formally verified implementation of category theory such as [17].

Acknowledgements Being non-native speakers, we want to thank John Baez for pointing out that the correct spelling in English was "Hierarchical Petri nets" and not "Hierarchic Petri nets" as we thought. We also want to thank G. Pasquini and M. De Jorio for having inspired this work.

The first author was supported by the project MIUR PRIN 2017FTXR7S "IT-MaTTerS" and by the Independent Ethvestigator Program.

The third author was supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

A video presentation of this paper can be found on Youtube at 4v5v8tgmiUM.

References

[1] J.C. Baez, F. Genovese, J. Master, and M. Shulman. *Categories of Nets*. Jan. 11, 2021. arXiv: 2101.04238 [cs, math]. URL: http://arxiv.org/abs/2101.04238 (cit. on p. 5).

[2] J.C. Baez and J. Master. "Open Petri Nets". In: *Mathematical Structures in Computer Science* 30.3 (Mar. 2020), pp. 314–341. arXiv: 1808.05415 (cit. on p. 1).

[3] P. Baldan, M. Bocci, D. Brigolin, N. Cocco, M. Heiner, and M. Simeoni. "Petri nets for modelling and analysing trophic networks". In: *Fundamenta Informaticae* 160.1-2 (2018), pp. 27–52 (cit. on p. 1).

[4] P. Baldan, F. Bonchi, F. Gadducci, and G.V. Monreale. "Asynchronous Traces and Open Petri Nets". In: *Programming Languages with Applications to Biology and Security*. Springer, 2015, pp. 86–102 (cit. on p. 1).

[5] P. Baldan, F. Bonchi, F. Gadducci, and G.V. Monreale. "Encoding synchronous interactions using labelled Petri nets". In: *International Conference on Coordination Languages and Models*. Springer. 2014, pp. 1–16 (cit. on p. 1).

[6] P. Baldan, F. Bonchi, F. Gadducci, and G.V. Monreale. "Modular encoding of synchronous and asynchronous interactions using open Petri nets". In: *Science of Computer Programming* 109 (2015), pp. 96–124 (cit. on p. 1).

[7] P. Baldan, N. Cocco, F. De Nes, M.L. Segura, and M. Simeoni. "MPath2PN-Translating metabolic pathways into Petri nets". In: *BioPPN2011 Int. Workshop on Biological Processes and Petri Nets, CEUR Workshop Proceedings*. Vol. 724. 2011, pp. 102–116 (cit. on p. 1).

[8] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. "Compositional modeling of reactive systems using open nets". In: *International Conference on Concurrency Theory*. Springer. 2001, pp. 502–518 (cit. on p. 1).

[9] P. Baldan, A. Corradini, H. Ehrig, and B. König. "Open Petri nets: Non-deterministic processes and compositionality". In: *International Conference on Graph Transformation*. Springer. 2008, pp. 257–273 (cit. on p. 1).

[10] P. Baldan, A. Corradini, F. Gadducci, and U. Montanari. "From Petri nets to graph transformation systems". In: *Electronic Communications of the EASST* 26 (2010) (cit. on p. 1).

[11] P. Baldan, A. Corradini, and U. Montanari. "Relating SPO and DPO graph rewriting with Petri nets having read, inhibitor and reset arcs". In: *Electronic Notes in Theoretical Computer Science* 127.2 (2005), pp. 5–28 (cit. on p. 1).

[12] P. Baldan and F. Gadducci. "Petri nets are dioids: a new algebraic foundation for non-deterministic net theory". In: *Acta Informatica* 56.1 (2019), pp. 61–92 (cit. on p. 1).

[13] J. Bénabou. "Introduction to Bicategories". In: J. Bénabou, R. Davis, A. Dold, J. Isbell, Saunders MacLane, U. Oberst, and J. -E. Roos. *Reports of the Midwest Category Seminar*. Vol. 47. Berlin, Heidelberg: Springer Berlin Heidelberg, 1967, pp. 1–77 (cit. on p. 10).

[14] P. Buchholz. "Hierarchical High Level Petri Nets for Complex System Analysis". In: *Application and Theory of Petri Nets*. 1994 (cit. on p. 1).

[15] J. Esparza and N. Mogens. "Decidability Issues for Petri Nets - a survey". In: *J. Inf. Process. Cybern.* 30.3 (1994), pp. 143–160 (cit. on p. 5).

[16] R. Fehling. "A concept of hierarchical Petri nets with building blocks". In: *International Conference on Application and Theory of Petri Nets*. Springer. 1991, pp. 148–168 (cit. on p. 1).

[17] F. Genovese, A. Gryzlov, J. Herold, A. Knispel, M. Perone, E. Post, and A. Videla. *Idris-Ct: A Library to Do Category Theory in Idris*. Nov. 25, 2019. arXiv: 1912.06191 [cs, math]. URL: http://arxiv.org/abs/1912.06191 (cit. on pp. 1, 14).

[18] F. Genovese, A. Gryzlov, J. Herold, M. Perone, E. Post, and A. Videla. *Computational Petri Nets: Adjunctions Considered Harmful*. Apr. 29, 2019. arXiv: 1904.12974 [cs, math]. URL: http://arxiv.org/abs/1904.12974 (cit. on pp. 1, 5).

[19] F. Genovese and J. Herold. "Executions in (Semi-)Integer Petri Nets Are Compact Closed Categories". In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 31, 2019), pp. 127–144 (cit. on p. 5).

[20] F. Genovese, F. Loregian, and D. Palombi. *A Categorical Semantics for Bounded Petri Nets*. Jan. 22, 2021. arXiv: 2101.09100 [cs, math]. URL: http://arxiv.org/abs/2101.09100 (cit. on pp. 1, 13).

[21] F. Genovese, F. Loregian, and D. Palombi. *Nets with Mana: A Framework for Chemical Reaction Modelling*. Jan. 15, 2021. arXiv: 2101.06234 [cs, math, q-bio]. URL: http://arxiv.org/abs/2101.06234 (cit. on pp. 1, 7, 13).

[22] F. Genovese and D.I. Spivak. "A Categorical Semantics for Guarded Petri Nets". In: *Graph Transformation*. Ed. by F. Gadducci and Timo Kehrer. Vol. 12150. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 57–74 (cit. on pp. 1, 2, 7, 10, 11, 13).

[23] P. Huber, K. Jensen, and R.M. Shapiro. "Hierarchies in coloured Petri nets". In: *International Conference on Application and Theory of Petri Nets*. Springer. 1989, pp. 313–341 (cit. on p. 1).

[24] K. Jensen and L.M. Kristensen. *Coloured Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009 (cit. on pp. 1, 5).

[25] G.M. Kelly. "Elementary observations on 2-categorical limits". In: *Bulletin of the Australian Mathematical Society* 39 (1989), pp. 301–317 (cit. on p. 4).

[26] M. Köhler-Bußmeier. "A Survey of Decidability Results for Elementary Object Systems". In: *Fundamenta Informaticae* 1 (2014), pp. 99–123 (cit. on p. 5).

[27] F. Loregian. *Coend Calculus*. first. Vol. 468. London Mathematical Society Lecture Note Series. ISBN 9781108746120. Cambridge University Press, July 2021 (cit. on p. 10).

[28] J. Master. "Petri Nets Based on Lawvere Theories". In: *Mathematical Structures in Computer Science* 30.7 (Aug. 2020), pp. 833–864. arXiv: 1904.09091 (cit. on pp. 1, 5).

[29] J. Meseguer and U. Montanari. "Petri Nets Are Monoids". In: *Information and Computation* 88.2 (Oct. 1990), pp. 105–155 (cit. on pp. 1, 5).

[30] H. Oswald, R. Esser, and R. Mattmann. "An environment for specifying and executing hierarchical Petri nets". In: *[1990] Proceedings. 12th International Conference on Software Engineering*. IEEE. 1990, pp. 164–172 (cit. on p. 1).

[31] D. Pavlović and S. Abramsky. "Specifying Interaction Categories". In: *Category Theory and Computer Science*. Ed. by Eugenio Moggi and Giuseppe Rosolini. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1290. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 147–158 (cit. on pp. 1, 10).

[32] V. Sassone. "On the Category of Petri Net Computations". In: *TAPSOFT '95: Theory and Practice of Software Development*. Ed. by Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach. Red. by Gerhard Goos, Juris Hartmanis, and Jan Leeuwen. Vol. 915. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 334–348 (cit. on p. 5).

[33] Statebox Team. *Statebox, Compositional Diagrammatic Programming Language*. 2017. URL: https://statebox.org (cit. on p. 12).

[34] Statebox Team. *The Mathematical Specification of the Statebox Language*. June 18, 2019. arXiv: 1906.07629 [cs, math]. URL: http://arxiv.org/abs/1906.07629 (cit. on p. 12).

[35] University of Torino. *GreatSPN Github Page*. 2018. URL: https://github.com/greatspn/SOURCES (cit. on p. 12).

[36] F. Zanasi. *Interacting Hopf Algebras: The Theory of Linear Systems*. May 4, 2018. arXiv: 1805.03032 [cs, math]. URL: http://arxiv.org/abs/1805.03032 (cit. on pp. 4, 9).

Resilience of Well-structured Graph Transformation Systems*

Okan Özkan Nick Würdemann
Department of Computing Science
University of Oldenburg
Oldenburg, Germany
{o.oezkan,wuerdemann}@informatik.uni-oldenburg.de

Resilience is a concept of rising interest in computer science and software engineering. For systems in which correctness w.r.t. a safety condition is unachievable, fast recovery is demanded. We investigate resilience problems of graph transformation systems. Our main contribution is the decidability of two resilience problems for well-structured graph transformation systems (with strong compatibility). We prove our results in the abstract framework of well-structured transition systems and apply them to graph transformation systems incorporating also the concept of adverse conditions.

1 Introduction

Resilience is a broadly used concept in computer science and software engineering (e.g., [22]), and a basic concept for, e.g., industrial control systems [18] and mobile cyber-physical systems [15]. For systems in which *correctness* w.r.t. a safety condition SAFE is unachievable, *fast recovery* is demanded. We interprete fast recovery as reachability of the safety condition in a bounded amount of time steps. The intuitive approach is to start from any *error state*, i.e., a state in which $\neg$SAFE($\equiv$ ERROR) holds, and try to reach a state in which SAFE holds again as fast as possible.

Another approach to formalizing resilience is to ask whether the system can *withstand an adverse effect* rather than to ask whether fast recovery is possible from any error state. To formally capture adverse effects we consider an environment which interacts with the system. In this setting, we investigate on the question whether a state satisfying SAFE can be reached in bounded time, starting from any state satisfying ENV, i.e., any state directly resulting from an environment interference.

For modeling systems we use *graph transformation systems (GTS)*, as considered, e.g., in [6], which are a visual but yet precise formalism. In this perception, system states are captured by graphs and state changes by graph transformations. Usually, the state set (the set of graphs) is infinite. To handle infinite state sets, we incorporate the concept of well-structuredness [1, 9, 12]. A *well-structured transition system (WSTS)* is informally a transition system equipped with a well-quasi order (wqo) satisfying that larger states simulate smaller states. They allow us to abstract from both of the approaches towards resilience described above. In the setting of WSTSs, we define resilience problems for a given downward-closed set J (a BAD condition, e.g., ERROR or ENV) and an upward-closed set I (e.g., a safety property SAFE). Given an initial state s and a natural number k, the *explicit resilience problem* asks whether we can, starting from s, reach I in at most k steps whenever we reach J. The *bounded resilience problem* asks whether there exists a k such that *k-step resilience* is satisfied.

We show that both resilience problems are decidable for *strongly* well-structured transition systems (SWSTS). We propose an algorithm which computes the minimal k s.t. we can recover from any BAD state in at most k steps, or returns `false` if there exists no such k. It is based on the ideal reachability algorithm proposed by Abdulla et al. [1], and solves both resilience problems at the same time.

*Supported by the German Research Foundation (DFG) through the Research Training Group (DFG GRK 1765) SCARE

When applying these results to GTSs, we assume that the corresponding graph class is of bounded path length in order to obtain a SWSTS. This sufficient condition for a GTS to be strongly well-structured is shown by König & Stückrath in [12]. The wqo on graphs used in this case is the subgraph order, so $I = I_{\text{SAFE}}$ corresponds to a constraint stating existence of subgraphs. We incorporate adverse conditions by distinguishing system and environment rules, and considering $J = J_{\text{ENV}}$, the set of graphs directly resulting from the application of an environment rule.

The rest of this paper is organized as follows: We recall preliminary concepts in Sec. 2. In Sec. 3, we present the concept of resilience in the context of adverse conditions and identify abstract resilience problems. In Sec. 4, we prove decidability of resilience for strongly well-structured transition systems. We apply these results to graph transformation systems incorporating adverse conditions in Sec. 5. In Sec. 6, we present related work. We close with a conclusion and an outlook in Sec. 7.

2 Preliminaries

We recall the concepts used in this paper, namely *graph transformation systems* [6, 5] and (in particular *well-structured*) *transition systems* [9].

2.1 Graph Transformation Systems

In the following, we recall the definitions of graphs, graph conditions, rules, and graph transformation systems [6, 5]. A directed, labeled graph consists of a set of nodes and a set of edges where each edge is equipped with a source and a target node and where each node and edge is equipped with a label. Note that this kind of graphs are a special case of the hypergraphs considered in [12].

Definition 1 (graphs & graph morphisms)**.** A *(directed, labeled) graph* (over a finite label alphabet Λ) is a tuple $G = \langle V_G, E_G, src_G, tgt_G, lab^V_G, lab^E_G \rangle$, with finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, functions $src_G, tgt_G : E_G \to V_G$ assigning *source* and *target* to each edge, and *labeling functions* $lab^V_G : V_G \to \Lambda$, $lab^E_G : E_G \to \Lambda$. A *(simple, undirected) path* p in G of length ℓ is a sequence $\langle v_1, e_1, v_2 \ldots, v_\ell, e_\ell v_{\ell+1} \rangle$ of nodes and edges s.t. $src_G(e_i) = v_i$ and $tgt_G(e_i) = v_{i+1}$, or $tgt_G(e_i) = v_i$ and $src_G(e_i) = v_{i+1}$ for every $1 \leq i \leq \ell$, and all contained nodes and edges occur at most once. Let $\ell(G)$ denote the length of a longest path in G. Given graphs G and H, a *(partial graph) morphism* $g : G \rightharpoonup H$ consists of partial functions $g_V : V_G \rightharpoonup V_H$ and $g_E : E_G \rightharpoonup E_H$ which preserve sources, targets, and labels, i.e., $g_V \circ src_G(e) = src_H \circ g_E(e)$, $g_V \circ tgt_G(e) = tgt_H \circ g_E(e)$, $lab^V_G(v) = lab^V_H \circ g_V(v)$, and $lab^E_G(e) = lab^E_H \circ g_E(e)$ on all egdes e and nodes v, for which $g_E(e), lab^E(e), lab^V(v)$ is defined. Furthermore, if a morphism is defined on an edge, it must be defined on all incident nodes. The morphism g is *total* (*injective*) if both g_V and g_E are total (injective). If g is total and injective, we also write $g : G \hookrightarrow H$. The composition of morphisms is defined componentwise.

We consider graph constraints [17, 10] whose validities are inherited to bigger/smaller graphs.

Definition 2 (positive & negative basic graph constraints)**.** The class of *positive (basic graph) constraints* is defined inductively: (i) $\exists G$ is a positive constraint where G is a graph, (ii) for positive constraints c, c', also $c \vee c'$, $c \wedge c'$ are positive constraints. Analogously, the *negative (basic graph) constraints* are defined by: (i) $\neg\exists G$ is a negative constraint for any graph G, (ii) for negative constraints c, c', also $c \vee c'$, $c \wedge c'$ are negative constraints.
A graph G *satisfies* $\exists G'$ if there exists an total injective morphism $G' \hookrightarrow G$. The semantics of the logical operators are as usual. We write $G \models c$ if G satisfies the positive/negative constraint c.

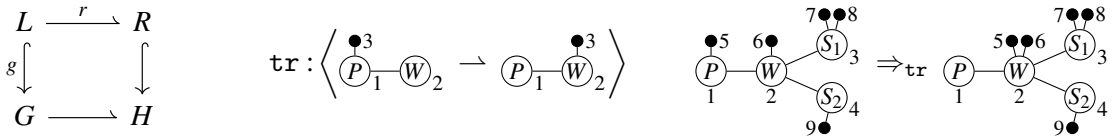
Remark. If c is a positive constraint, $\neg c$ is equivalent to a negative constraint, and vice versa.

Fact 1 (upward & downward inheritance)**.** *Let $G \hookrightarrow H$ be a total injective morphism, c be a positive constraint, and c' a negative constraint. If $G \models c$, then also $H \models c$. If $H \models c'$, then also $G \models c'$.*

We use the *single pushout (SPO)* approach [6, 12] with injective matches for modeling graph transformations. The reason for choosing SPO and not, e.g., the *double pushout approach (DPO)* [5] is that the dangling condition disturbs the compatibility condition in Def. 10.

Definition 3 (rules & transformations)**.** A *(graph transformation) rule* $r = \langle L \rightharpoonup R \rangle$ (over a finite label alphabet Λ) is a partial morphism from L to R (both graphs over Λ). A *(direct) transformation* $G \Rightarrow H$ from a graph G to a graph H applying rule r at a total injective *match morphism* $g : L \hookrightarrow G$ is given by a *pushout* as shown in Fig. 1a (for existence and construction of pushouts, see, e.g., [6]). We write $G \Rightarrow_r H$ to indicate the applied rule, and $G \Rightarrow_{\mathscr{R}} H$ if $G \Rightarrow_r$ for a rule r contained in the rule set $\mathscr{R}$.

Note that we do not have any application conditions. The pushout of a rule application is visualized in Fig. 1a. An example for a rule is presented in Fig. 1b, and an application of that rule in Fig. 1c.



(a) Pushout for the transformation $G \Rightarrow_r H$

(b) A graph transformation rule `tr`

(c) An application of the rule `tr`

Figure 1: Pushout and example of a direct graph transformation.

GTSs are simply finite sets of rules. We will specify the state set later.

Definition 4 (graph transformation system)**.** A *graph transformation system (GTS)* is a finite set of graph transformation rules.

2.2 Transition Systems

We recall the notion of transition systems. In Sec. 4, we prove our results on the level of transition systems and explicate the concept for graph transformation systems in Sec. 5.

Definition 5 (transition system)**.** A *transition system (TS)* $\langle S, \to \rangle$ consists of a (possibly infinite) set S of *states* and a *transition relation* $\to \subseteq S \times S$. Let $\to^0 = \mathrm{Id}_S$ (identitiy on S), $\to^1 = \to$, and $\to^k = \to^{k-1} \circ \to$ for every $k \geq 2$. Let $\to^{\leq k} = \bigcup_{0 \leq j \leq k} \to^j$ for every $k \geq 0$. The *transitive closure* is given by $\to^* = \bigcup_{k \geq 0} \to^k$.

A GTS is simply a set of rules. The following definition shows how it can be interpreted as a TS.

Definition 6 (graph transition system)**.** Let $\mathscr{R}$ be a GTS and $\mathscr{G}$ a set of graphs which is closed under rule application of $\mathscr{R}$. The *graph transition system* w.r.t. $\mathscr{R}$ and $\mathscr{G}$ is the transition system $\langle \mathscr{G}, \Rightarrow_{\mathscr{R}} \rangle$. A graph transition system $\langle \mathscr{G}, \Rightarrow_{\mathscr{R}} \rangle$ *is of bounded path length* if $\mathscr{G}$ is of bounded length, i.e., $\sup_{G \in \mathscr{G}} \ell(G) < \infty$.

Often we are interested in the predecessors or successors of a given set of states in a transition system.

Definition 7 (pre- & postsets)**.** Let $\langle S, \to \rangle$ be a transition system. For $S' \subseteq S$ and $k \geq 0$, we define $\mathrm{pre}^k(S') = \{s \in S \mid \exists s' \in S' : s \to^k s'\}$ and $\mathrm{post}^k(S') = \{s \in S \mid \exists s' \in S' : s' \to^k s\}$. Let $\mathrm{pre}^*(S') = \bigcup_{k \geq 0} \mathrm{pre}^k(S')$ and $\mathrm{post}^*(S') = \bigcup_{k \geq 0} \mathrm{post}^k(S')$. We abbreviate $\mathrm{post}^1(S')$ by $\mathrm{post}(S')$ and $\mathrm{pre}^1(S')$ by $\mathrm{pre}(S)$.

GTSs, when interpreted as TSs, have in general an infinite state space.

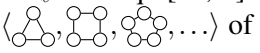
2.3 Well-structuredness

While several problems are undecidable for transition systems in general due to their infinite state space, many interesting decidability results can be achieved if the system is *well-structured* [9, 1, 12].

Definition 8 (well-quasi order)**.** A *well-quasi order (wqo)* over a set X is a quasi-order (a reflexive, transitive relation) $\leq \subseteq X \times X$ s.t. every infinite sequence $\langle x_0, x_1, \ldots \rangle$ in X contains an increasing pair $x_i \leq x_j$ with $i < j$.

We give two examples for wqos on graphs. In our setting, the subgraph order is of crucial importance.

Example 1 (subgraph & minor order)**.**

(i) The subgraph order is given by $G \leq H$ iff there is a total injective morphism $G \hookrightarrow H$. Let $\mathscr{G}_\ell$ be a graph class of bounded path length (with bound ℓ). The restriction of $\leq$ to $\mathscr{G}_\ell$ is a wqo [12, 4]. However, it is not a wqo on all graphs: consider, e.g., the infinite sequence $\langle$  $\rangle$ of cyclic graphs of increasing length.

(ii) The minor order is given by $G \preccurlyeq H$ iff G can be obtained from H by a sequence of edge contractions, node and edge deletions. The minor order is a wqo on all graphs [12, 19].

Assumption. From now on, we implicitly equip every set of graphs with the subgraph order. By $\leq$ we mean either an abstract wqo or the subgraph order, depending on the context.

Definition 9 (closure & basis)**.** Let X be a set and $\leq$ a wqo on X. For every subset A of X, we denote by $\overline{A} = \{x \in X \mid \exists a \in A : a \leq x\}$ the *upward-closure* and $\underline{A} = \{x \in X \mid \exists a \in A : a \geq x\}$ the *downward-closure* of A. If $A = \overline{A}$, then a *basis* of A is a subset $B \subseteq A$ s.t. (i) B *generates* A, i.e., $\overline{B} = A$, and (ii) any two distinct elements in B are *incomparable*, i.e., $\forall b_1, b_2 \in B : b_1 \neq b_2 \Rightarrow b_1 \not\leq b_2$.

Sets A satisfying $A = \overline{A}$ will later be called ideals. For well-structuredness, we demand that the well-quasi order yields a simulation of smaller states by larger states. This condition is called *compatibility*.

Definition 10 (well-structured transition systems)**.** Let $\langle S, \to \rangle$ be transition system and $\leq$ a decidable wqo on S, i.e., for each two given states $s, s' \in S$, it is decidable whether $s \leq s'$. The tuple $\langle S, \leq, \to \rangle$ is a *(strongly) well-structured transition system*, if

(i) The wqo is *(strongly) compatible* with the transition relation, i.e., for all $s_1, s_1', s_2 \in S$ with $s_1 \leq s_1'$ and $s_1 \to s_2$, there exists $s_2' \in S$ with $s_2 \leq s_2'$ and $s_1' \to^* s_2'$ (strongly: $s_1' \to^1 s_2'$).

(ii) For every $s \in S$, a basis of $\overline{\mathrm{pre}(\overline{\{s\}})}$ is computable.

(a) Upwards compatibility (b) Strong upwards compatibility

Figure 2: Visualization of the (strong) upwards compatibility property for transition systems.

In Fig. 2, both versions of compatibility are visualized. The term *(strongly) well-structured transition system* is often abbreviated by (S)WSTS. In Sec. 4, we prove the decidability of resilience for SWSTS. We include the definition of general WSTS for clarity and to point out the differences. Note that for GTSs, strong compatbility is achieved by applying the same (SPO) rule to bigger graphs. However, in DPO, the bigger graph may not fullfill the dangling condition. Consider, e.g., the rule which deletes a node. This rule can be applied to the graph consisting of a single node but not to the graph ○—○ in DPO.

The following result of König & Stückrath terms sufficient conditions for GTSs to be well-structured.

Lemma 1 ([12]). *Every graph transition system of bounded path length is strongly well-structured (equipped with the subgraph order).*

Note that in [12], König & Stückrath consider labeled hypergraphs. However, the proof in this case is the same. The premise of bounded path length seems very restrictive, but we can still capture infinitely many graphs. An usual example are graphs where the "topology" remains unchanged. It is also shown in [12] that every *lossy GTS* is well-structured w.r.t. the minor order and without restriction of the graph class. "Lossy" means that every edge contraction rule is contained in the GTS. However, in this case, we do not obtain strong compatibility.

Assumption. In the following, let $\langle S, \leq, \to \rangle$ be a strongly well-structured transition system.

Upward- and downward-closed sets w.r.t. a given wqo are of special interest. Such sets are called ideals and used in Sec. 3 to define resilience problems for WSTSs.

Definition 11 (ideal). An *ideal* $I \subseteq S$ is an upward-closed set, i.e., $\overline{I} = I$. A *bi-ideal* $J \subseteq S$ is an ideal which is also downward-closed, i.e., $\overline{J} = J = \underline{J}$. An *anti-ideal* $J \subseteq S$ is a downward-closed set, i.e., $\underline{J} = J$. The anti-ideal J is decidable if, given $s \in S$, it is decidable whether $s \in J$.

Example 2 (ideal). Let $\mathscr{G}_\ell$ be a graph class of bounded path length. For every positive constraint c, $I_c = \{G \in \mathscr{G}_\ell \mid G \models c\}$ is an ideal.

Bi-ideals often represent "control states" as in [1]. The notion of anti-ideal is the pendent to ideal. Since a downward-closed set does not have an "upward-basis" in general, we will demand that membership is decidable.

Example 3 (anti-ideal). Let $\mathscr{G}_\ell$ be a graph class of bounded path length. For every negative constraint c, $J_c = \{G \in \mathscr{G} \mid G \models c\}$ is a decidable anti-ideal.

The set of ideals of S is closed under preset, union, and intersection.

Fact 2 (stability of ideals). *Let $I, J \subseteq S$ be ideals. Then the sets* $\mathrm{pre}(I)$*, $I \cup J$, and $I \cap J$ are ideals.*

A major point in our argumentation is the observation that every infinite ascending sequence of ideals w.r.t. a wqo eventually becomes stationary.

Lemma 2 ([1]). *For every infinite ascending sequence $\langle I_0 \subseteq I_1 \subseteq \ldots \rangle$ of ideals, there exists a $k \geq 0$ s.t. $I_k = I_{k+1}$. This directly implies $\exists k_0 \geq 0\ \forall k \geq k_0 : I_k = I_{k_0}$.*

Since ideals are in general infinite, we need a computable finite representation of them. Similar to algebraic structures, ideals are represented by a finite basis (a minimal generating set). Indeed, every ideal has a basis and every basis is finite. We consider bases for complexity reasons. In theory, finite generating sets are sufficient to carry out our approach.

Fact 3 ([1]). *(i) For every ideal $I \subseteq S$, there exists a finite basis B of I. (ii) Given a finite set $A \subseteq S$ with $I = \overline{A}$, we can compute a finite basis B of I.*

2.4 Ideal Reachability

In [1], Abdulla et al. exploit Lemma 2 to show the decidability of *ideal reachability* (also called *coverability*) for strongly well-structured transition systems. The corresponding algorithm forms the basis of our results. We present its basic idea. For any ideal I, another ideal I^* is constructed, s.t. $\exists s' \in I : s \to^* s'$ iff $s \in I^*$. This is clearly the case for $I^* = \mathrm{pre}^*(I) = \bigcup_{j \geq 0} \mathrm{pre}^j(I)$. The idea is to iteratively construct the sequence of the ideals $I^k = \bigcup_{0 \leq j \leq k} \mathrm{pre}^j(I)$ until it becomes stable.

Definition 12 (index)**.** For an ideal $I \subseteq S$ and $k \geq 0$, let $I^k = \bigcup_{0 \leq j \leq k} \mathrm{pre}^j(I) \subseteq I^{k+1}$. The *index* $k(I)$ is the smallest k_0 s.t. $I^k = I^{k_0}$ for all $k \geq k_0$.

Lemma 2 ensures that $k(I)$ always exists. However, we have to show that $I^k = I^{k+1}$ implies that $k(I) \leq k$ to obtain a stop condition. This follows by the observation that $I^{k+1} = I \cup \mathrm{pre}(I^k)$.

Fact 4 (stop condition)**.** *Let $I \subseteq S$ be an ideal and $k \geq 0$ s.t. $I^k = I^{k+1}$, then $I^\ell = I^k$ for all $\ell \geq k$, i.e., $k(I) \leq k$. This also implies that* $\mathrm{pre}^*(I) = I^k$.

Since ideals are infinite, we cannot carry this construction out directly, but we use a basis for representing an ideal. If we can show the computability of a basis in every iteration step, we obtain an algorithm which can decide whether we can reach an ideal I from a given state s.

Lemma 3 ([1])**.** *Given a basis of an ideal $I \subseteq S$, and a state s of a strongly well-structured transition system, we can decide whether we can reach I from s.*

Proof. We have to show that we can compute a basis of I^{k+1} if we are given a basis of I^k. Then the decidability of the stop condition follows directly. Let B be a basis of I^k. We have

$$I^{k+1} = I \cup \mathrm{pre}(I^k) = I \cup \bigcup_{s' \in B} \mathrm{pre}(\overline{\{s'\}}).$$

Since $\mathrm{pre}(\overline{\{s'\}})$ is computable for any $s' \in S$ by definition, we obtain a finite generating set of I^{k+1}. By Fact 3, we can compute a basis of I^{k+1}. □

3 Adverse Conditions and Resilience Problems

We put adverse conditions and resilience on the level of GTSs into context by using joint graph transformation systems [14]. Abstracting from this setting, we identify resilience problems for TSs.

3.1 Joint Graph Transformation Systems

We recapitulate the modeling of adverse conditions by joint graph transformation systems, introduced in [14]. While we considered DPO rules in [14], in order to obtain strong compatbility, SPO is expedient. We define joint graph transformation systems, each of which involves the system, the environment, and an automaton modeling the interaction between them. Both system and environment are GTSs.

Assumption. In the following, let Λ be a fixed label alphabet, and $\mathscr{S}$ and $\mathscr{E}$ be GTSs over Λ, called *system* and *environment*, respectively. W.l.o.g., we assume that $\mathscr{S}$ and $\mathscr{E}$ are disjoint. (If $\mathscr{S}$ and $\mathscr{E}$ share a common rule r, we assign r different names in $\mathscr{S}$ and $\mathscr{E}$.)

We specify the class of automata which are used to regulate the interaction between system and environment. These control automata are similar to ω-automata, see, e.g., [21].

Definition 13 (control automaton)**.** A *control automaton* of $\langle \mathscr{S}, \mathscr{E} \rangle$ is a tuple $A = \langle Q, q_0, \delta, \mathrm{sel} \rangle$ consisting of a finite set Q disjoint from Λ, called the *state set*, an *initial state* $q_0 \in Q$, a *transition relation* $\delta \subseteq Q \times Q$, and a function $\mathrm{sel} : \delta \to \mathfrak{P}(\mathscr{S} \cup \mathscr{E})$ (into the power set of $\mathscr{S} \cup \mathscr{E}$), called the *selection function*.

Remark. Alternatively, the interaction between system and environment can be specified by an ω-regular language [21] over the alphabet $\mathfrak{P}(\mathscr{S} \cup \mathscr{E})$ corresponding to the accepting control automaton. In order to obtain a GTS as joint system, we prefer to use control automata.

A joint graph transformation system is obtained by *synchronizing* the system, repectively, the environment, with the control automaton, and then joining both sets of enriched rules.

Definition 14 (joint graph transformation system)**.** Let be $A = \langle Q, q_0, \delta, \mathrm{sel}\rangle$ a control automaton of $\langle \mathscr{S}, \mathscr{E}\rangle$. The *joint graph transformation system* of $\mathscr{S}$ and $\mathscr{E}$ w.r.t. A is the graph transformation system $\mathscr{S}_A \cup \mathscr{E}_A$, shortly $\mathscr{S}\mathscr{E}$, where for a rule set $\mathscr{R} \in \{\mathscr{S}, \mathscr{E}\}$, the *enriched rule set* $\mathscr{R}_A$ is given by

$$\mathscr{R}_A = \{\langle L, q\rangle \rightharpoonup \langle R, q'\rangle \mid \langle q, q'\rangle \in \delta \text{ and } \langle L \rightharpoonup R\rangle \in \mathscr{R} \cap \mathrm{sel}\langle q, q'\rangle\},$$

and for a graph G and a state q, the tuple $\langle G, q\rangle$ denotes the disjoint union of G and a node labeled with q. In the partial morphism $\langle L, q\rangle \rightharpoonup \langle R, q'\rangle$, the node labeled with q is mapped to the node labeled with q'.

We refine our notion of joint graph transformation systems, namely to *annotated joint graph transformation systems* which also carry the information whether the last applied rule was a system or environment rule. This is realized by a node labeled with "**s**" or "**e**".

Notation. For a joint graph transformation system $\mathscr{S}\mathscr{E}$, the symbol $\mathrm{m}(\mathscr{S}) = \mathbf{s}$ or $\mathrm{m}(\mathscr{E}) = \mathbf{e}$, is the *marker* of $\mathscr{S}$ or $\mathscr{E}$, respectively. For a rule $r \in \mathscr{R}$ and $\mathscr{R} \in \{\mathscr{S}, \mathscr{E}\}$, let $\mathrm{m}(r) = \mathrm{m}(\mathscr{R})$ be the marker of r. The set of all markers $M = \{\top, \mathbf{s}, \mathbf{e}\}$ includes also the symbol $\top$, usually indicating a start graph.

For the explicit construction, we can use *premarkers* to reduce the number of rules. This technical detail is included in App. A.1.

Definition 15 (annotated joint graph transformation system)**.** Let $\mathscr{S}\mathscr{E}$ be a joint graph transformation systems w.r.t. a control automaton $A = \langle Q, q_0, \delta, \mathrm{sel}\rangle$ of $\langle \mathscr{S}, \mathscr{E}\rangle$. The *annotated joint graph transformation system* of $\mathscr{S}$ and $\mathscr{E}$ w.r.t. A is $\mathscr{S}'_A \cup \mathscr{E}'_A$, shortly $(\mathscr{S}\mathscr{E})'$, where for a rule set $\mathscr{R} \in \{\mathscr{S}, \mathscr{E}\}$, $\mathscr{R}'_A$ denotes the *marked rule set*

$$\mathscr{R}'_A = \{\langle L, q, m\rangle \rightharpoonup \langle R, q', m'\rangle \mid \langle L, q\rangle \rightharpoonup \langle R, q'\rangle \in \mathscr{R}_A,\, m \in M,\, m' = \mathrm{m}(\mathscr{R})\}$$

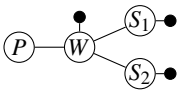
where for a graph G, a state q, and a marker m, $\langle G, q, m\rangle$ denotes the disjoint union of G, a node labeled with q, and a node labeled with m. In the partial morphism $\langle L, q, m\rangle \rightharpoonup \langle R, q', m'\rangle$, the node lableled with m is mapped to the node labeled with m'.

We explicate the state set of annotated joint GTSs. These graphs are of the form $\langle G, q, m\rangle$ for state q of the control automaton and a marker m. We denote a class of all such graphs by $\mathscr{G} \oplus Q \oplus M$. Using such graphs instead of the product of graphs we can directly apply the result (Lemma 1) of [12] for GTSs.

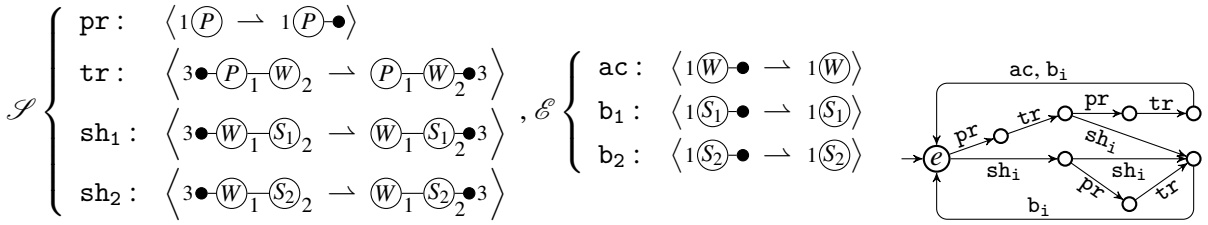
Definition 16 (joint graph transition system)**.** Let $(\mathscr{S}\mathscr{E})'$ be an annotated joint GTS and $\mathscr{G}'$ be a class of graphs which is of the form $\mathscr{G} \oplus Q \oplus M$ and closed under rule application of $(\mathscr{S}\mathscr{E})'$. The graph transition system $\langle \mathscr{G}', \Rightarrow_{(\mathscr{S}\mathscr{E})'}\rangle$ is called *annotated joint graph transition system*.

Note that we usually begin our analysis at a start graph of the form $\langle G, q_0, \top\rangle$.

Example 4 (supply chain)**.** We model a simplified supply chain with graph transformation rules. The infrastructure (topology) is given in the following start graph:



A production site (P) is connected to a warehouse (W) which again is connected to two stores S_1 and S_2. Each of the black nodes indicates one product at the corresponding (connected) location. The behavior in this production chain is modeled by the graph transformation rules in Fig. 3a. The system rules consists of `pr` (the completion of a product at the production site P), `tr` (transporting a product from P



(a) The two GTSs $\mathscr{S}$ (system) and $\mathscr{E}$ (environment) (b) Control automaton

Figure 3: A joint GTS consisting of a GTS for system and environment, each, and a control automaton.

to the warehouse W), and $\mathtt{sh}_1$ and $\mathtt{sh}_2$ (shipping a product from W to one of the two stores S_1, S_2). The environment rules describe external impacts. Namely, $\mathtt{ac}$ describes an accident in the warehouse which leads to the loss of one product, and $\mathtt{b}_1$ and $\mathtt{b}_2$ describe that a product is bought from S_1 or S_2, respectively. The control automaton in Fig. 3b describes the possible order of rule applications. We are interested in the question when the product is again *in stock* (at least 1 product in the warehouse and in each of both stores) whenever a customer buys a product or when an accident in the warehouse happens. After each such transition, the automaton is in the state e. Regardless of the current situation, in 17 steps we can accomplish that the product is in stock by first producing and transporting 6 products with a following accident (3 products will get lost) and shipping them to the stores afterwards. However, what is the minimal number of steps in which we can reach a situation where the product is in stock whenever someone bought a product or a product got lost?

We will come back later to that question in Ex. 4. We describe the setting for joint GTSs which we investigate: Consider a safety condition c, given as positive constraint, and the set of graphs $I_c = \{G' \in \mathscr{G} \oplus Q \oplus M \,|\, G' \models c\}$ which satisfy c. Similarly, let $J_{\mathbf{e}} = \{G' \in \mathscr{G} \oplus Q \oplus M \,|\, G' \models \exists \mathbf{e}\}$ (all graphs obtained by an environment interference; $\exists \mathbf{e}$ means that there exists a node labeled with $\mathbf{e}$). The environment is usually modeled in a such way that it has an adverse effect on the satisfaction of c. *Resilience* in this context means that the system can withstand such an adverse condition. We ask whether we can reach a graph in I_c in a reasonable amount of time whenever we reach a graph in $J_{\mathbf{e}}$. By a "reasonable amount of time", we mean either that a number k of steps is given in which I_c should be reached (*explicit resilience*), or that I_c should be reached in a bounded number of steps (*bounded resilience*).

Another approach is to consider the set $J_{\neg c} = \{G' \in \mathscr{G} \oplus Q \oplus M | G' \not\models c\}$ instead of $J_{\mathbf{e}}$. So, we ask whether we can reach a graph which satisfies c in a bounded amount of time/in at most k steps whenever we reach a graph which does not satisfy c, i.e., an error state. Both instances of the problem are reasonable, and if we can give a positive answer for the latter one, we can also give a positive answer for the first one. We focus on the first problem (adverse conditions), but the results which we will obtain abstract from a specific J and therefore also apply to the latter one (error states).

3.2 Abstract Resilience Problems

The previous motivation gives rise to a more abstract definition of resilience problems, namely in the framework of TSs. When we explicate state set, every GTS can be interpreted as a TS.

We assume that a TS $\langle S, \to \rangle$ comes along with a set of propositions each of which is either satisfied or not satisfied by a state of the TS. Let SAFE (*safety property*) and BAD (*bad* condition) be propositions. Note that BAD is not necessarily equivalent to ¬SAFE. We ask whether we can reach a state which satisfies SAFE in a reasonable amount of time whenever we reach a state which satisfies BAD. From

this we formulate two resilience problems. First consider the case where the recovery time is bound by a natural number $k \geq 0$, i.e., the *(abstract) explicit resilience problem*.

EXPLICIT RESILIENCE PROBLEM

Given: A state s of a TS $\langle S, \to \rangle$, propositions SAFE and BAD, a natural number $k \geq 0$.

Question: $\forall s' \in S : (s' \models \text{BAD} \wedge s \to^* s') \Rightarrow \exists s'' \in S : s' \to^{\leq k} s'' \wedge s'' \models \text{SAFE}$?

If we assume that the transition system yields infinite sequences of transitions, we can express the property to be evaluated in CTL by $s \models \mathbf{AG}(\text{BAD} \to \bigvee_{0 \leq j \leq k} \mathbf{EX}^j \text{SAFE})$. We can also ask whether there exists such a bound k. We call this problem the *(abstract) bounded resilience problem*.

BOUNDED RESILIENCE PROBLEM

Given: A state s of a TS $\langle S, \to \rangle$, propositions SAFE and BAD.

Question: $\exists k \geq 0 \; \forall s' \in S : (s' \models \text{BAD} \wedge s \to^* s') \Rightarrow \exists s'' \in S : s' \to^{\leq k} s'' \wedge s'' \models \text{SAFE}$?

Both problems are undecidable: For SAFE $=$ `false`, resilience is equivalent to reachability of BAD.

4 Decidability Results

Many interesting decidability results can be obtained if we assume that a transition system is well-structured [1, 9, 12]. We formulate the resilience problems from above for WSTSs and show decidability of both, the explicit and the bounded resilience problem in the setting of SWSTSs.

4.1 Resilience Problems in a Well-structured Framework

Properties in well-structured transition systems are often given as upward- or downward closed sets [1, 9]. Ideals enjoy suitable features for verification such as finite representation and stability, and anti-ideals are their complements (cp. Sec. 2.3). Transfering the abstract resilience problems into this framework, it is therefore reasonable to demand that both propositions, SAFE and BAD, are given by ideals or anti-ideals. For our purpose, the following setting suits very well: we assume that the safety property is an ideal and the bad condition is a decidable anti-ideal.

From these considerations, we formulate "instances" of the abstract resilience problems for well-structured transition systems. Again, we first consider the case where the recovery time is bounded by a $k \in \mathbb{N}$, the *explicit resilience problem for WSTSs*.

EXPLICIT RESILIENCE PROBLEM FOR WSTSS

Given: A state s of a WSTS $\langle S, \leq, \to \rangle$, a basis of $\overline{\text{post}^*(s)}$, an ideal I with a given basis, a decidable anti-ideal J, a natural number $k \geq 0$.

Question: $\forall s' \in J : (s \to^* s') \Rightarrow \exists s'' \in I : s' \to^{\leq k} s''$?

Analogously, we formulate the *bounded resilience problem for WSTSs*.

BOUNDED RESILIENCE PROBLEM FOR WSTSS

Given: A state s of a WSTS $\langle S, \leq, \to \rangle$, a basis of $\overline{\text{post}^*(s)}$, an ideal I with a given basis, a decidable anti-ideal J.

Question: $\exists k \geq 0 \; \forall s' \in J : (s \to^* s') \Rightarrow \exists s'' \in I : s' \to^{\leq k} s''$?

From now on, we mean one of the previously defined resilience problems for WSTSs if we speak of a resilience problem. If the answer of the bounded (explicit) resilience problem is positive, we say that $\langle S, \leq, \rightarrow \rangle$ is *resilient (k-step resilient)* w.r.t. I and J starting from s. In this context, s is a *start state/graph*.

Remark. The premise that a basis of $\overline{\text{post}^*(s)}$ is given is a strong but reasonable assumption. In general, we cannot simply compute the sequence of ideals $P_k = \bigcup_{0 \leq j \leq k} \overline{\text{post}^j(s)}$ until it becomes stationary. This sequence does become stationary by Lemma 2. However, in contrast to the case in Lemma 3, $P_{k+1} = P_k$ is not a sufficient stop condition. So, this way it is not algorithmically checkable when we have reached k_0 s.t. $P_\ell = P_{k_0}$ for every $\ell \geq k_0$. However, we investigate resilience of GTSs each of which constitutes a SWSTS. A sufficient condition for strong well-structuredness is boundedness of the path length (cp. Lemma 1). This holds, e.g., for graph classes where the "topology" is static. For these graph classes, a basis of all successors is often easier to determine than in general. A typical example for such GTSs are Petri nets (Sec. 5.3). For Petri nets, it is even computable (App. A.3).

4.2 Decidability

Abdulla et al. show in [1] that ideal reachability is decidable for SWSTSs (cp. Lemma 3). Finkel & Schnoebelen [9] show that ideal reachability (or *coverability*) is also decidable for WSTSs. Both algorithms coincide in the case of strong well-structuredness. König & Stückrath [12] use the algorithm of [9] for the *backwards analysis* for (generalized) well-structured GTSs.

The main difference between the algorithms in [1] and [9] is that for (not necessarily strongly) WSTSs, $\text{pre}(I')$ in general, for any ideal I', is not an ideal. Thus, Finkel & Schnoebelen consider in every iteration step the ideal $\overline{\text{pre}(I')}$ instead of $\text{pre}(I')$. Now the same arguments like before hold (cp. Sec. 2.4) and a basis of $\text{pre}^*(I) = \overline{\text{pre}^*(I)}$ for a given ideal I can be computed.

We are interested in the exact number of steps which we need to reach an ideal. Thus, $\text{pre}(I')$ should be an ideal and we cannot use the technique from [9] for WSTSs. We need to restrict our setting to *strongly* WSTSs like in [1]. First, we state our main result for SWSTSs, the decidability of resilience.

Theorem 1 (decidability of resilience)**.** *The explicit and the bounded resilience problem are both decidable for strongly well-structured transition systems.*

We prove this theorem by giving a respective algorithm. It exploits a modified version of the ideal reachability algorithm in [1] (cp. Lemma 3). We check in every iteration step inclusion in $I^k = \bigcup_{0 \leq j \leq k} \text{pre}^j(I)$. Before doing so, we need a finite representation of $\text{post}^*(s) \cap J$ to check the inclusion in an ideal I'. The next lemma uses that J and I' are downward- and upward-closed, respectively.

Lemma 4 (intersection with anti-ideal)**.** *Let $A \subseteq S$ be a set, $J \subseteq S$ an anti-ideal and $I' \subseteq S$ an ideal. Then $A \cap J \subseteq I' \Leftrightarrow \overline{A} \cap J \subseteq I'$.*

Proof. "$\Leftarrow$": Holds since $A \subseteq \overline{A}$.
"$\Rightarrow$": Let $s' \in \overline{A} \cap J$. Then $s' \in J$ and there exists $s'' \in A$ s.t. $s'' \leq s'$. Thus, $s'' \in \underline{J} = J$. So we have $s'' \in A \cap J \subseteq I'$. Hence, $s' \in \overline{I'} = I'$. □

This lemma enables us to prove Thm. 1 given above. We iteratively determine the minimal k satisfying $\text{post}^*(s) \cap J \subseteq I^k$ (or stop, if there does not exist such k).

Proof of Theorem 1. Let B_{post} be a basis of $\overline{\text{post}^*(s)}$, B_0 a basis of I, and J a decidable anti-ideal. For every $k \geq 0$, I^k is an ideal due to strong compatibility. By applying Lemma 4 twice, we obtain

$$\text{post}^*(s) \cap J \subseteq I^k \quad \Leftrightarrow \quad B_{\text{post}} \cap J \subseteq I^k$$

for any $k \geq 0$. Since B_{post} is finite and J is a decidable anti-ideal, we can directly compute $B_{\text{post}} \cap J$. We perform a modification of the ideal reachability algorithm: Iteratively check whether $B_{\text{post}} \cap J \subseteq I^k$. If this is the case, return $k_{\min} = k$. Otherwise check whether $I^{k+1} = I^k$. If so, return -1 (`false`), otherwise continue. We have to make sure that every iteration step is decidable. In fact, we can compute a basis of I^{k+1} if we have a basis of I^k. This follows by the proof of Lemma 3. The stop condition is decidable and by Fact 4 also sufficient. Soundness and completeness follow by the previous considerations and the fact that

$$\text{post}^*(s) \cap J \subseteq I^k \quad \Leftrightarrow \quad (\forall s' \in J : (s \to^* s') \Rightarrow \exists s'' \in I : s' \to^{\leq k} s'')$$

for any $k \geq 0$. Termination is guaranteed by Lemma 2.

To sum up, our algorithm decides whether there exists a $k \geq 0$ s.t. $\text{post}^*(s) \cap J \subseteq I^k$, and returns the minimal such k in the positive case. Thus, it decides the bounded resilience problem. Given any k, we can check whether $k_{\min} \leq k$ and therefore decide the explicit resilience problem. □

We denote the above described algorithm deciding resilience by MINIMALSTEP(B_{post}, J, B_0) and the used procedure returning a basis of $\text{pre}(\overline{B'})$ by PREBASIS(B'). We give the detailed algorithm in pseudocode in App. A.2.

Remark. In our proof, it was crucial that we have *strong* compatibility. This approach does not work for WSTSs in general. We loose precision when we only demand compatibility. Thus, we conjecture that both resilience problems are undecidable for WSTSs in general, but this question remains still open.

5 Application to Graph Transformation Systems

We apply our abstract results to (joint) graph transformation systems and present a framework for verifying resilience of GTSs. We exemplarily show how Petri nets fit in this setting.

We considered ideals as safety, and decidable anti-ideals as "bad" properties. In the setting of well-structured GTSs w.r.t. the subgraph order, these can be expressed as positive and negative constraints. Recall that for a class $\mathscr{G}$ of graphs, $I_c = \{G \in \mathscr{G} \mid G \models c\}$ for a positive constraint c, and $J_{c'} = \{G \in \mathscr{G} \mid G \models c'\}$ for a negative constraint c'.

Fact 5 (ideals of graphs)**.** *Let $\mathscr{G}_\ell$ be a class of graphs of bounded path length. Let $I, J \subseteq \mathscr{G}_\ell$ be sets.*

(i) I is an ideal. $\Leftrightarrow I = I_c$ for a positive constraint c.

(ii) J is a decidable anti-ideal. $\Leftrightarrow J = J_c$ for a negative constraint c.

Thus, for GTSs, our safety conditions are equivalent to positive constraints and bad conditions are equivalent to negative constraints.

Remark. In general, more expressive graph constraints [17, 10], e.g., $\forall(\,\overset{1}{\circ}\,, \exists(\,\overset{1}{\circ}\,\overset{2}{\circ}\,))$, do not consitute ideals w.r.t. the subgraph order, as the relation $\overset{1}{\circ}\,\overset{2}{\circ} \leq \overset{1}{\circ}\,\overset{2}{\circ}\,\circ$ shows. However, for particular GTSs, nested graph constraints may yield ideals.

5.1 Verifying Resilience of Graph Transformation Systems

Using the sufficient conditions for strong well-structuredness of König & Stückrath [12], we obtain the decidability of both resilience problems for a subclass of GTSs. We need to use the subgraph order as wqo. Thus, we have the restriction of boundedness of the path length for the generated graph class. Instead of considering GTSs, we consider graph transition systems, i.e., we always explicate the state set. Thm. 1 and the result in [12] (see Lemma 1) imply our main result for GTSs:

Theorem 2 (decidability of resilience for well-structured GTSs)**.** *The explicit and the bounded resilience problem are decidable for graph transition systems which are of bounded path length (and equipped with the subgraph order).*

As joint GTSs are also GTSs, the same sufficient conditions for strong well-structuredness apply.

Fact 6 (strongly well-structured joint GTSs)**.** *Every annotated joint graph transition system which is of bounded path length is strongly well-structured (equipped with the subgraph order).*

An immediate consequence of Thm. 2 and Fact 6 is the following:

Corollary 1 (decidability of resilience for joint GTSs)**.** *The explicit and the bounded resilience problem are decidable for annotated joint graph transition systems which are of bounded path length (and equipped with the subgraph order).*

Thus, we can apply the algorithm MINIMALSTEP described in Sec. 4.2 to verify resilience of annotated joint graph transition systems. We consider an ideal I_c for a positive constraint c with a given basis B_c. The anti-ideal (bi-ideal) is given by $\underline{J_{\mathbf{e}}} = \{G' \in \mathscr{G} \oplus Q \oplus M \mid G' \models \exists \mathbf{e}\}$. We assume that a start graph $G \in \mathscr{G} \oplus \{q_0\} \oplus \{\top\}$ and a basis B_G of $\underline{\text{post}^*(G)}$ are given. The PREBASIS procedure for the subgraph order needed in the algorithm is given by König & Stückrath in [12] (and more detailed in [20]).

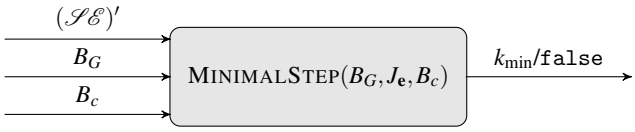
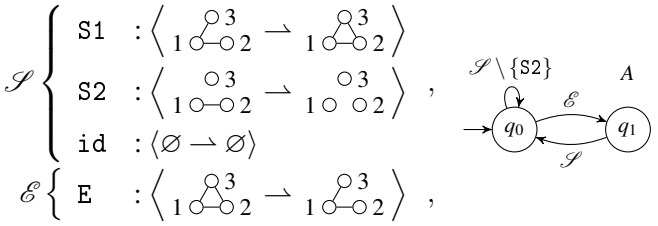


Figure 4: Verifying resilience in the adverse conditions approach.

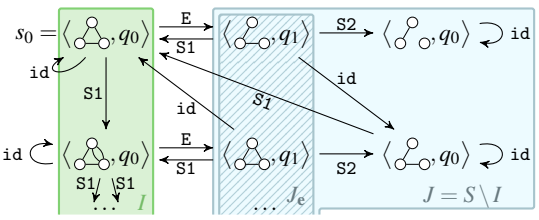
5.2 Adverse Conditions vs. Error States

We compare the adverse conditions approach with the error state approach. As pointed out, these two views of resilience are not equivalent. While every system that is resilient w.r.t. error states (i.e., $J = S \setminus I$) is also resilient w.r.t. adverse conditions (i.e., $J = J_{\mathbf{e}}$) due to $J_{\mathbf{e}} \setminus I \subseteq S \setminus I$ (meaning that if we can reach I from *every* state, then also from every state in $J_{\mathbf{e}}$), the opposite is not true in general.

We do not define a restriction on the system/environment to allow more freedom of modeling but our counterexample in Fig. 5 captures the adverse effect of the environment. The joint GTS in Fig. 5a,



(a) Components of the joint GTS



(b) The state set of the joint GTS

Figure 5: A joint GTS example that is 1-step resilient w.r.t. $J_{\mathbf{e}}$ (adverse conditions), but not resilient w.r.t. $J = S \setminus I$ (error states).

together with a start graph  , results in the state set in Fig. 5b. A basis of I is given by $\langle$$, q_0\rangle$. From the definition of A, we see that $J_{\mathbf{e}}$ is given by $\{\langle G, q_1\rangle \mid$  $\leq G\}$, indicated by the hatched area. We see that from every reachable state in $J_{\mathbf{e}}$ we can reach I in one step, which means that the system is 1-step resilient w.r.t. adverse conditions. On the other hand, we cannot reach I from the state $\langle$  $, q_0\rangle \in S \setminus I$, which is reachable from $J_{\mathbf{e}}$ when the "wrong" system rule is applied. This means the system is *not* resilient w.r.t. error states.

If, due to the structure of a joint GTS, we can reach $J_{\mathbf{e}}$ from every reachable error state, as, e.g., in Ex. 4, both approaches coincide. The computed $k_{\min}$'s then only differ by at most the index $k(J_{\mathbf{e}})$.

5.3 An Example Class: Petri Nets

Petri nets [16] are a common model for discrete distributed systems in computer science, often applied, e.g., in logistics or supply chains [24]. It is a classical example for strongly well-structured (graph) transition systems. We will give a definition of Petri nets and show how our example fits in this setting.

Definition 17 (Petri nets)**.** A *Petri net* is a tuple $N = \langle P, T, F\rangle$ with disjoint finite sets of *places* P and *transitions* T, and a *flow function* $F : (P\times T)\cup(P\times T) \to \mathbb{N}$. A *marking* in N is a multi-set $M : P \to \mathbb{N}$ that indicates the number of tokens on each place. $F(x,y) = n > 0$ means there is an *arc* of *weight* n from node x to y describing the flow of tokens in the net. A transition $t \in T$ is *enabled* in a marking M if $\forall p \in P : F(p,t) \leq M(p)$. If t is enabled, then t can *fire* in M, leading to a new marking M' calculated by $\forall p \in P : M'(p) = M(p) - F(p,t) + F(t,p)$. This is denoted by $M[t\rangle M'$. Usually, a Petri net N is equipped with an *initial marking* M_0. The tuple $\langle N, M_0\rangle$ is then called a *marked Petri net*.

Any Petri net N can be interpreted as a transition system with the states S given by $\mathscr{M}(N)$, the set of all markings of N, and the transitions $\to$ given by $M \to M' \Leftrightarrow \exists t \in T : M[t\rangle M'$. Together with the wqo $\leq_N$, given by $\forall M, M' \in \mathscr{M}(N) : M \leq_N M' :\Leftrightarrow \forall p \in P : M(p) \leq M'(p)$, this constitutes a SWSTS. For Petri nets, many problems, e.g., reachability or coverability, are decidable [16, 7, 23]. From this fact, one can show that a basis of $\overline{\text{post}^*(M_0)}$ is computable. We give more details on this topic in App. A.3.

Petri nets can also be seen as an instance of GTSs, as shown in [2]. From that point of view, every transition corresponds to a graph transformation rule. A marking is given by the structure of the Petri net represented as a graph, with the number of tokens on a place represented by extra nodes connected to it, as in Fig. 1c. The wqo $\leq_N$ then directly corresponds to the subgraph order. Together with the start graph representing the initial marking, interpreting the GTS as a WSTS results in exactly the same SWSTS above. This means we can apply the algorithm deciding resilience in GTS to Petri nets. We demonstrate this by the following example, where we consider a Petri net that, when interpreted as a GTS, is exactly the supply chain modeled in Ex. 4.

Example 5 (supply chain as Petri net)**.** We consider a marked Petri net modeling a simplified scenario of a supply chain, shown in Fig. 6. As usual we depict places as circles, transitions as rectangles, and the flow as weighted directed arcs between them. In the example, all weights are 1 and therefore not indicated. Dots on places indicate the number of tokens on the respective place in the initial marking.

The Petri net corresponds directly to the graph transformation rules in Ex. 4, with the blue transitions simulating $\mathscr{S}$, and the red (checkered) transitions simulating $\mathscr{E}$. The initial marking represents the start graph. Correspondingly, the control automaton has the same structure as in Ex. 4, with transitions replacing rules. Let $I = \{\langle M, q\rangle \mid M(\mathit{warehouse}), M(\mathit{store}_1), M(\mathit{store}_2) \geq 1 \wedge q \in Q\}$, i.e., in the warehouse and in both stores products are available for shipping or purchase, respectively. The transitions corresponding to $\mathscr{E}$ reduce the number of tokens in the net. We consider the resilience problem with adverse conditions. By definition of the control automaton, we know that $J_{\mathbf{e}} = \{\langle M, e\rangle \mid M \text{ is a marking}\}$.

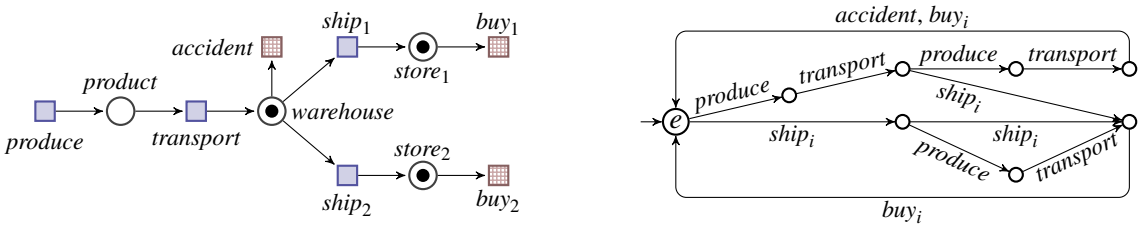


Figure 6: A Petri net modeling a supply chain, and its control automaton.

We interpreted the Petri net as a GTS and applied a prototype implementation of the algorithm MINIMALSTEP from Sec. 4.2 to it. We obtained that $k_{\min} = 6$ is the smallest k for which the system is k-step resilient. More detailed results of the computations are shown in App. A.4.

6 Related Work

We use SPO graph transformation for modeling systems as in **Ehrig et al.** [6] (see also **Löwe** [13]).

Our notion of joint GTSs is a special case of graph-transformational interacting systems. Another approach considering dependencies can be found, e.g., in **Corradini et al.** [3].

The concept of resilience is broadly used in different areas, e.g., in industrial control systems [22, 18], with varying definitions. Following these ideas, we formulated resilience in the abstract settings of TSs and GTSs. Our interpretation of resilience captures recovery in bounded time.

Abdulla et al. [1] show the decidability of ideal reachability (coverability), eventuality properties and simulation in (labeled) SWSTSs. We use the presented algorithm to show the decidability of resilience problems in SWSTSs.

Finkel & Schnoebelen [9] show that the concept of well-structuredness is ubiquitous in computer science by providing a large class of example models (e.g., Petri nets and their extensions, communicating finite state machines, lossy systems, basic process algebras). Moreover, they give several decidability results for systems with different degrees of well-structuredness. They also generalize the algorithm of [1] to (not necessarily strongly) WSTSs to show decidability of coverability.

König & Stückrath [12] extensively study the well-structuredness of GTSs. More detailed considerations can be found in [20]. They identify three types of wqos (minor, subgraph, induced subgraph) on graphs based on results of **Ding** [4] and **Robertson & Seymour** [19]. The fact that the subgraph order is a wqo on graphs of bounded path length while the minor order allows all graphs comes with a trade-off: For obtaining well-structuredness w.r.t. the minor order, the GTS must contain all edge contraction rules, i.e., it must be a "lossy" GTS. On the other hand, all GTSs (without application conditions) are strongly well-structured on graphs of bounded path length w.r.t. the subgraph order. This result enables us to apply our abstract results to GTSs (in particular, we use the pred-basis procedure in the case of the subgraph order for our algorithm). In our setting, the regarded wqo is the subgraph order since it yields strong compatibility. They also generalize the notion of well-structured transition systems by regarding Q-restricted WSTSs whose state sets needs not to be a wqo but rather a subset Q of the states is a wqo. König & Stückrath develope a backwards algorithm based on [9] for Q-restricted WSTSs obtaining decidability of coverability under additional assumptions. For SWSTSs, this approach coincides with the ideal reachability algorithm [1].

All in all, our result for SWSTSs uses a modification of Abdulla et. al [1], and our application to GTSs additionally uses the predecessor-basis procedure from König & Stückrath [12] in every computation

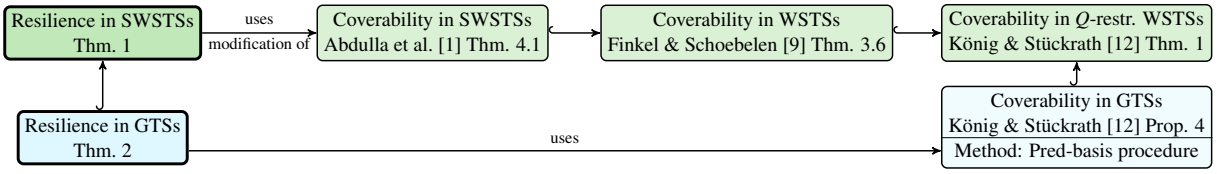


Figure 7: Relation of the decidability results for resilience (bold) and the results in related work. The bottom (blue) and the top (green) layers contain decidability results for GTSs and WSTSs, respectively. The hooked arrows (↪) mean "generalized to" or "instance of".

step. It can also be seen as a modification of the backwards analysis of König & Stückrath [12] in the case of the subgraph order. We summarize the relations of our results and the used concepts in Fig. 7.

7 Conclusion

We (1) provided a definition of resilience in an abstract framework, namely the explicit and the bounded resilience problem, (2) proved decidability of both problems for strongly well-structured transition systems, and, (3) by applying them, obtained decidability results for GTSs of bounded path length, and in particular a verification framework for GTSs which incorporates adverse conditions. Our approach does not work for WSTSs in general/lossy GTSs for which we conjecture that both problems are undecidable.

For future work, we will investigate on (i) the (un)decidability of resilience for WSTSs/lossy GTSs, (ii) synthesis of resilient GTSs, i.e., using the presented approach to construct provably resilient GTSs, and (iii) the computability of a basis of the upward-closure of all successors for (a subclass of) strongly well-structured GTSs. Regarding (ii), we will investigate on the construction of strongly well-structured GTSs since it is a requirement for the application of the presented concept.

Acknowledgement. We are grateful to Annegret Habel, Christian Sandmann, and the anonymous reviewers for their helpful comments to this paper.

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson & Yih-Kuen Tsay (1996): *General Decidability Theorems for Infinite-State Systems*. In: *Proc. LICS 1996*, IEEE Computer Society Press, pp. 313–321, doi:10.1109/LICS.1996.561359.

[2] Paolo Baldan, Andrea Corradini, Fabio Gadducci & Ugo Montanari (2010): *From Petri Nets to Graph Transformation Systems*. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 26, doi:10.14279/tuj.eceasst.26.368.

[3] Andrea Corradini, Luciana Foss & Leila Ribeiro (2008): *Graph Transformation with Dependencies for the Specification of Interactive Systems*. In: *Proc. WADT 2008*, *LNCS* 5486, Springer, pp. 102–118, doi:10.1007/978-3-642-03429-9_8.

[4] Guoli Ding (1992): *Subgraphs and well-quasi-ordering*. *J. Graph Theory* 16(5), pp. 489–502, doi:10.1002/jgt.3190160509.

[5] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/3-540-31188-2.

[6] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner & Andrea Corradini (1997): *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach*. In Grzegorz Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, World Scientific, pp. 247–312, doi:10.1142/9789812384720_0004.

[7] Javier Esparza & Mogens Nielsen (1994): *Decidability Issues for Petri Nets*. *BRICS Report Series* 1(8), doi:10.7146/brics.v1i8.21662.

[8] Alain Finkel (1987): *A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems*. In: *Proc. ICALP 1987*, *LNCS* 267, Springer, pp. 499–508, doi:10.1007/3-540-18088-5_43.

[9] Alain Finkel & Philippe Schnoebelen (2001): *Well-structured transition systems everywhere!* *Theor. Comput. Sci.* 256(1-2), pp. 63–92, doi:10.1016/S0304-3975(00)00102-X.

[10] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. *Math. Struct. Comput. Sci.* 19(2), pp. 245–296, doi:10.1017/S0960129508007202.

[11] Richard M. Karp & Raymond E. Miller (1969): *Parallel Program Schemata*. *J. Comput. Syst. Sci.* 3(2), pp. 147–195, doi:10.1016/S0022-0000(69)80011-5.

[12] Barbara König & Jan Stückrath (2017): *Well-structured graph transformation systems*. *Inf. Comput.* 252, pp. 71–94, doi:10.1016/j.ic.2016.03.005.

[13] Michael Löwe (1991): *Extended algebraic graph transformation*. Ph.D. thesis, Technical University of Berlin, Germany. Available at `http://d-nb.info/910935696`.

[14] Okan Özkan (2020): *Modeling Adverse Conditions in the Framework of Graph Transformation Systems*. In: *Proc. GCM@STAF 2020*, *EPTCS* 330, pp. 35–54, doi:10.4204/EPTCS.330.3.

[15] Subhav Pradhan, Abhishek Dubey, Tihamer Levendovszky, Pranav Srinivas Kumar, William A. Emfinger, Daniel Balasubramanian, William Otte & Gabor Karsai (2016): *Achieving resilience in distributed software systems via self-reconfiguration*. *Journal of Systems and Software* 122, pp. 344–363, doi:10.1016/j.jss.2016.05.038.

[16] Wolfgang Reisig (1985): *Petri Nets: An Introduction*. *EATCS Monographs on Theoretical Computer Science* 4, Springer, doi:10.1007/978-3-642-69968-9.

[17] Arend Rensink (2004): *Representing First-Order Logic Using Graphs*. In: *Proc. ICGT 2004*, *LNCS* 3256, Springer, pp. 319–335, doi:10.1007/978-3-540-30203-2_23.

[18] Craig G. Rieger, Kevin L. Moore & Thomas L. Baldwin (2013): *Resilient control systems: A multi-agent dynamic systems perspective*. In: *Proc. EIT 2013*, IEEE, pp. 1–16, doi:10.1109/EIT.2013.6632721.

[19] Neil Robertson & Paul D. Seymour (2004): *Graph Minors. XX. Wagner's conjecture*. *J. Comb. Theory, Ser. B* 92(2), pp. 325–357, doi:10.1016/j.jctb.2004.08.001.

[20] Jan Stückrath (2016): *Verification of Well-Structured Graph Transformation Systems*. Ph.D. thesis, University of Duisburg-Essen. Available at `https://nbn-resolving.org/urn:nbn:de:hbz:464-20160425-093027-1`.

[21] Wolfgang Thomas (1990): *Automata on Infinite Objects*. In Jan van Leeuwen, editor: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier and MIT Press, pp. 133–191, doi:10.1016/b978-0-444-88074-1.50009-3.

[22] Kishor S. Trivedi, Dong Seong Kim & Rahul Ghosh (2009): *Resilience in computer systems and networks*. In: *Proc. ICCAD 2009*, ACM, pp. 74–77, doi:10.1145/1687399.1687415.

[23] Rüdiger Valk & Matthias Jantzen (1985): *The Residue of Vector Sets with Applications to Decidability Problems in Petri Nets*. *Act. Inf.* 21, pp. 643–674, doi:10.1007/BF00289715.

[24] Xiaoling Zhang, Qiang Lu & Teresa Wu (2009): *Petri-net based application for supply chain management: An overview*. In: *Proc. IEEM 2009*, pp. 1406–1410, doi:10.1109/IEEM.2009.5373050.

A Appendix

A.1 Annotated Joint Graph Transformation Systems

Notation. For a joint graph transformation system $\mathscr{SE}$, the symbol $\mathrm{m}(\mathscr{S}) = \mathbf{s}$ or $\mathrm{m}(\mathscr{E}) = \mathbf{e}$, is the *marker* of $\mathscr{S}$ or $\mathscr{E}$, respectively. For a rule $r \in \mathscr{R}$ and $\mathscr{R} \in \{\mathscr{S}, \mathscr{E}\}$, let $\mathrm{m}(r) = \mathrm{m}(\mathscr{R})$ be the marker of r. Let $A = \langle Q, q_0, \delta, \mathrm{sel}\rangle$ be the regarded control automaton. We define

$$\mathrm{prem}(r) = \cup_{q \in Q_{\mathrm{prem}}(r)} \{\mathrm{m}(r') | r' \in \mathscr{S} \cup \mathscr{E} : q \in Q_{\mathrm{postm}}(r')\} \cup \{\top | q = q_0\} \subseteq \{\top, \mathbf{s}, \mathbf{e}\}$$

as *premarkers* of r, where $Q_{\mathrm{prem}}(r) = \{q \in Q | \exists q' \in Q : \langle q, q'\rangle \in \delta, r \in \mathrm{sel}\langle q, q'\rangle\}$ and $Q_{\mathrm{postm}}(r') = \{q \in Q | \exists q'' \in Q : \langle q'', q\rangle \in \delta, r' \in \mathrm{sel}\langle q'', q\rangle\}$.

The set $\mathrm{prem}(r)$ of premarkers of a rule r depicts all markers of rules which could have been applied prior to an application of r. We give an alternative definition of annotated joint GTSs using premarkes.

Definition 18 (annotated joint graph transformation system)**.** Let $\mathscr{SE}$ be a joint graph transformation systems w.r.t. a control automaton $A = \langle Q, q_0, \delta, \mathrm{sel}\rangle$ of $\langle \mathscr{S}, \mathscr{E}\rangle$. The *annotated joint graph transformation system* of $\mathscr{S}$ and $\mathscr{E}$ w.r.t. A is $\mathscr{S}'_A \cup \mathscr{E}'_A$, shortly $(\mathscr{SE})'$, where for a rule set $\mathscr{R} \in \{\mathscr{S}, \mathscr{E}\}$, $\mathscr{R}'_A$ denotes the *marked rule set*

$$\mathscr{R}'_A = \{\langle L, q, m\rangle \rightharpoonup \langle R, q', m'\rangle \,|\, \langle L, q\rangle \rightharpoonup \langle R, q'\rangle \in \mathscr{R}_A, m \in \mathrm{prem}(L \rightharpoonup R), m' = \mathrm{m}(\mathscr{R})\}$$

where for a graph G, a state q, and a marker m, $\langle G, q, m\rangle$ denotes the disjoint union of G, a node labeled with q, and a node labeled with m. In the partial morphism $\langle L, q, m\rangle \rightharpoonup \langle R, q', m'\rangle$, the node lableled with m is mapped to the node labeled with m'.

A.2 Minimal k Algorithm

We give the algorithm for deciding resilience for general SWSTSs in pseudocode as Alg. 1 on p. 18. The algorithm uses two methods, namely PREBASIS and MIN. PREBASIS(B) computes the basis of $\overline{B}$ for a set B. It is shown in [12], that such a prebasis is computable for GTSs, and described in detail in [20]. The method MIN(B) minimizes a finite set B by deleting every element in B for which there is already a smaller element in B.

A.3 Basis calculation for Petri nets

In Sec. 5.3 we statet that for every marked Petri net $\langle N, M_0\rangle$, a basis of $\overline{\mathrm{post}^*(M_0)}$ is computable. We elaborate on this statement. In [23], it is shown that for any ideal I of markings in a Petri net, a basis of I is computable iff for every ω-marking M it is decidable whether $I \cap \underline{\{M\}} = \varnothing$. An ω-marking is a function $M : P \to \mathbb{N} \cup \{\omega\}$, and $\underline{\{M\}} := \{M' \in \mathscr{M}(N) \,|\, \forall p \in P : M'(p) \leq M(p) \vee M(p) = \omega\}$ Since $\overline{\mathrm{post}^*(M_0)}$ is an ideal, we can apply this result and ask whether $\overline{\mathrm{post}^*(M_0)} \cap \underline{\{M\}} = \varnothing$ is decidable. This is obviously equivalent to $\overline{\mathrm{post}^*(M_0)} \cap \underline{\{M\}} \subseteq \varnothing$, which allows us to apply Lemma 4, since $\varnothing$ is an ideal. Thus, we now ask whether

$$\mathrm{post}^*(M_0) \cap \underline{\{M\}} = \varnothing.$$

This problem corresponds to the so-called *submarking reachability problem*, which is decidable (cp., e.g., [7]), since it is recursively equivalent the to reachability problem. Therefore, we get that a basis of $\overline{\mathrm{post}^*(M_0)}$ is computable.

Algorithm 1 Minimal k Algorithm

procedure MINIMALSTEP(B_{post}, J, B_0) ▷ $k_{\min}$ (minimal upper bound for recovery time)/-1
  $B \leftarrow B_{\text{post}} \cap J$ ▷ compute B by taking out elements which are not in J
  $k \leftarrow 0$ ▷ increasing counter
  $B_1 \leftarrow B_0$ ▷ basis of the current I^k; B_0 is a given basis of I
  $B_2 \leftarrow \varnothing$ ▷ basis of the current I^{k+1}
  while true **do**
    if $B \subseteq \overline{B_1}$ **then**
      return k ▷ we found $k_{\min}$
    else
      $B_2 \leftarrow B_0 \cup \text{PREBASIS}(B_1)$ ▷ PREBASIS(B_1) computes the basis of $\overline{B_1}$
      $B_2 \leftarrow \text{MIN}(B_2)$ ▷ MIN(B_2) minimizes the set B_2
      if $B_2 \subseteq \overline{B_1}$ **then**
        return -1 ▷ there exists no such k
      else
        $B_1 \leftarrow B_2$ ▷ continue
        $k \leftarrow k+1$
      end if
    end if
  end while
end procedure

A.4 Computations

We implemented Ex. 4/ Ex. 5 as joint GTS to compute a basis of the predecessors in every step of the minimal k algorithm. The following set $B_{\mathbf{e}}^{M_0}$ is the intersection of a basis of $\overline{\text{post}^*(M_0)}$ with $J_{\mathbf{e}}$ where $M_0 = \langle 0,1,1,1,q_0\rangle$. The first coordinate corresponds to *P*/*product*, the second coordinate to *W*/*warehouse*, and the third and fourth coordinate correspond to S_1/$store_1$ and S_2/$store_2$, respectively.

$$B_{\mathbf{e}}^{M_0} = \{\langle 0,1,1,1\rangle, \langle 0,5,0,0\rangle, \langle 0,0,3,0\rangle, \langle 0,0,0,3\rangle, \langle 0,1,2,0\rangle, \langle 0,1,0,2\rangle, \langle 0,0,2,1\rangle, \langle 0,0,1,2\rangle, \langle 0,3,1,0\rangle, \langle 0,3,0,1\rangle\} \times \{q_0\}$$

We computed B^k, a basis of I^k, for $1 \leq k \leq 21$. For the sake of comprehensibility, we only give $B^k \cap J_{\mathbf{e}}$ for $1 \leq k \leq 6$:

$$B^1 \cap J_{\mathbf{e}} = \{\langle 0,1,1,1\rangle, \langle 0,2,0,1\rangle, \langle 0,2,1,0\rangle\} \times \{q_0\}$$
$$B^2 \cap J_{\mathbf{e}} = \{\langle 0,0,1,1\rangle, \langle 0,2,0,1\rangle, \langle 0,2,1,0\rangle, \langle 0,3,0,0\rangle\} \times \{q_0\}$$
$$B^3 \cap J_{\mathbf{e}} = \{\langle 0,0,1,1\rangle, \langle 0,1,0,1\rangle, \langle 0,1,1,0\rangle, \langle 0,3,0,0\rangle\} \times \{q_0\}$$
$$B^4 \cap J_{\mathbf{e}} = B^5 \cap J_{\mathbf{e}} = B^3 \cap J_{\mathbf{e}}$$
$$B^6 \cap J_{\mathbf{e}} = \{\langle 0,0,1,1\rangle, \langle 0,1,0,1\rangle, \langle 0,1,1,0\rangle, \langle 0,3,0,0\rangle, \langle 0,0,2,0\rangle, \langle 0,0,0,2\rangle\} \times \{q_0\}$$

We obtain $B_{\mathbf{e}}^{M_0} \not\subseteq \overline{B^k \cap J_{\mathbf{e}}}$ for $1 \leq k \leq 5$, but $B_{\mathbf{e}}^{M_0} \subseteq \overline{B^6 \cap J_{\mathbf{e}}}$. Thus, $k_{\min} = 6$. We also rediscover that we can reach I from any state in $J_{\mathbf{e}}$ in at most 17 steps since

$$B^{17} \cap J_{\mathbf{e}} = \{\langle 0,0,0,0,q_0\rangle\}.$$

We also computed the indices $k(I) = 20$ and $k(J_{\mathbf{e}}) = 3$.

Engineering Grammar-based Type Checking for Graph Rewriting Languages

Naoki Yamamoto Kazunori Ueda
Dept. of Computer Science and Engineering
Waseda University, Tokyo, Japan
{yamamoto,ueda}@ueda.info.waseda.ac.jp

The ability to handle evolving graph structures is important both for programming languages and modeling languages. Of various languages that adopt graphs as primary data structures, a graph rewriting language LMNtal provides features of both (concurrent) programming languages and modeling languages, and its implementation unifies ordinary program execution and model checking functionalities. Unlike pointer manipulation in imperative languages, LMNtal allows us to manipulate graph structures in such a way that the well-formedness of graphs is an invariant guaranteed by the language itself. However, since the shapes of graphs can be complex and diverse compared to algebraic data structures such as lists and trees, it is a non-obvious important task to formulate types of graphs to verify individual programs. With this motivation, this paper discusses LMNtal ShapeType, a type checking framework that applies the basic idea of Structured Gamma to a concrete graph rewriting language. Types are defined as generative grammars written as LMNtal rules, and type checking of LMNtal programs can be done by exploiting the model checking features of LMNtal itself. We gave a full implementation of type checking using the features of the LMNtal meta-interpreter.

1 Introduction

The ability to handle dynamically evolving graph structures is important both for programming languages and modeling languages. In programming, graphs appear both as *data structures* supporting efficient algorithms and as *process structures* exchanging messages through channels. In modeling, network structures can be found in many fields from the Internet to transportation, which can be modeled as graphs. Of various languages that adopt graphs as primary data structures, including GP 2 [1] and GROOVE [7], a graph rewriting language LMNtal [16] provides features of *programming* languages (including I/O and various other APIs) and those of *modeling* languages (including state space search). Its implementation, SLIM [8] (available from GitHub), provides ordinary program execution and parallel model checking (with 10^9 states) in a single framework. LMNtal allows us to handle data structures that cannot be succinctly modeled in functional languages. An example is a skip list [10] in Fig. 1(a), a linked list with additional edges skipping some nodes, which can be encoded into an LMNtal graph as in Fig. 1(b). Although LMNtal has simple syntax and semantics consisting of atoms, links and rewrite rules, it is Turing-complete and allow the encoding of various process calculi and the strong reduction of the λ-expressions in which both term structures and bound variables are represented as graphs [15].

Although LMNtal programs are pointer-safe in the sense that phenomena corresponding to dangling pointers and unintended aliasing in imperative languages never happen, it is possible that a graph with an unexpected shape is generated as a result of rewriting or computation gets stuck. For instance, see the two rewrite rules in Fig. 2. While the correct rule preserves the structure of a skip list, the incorrect one destroys the structure. Appropriate static type checking would detect such errors at compile time.

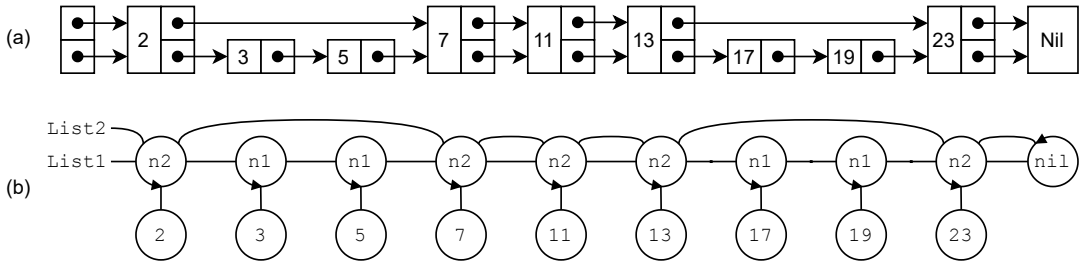


Figure 1: An example of skip list: (a) with pointers, (b) as an LMNtal graph. An arrowhead of each non-unary atom indicates the first argument and the ordering of arguments.

Figure 2: Example rules for skip list.

LMNtal ShapeType [19] is a type checking framework for LMNtal inspired by Shape types [5] which in turn is based on Structured Gamma [6]. Types are defined as generative grammars represented by rewrite rules of LMNtal. This makes it possible to do type checking by the non-deterministic execution of SLIM. Positioning a type description language as a sublanguage of the host language and making full use of the functionalities of the latter's implementation is a major strength of the present approach.

LMNtal ShapeType provides two basic type checking algorithms. One is *graph type checking* to check that a given graph is of the specified type. The other is *rule type checking* to check that a given rule will not destroy the structure of typed graphs.

The main contributions of this paper are threefold. First, we formalize LMNtal ShapeType addressing various subtleties. In previous studies, rule type checking had no explicit algorithm or correctness proof. Second, we expand the expressive power of the type checking. In order to handle constraints such as the balancing of trees and the number of elements an "express" link of a skip list can skip, we propose *extensive types* and *indexed types*[1] as new classes of graph types which are broader than context-free grammars. Finally, we propose a unified approach to check the type safety of *functional atoms*, i.e., atoms corresponding to functions (as opposed to data constructors) in other languages. In existing methods, type safety meant that each rewriting step would not destroy the structure of typed graphs. However, we often need to perform multi-step operations which may result in graphs of different types. We introduce a design pattern called functional atoms to check the type safety of such operations.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 introduces Flat LMNtal, the base language of the present work. Section 4 introduces LMNtal ShapeType and describes its type checking algorithms. Section 5 describes notable properties of LMNtal ShapeType. Section 6 introduces functional atoms and shows how to check their type safety. Section 7 discusses implementation.

[1] A variety of graph type definitions including the ones introduced in this paper can be found in `http://bit.ly/LMNtalShapeTypeEx`.

2 Related Work

There are several typing frameworks for graphs. Graph types [9] are a framework based on regular expressions. Structured Gamma [6] is a framework for graphs in which types are defined by production rules in context-free grammar. Shape types [5] are a subset of Structured Gamma and handle types which make the type checking algorithm complete. These methods ensure that graph structures expressed using pointers in C-like languages are consistent with the type definitions and that one-step operations on the graph structures will not affect the types of the structure. An algorithm proposed in [2] is able to handle shape-changing computation by specifying intermediate shapes, whereas our approach achieves the same objective using functional atoms. Besides, these methods are based on networks of pointers, and express graph edges by names (such as '`next`') and graph nodes by variables. This style is dual of our approach in which edges are expressed by α-convertible variables and nodes are expressed by atom names (such as '`cons`'). Although various formalisms of graph grammars [13] are well studied for decades, our technique, formulated in the framework of a practical concrete language, differs from those in many respects including the formulation of graphs and rewriting.

For our base language LMNtal, the method of [17] deals with 'microscopic' properties by giving capability types, which represent both polarities and sharing of (hyper)links, to local connection between nodes, while LMNtal ShapeType handles 'macroscopic' graph structures, i.e., shapes.

Separation logic [11] is well studied for reasoning about pointer structures, but the approach is different from ours in several respects: it deals with low-level languages and properties, while we consider graph structures formed by higher-level languages, abstracting pointers and heaps. Reasoning with separation logic uses proof assistants except for certain properties, while our objective is to pursue what properties can be established automatically using rather simple typing framework.

There are a lot of studies on handling quantitative properties in type systems [3][12]. Most of them enhance type systems with dependent types and employ decision procedures such as constraint solvers. While we pursue a somewhat close approach (Section 5.4), we also pursue an approach that does not employ numerical types for broader applications in mind (Section 5.1).

3 LMNtal: Graph Rewriting Language

In order to focus on the shape properties of graph rewriting in a concrete setting but without unnecessary complication, we consider a subset of LMNtal, called Flat LMNtal, which omits another structuring mechanism called *membranes*. This results in a significantly simpler fragment compared to the original setting [16]. We do not handle guards (for operations on built-in data types) or hyperlinks (for multi-point connectivity) either, but this core language still provides a powerful structuring mechanism. Hereinafter we simply call this subset LMNtal.

3.1 Syntax

The syntax of LMNtal is shown in Fig. 3. An LMNtal program is represented as a pair of a *graph* G (a multiset of atoms) and a *ruleset* R (a multiset of rewrite rules). This pair is called a *process*. An atom consists of an m-ary atom name p followed by m *totally ordered* link names $X_1, \ldots, X_m$. Names starting with capital letters are interpreted as link names, and others as atom names. The pair of the name and the arity of an atom are referred to as the *functor* of the atom and written as p/m. Atoms and links correspond to nodes and edges in graph theory, respectively. Unlike many other graph rewriting formalisms, graphs of LMNtal are defined in a syntax-directed manner and each atom has its own arity.

Process		::=	G, R	
Graph	G	::=	**0**	(null)
		|	$p(X_1, \ldots, X_m)$	(atom, $m \geq 0$)
		|	G, G	(molecule)
Ruleset	R	::=	**0**	(null)
		|	[*RuleName* `@@`] G `:-` G	(rule)
		|	R, R	(molecule)

Figure 3: Syntax of LMNtal.

In LMNtal, a multiset of atoms stands for an undirected multigraph, i.e., a graph that allows multi-edges and self-loops. For instance, a multiset of atoms `a(L1,F),b(L1,L2,L3,L4),c(L2,L5,L6,L6),d(L5,L3,L4)` stands for the undirected graph shown in Fig. 4.

A link name occurs at most twice in a graph. Link names occurring twice in a graph are called *local links* and those occurring once are called *free links* (a.k.a. half-edges) one of whose ends remains unconnected (or connected to the outside of the graph). Graphs without free links are *closed*. When we compose two graphs with a comma, we regard them to be α-converted as necessary to avoid collisions among local link names.

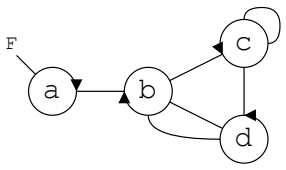


Figure 4: Pictorial representation of an LMNtal graph: `a(L1,F),b(L1,L2,L3,L4),c(L2,L5,L6,L6),d(L5,L3,L4)`

The following two abbreviations are allowed.

1. An atom written as another atom's argument is regarded as connected to the last argument of the outer atom, that is, $p(X_1, \ldots, X_{k-1}, q(Y_1, \ldots, Y_n), X_{k+1}, \ldots, X_m)$ $(1 \leq k \leq m, 1 \leq n)$ is interpreted as $p(X_1, \ldots, X_{k-1}, L, X_{k+1}, \ldots, X_m), q(Y_1, \ldots, Y_n, L)$ where L is a fresh link name. For instance, `a(b(c),d)` means `a(B,D),b(C,B),c(C),d(D)`.
2. An atom p`()` with no arguments may be written as p.

A *rule* (with an optional *RuleName*) describes rewriting of a subgraph to a subgraph. For instance, a rule `to(X,Y) :- from(Y,X)` rewrites a binary atom '`to`' to a binary atom '`from`' with its arguments swapped. For readability, rules may be written in a period-terminated form as well as in a comma-separated form. Because free links must not appear or disappear by rewriting, a link name in a rule must occur exactly twice.

3.2 Semantics

The semantics of LMNtal consists of *structural congruence* and *reduction relation*. We will introduce them in detail.

3.2.1 Structural Congruence

The syntax defined above does not (yet) characterize LMNtal graphs because the figure depicted in Fig. 4 corresponds to other syntactic representations of LMNtal graphs as well. We need to define an equivalence relation "$\equiv$", called *structural congruence*, to absorb syntactic variations. Structural

$$
\begin{array}{lrcll}
\text{(E1)} & \mathbf{0}, P & \equiv & P & \\
\text{(E2)} & P, Q & \equiv & Q, P & \\
\text{(E3)} & P, (Q, R) & \equiv & (P, Q), R & \\
\text{(E4)} & P & \equiv & P[Y/X] & \text{(if } X \text{ is a local link of } P\text{)} \\
\text{(E5)} & P \equiv P' & \Rightarrow & P, Q \equiv P', Q & \\
\text{(E7)} & X{=}X & \equiv & \mathbf{0} & \\
\text{(E8)} & X{=}Y & \equiv & Y{=}X & \\
\text{(E9)} & X{=}Y, P & \equiv & P[Y/X] & \text{(if } P \text{ is an atom and } X \text{ is a free link of } P\text{)}
\end{array}
$$

Figure 5: Structural congruence on LMNtal graphs.

$$
\text{(R1)}\ \frac{G_1 \xrightarrow{T :- U} G_1'}{G_1, G_2 \xrightarrow{T :- U} G_1', G_2} \qquad \text{(R3)}\ \frac{G_2 \equiv G_1 \quad G_1 \xrightarrow{T :- U} G_1' \quad G_1' \equiv G_2'}{G_2 \xrightarrow{T :- U} G_2'} \qquad \text{(R6)}\ T \xrightarrow{T :- U} U
$$

Figure 6: Reduction relation on LMNtal graphs.

congruence is defined as the minimum equivalence relation satisfying the rules in Fig. 5. Structurally congruent LMNtal graphs are considered indistinguishable from each other. $P[Y/X]$ in (E4) and (E9) means to replace the link name X occurring in the graph P with the link name Y. Note that (E6) and (E10) in the original definition [16] are omitted because these are rules for membranes.

(E1)–(E3) characterize graph nodes as multisets. (E5) is a structural rule to make $\equiv$ a congruence. Both of them are standard rules found also in process algebra. (E4) is α-conversion of local link names[2]. Rules (E7) to (E9) are about the special binary atom = called a *connector*. An atom $=(X, Y)$, also written as $X{=}Y$, fuses two links X and Y. (E7) says that a self-closed link is regarded as a null graph, (E8) says that a connector is symmetric, and (E9) says that a connector may be absorbed or emitted by an atom. Connectors play an important role in writing rewrite rules such as

```
append(X,Y,Z), nil(X) :- Y=Z.
```

They play an important role in LMNtal ShapeType also.

3.2.2 Reduction Relation

"$\xrightarrow{T :- U}$", called a *reduction relation by the rule* $T :- U$, is a binary relation between two graphs, which describes the principal computation step in LMNtal. It is defined as the minimum binary relation satisfying the rules in Fig. 6. Note that (R2), (R4) and (R5) in [16] are omitted because these are for membranes.

The most important rule is (R6) which states that if there is a subgraph that matches the LHS of a rule, the subgraph can be rewritten into the RHS[3]. This definition can be naturally extended for rulesets, i.e., we say "G can transition (in one step) to G' by the ruleset R," written $G \xrightarrow{R} G'$, if $\exists r \in R.\ G \xrightarrow{r} G'$.

[2]The new link name Y must be "fresh" here; otherwise the graph $P[Y/X]$ violates the prerequisite that each link name can occur at most twice.

[3]For simplicity, we intentionally allow the case where T is null, which readily introduces divergence, though a legitimate implementation need not compile such rules.

In order to facilitate the formulation of our type checking method, the syntax and semantics described above separate graphs and rulesets, while in the standard definition they conjunctively form a process that may evolve autonomously.

3.3 Reverse Execution in LMNtal

Before we move on to the definition of LMNtal ShapeType, we introduce reverse execution in LMNtal. First we define the *inversion* of rules and rulesets.

Definition 1. The *inversion* r^{inv} of an LMNtal rule $r = T :- U$ is defined by $r^{\mathrm{inv}} = U :- T$. Likewise, the inversion R^{inv} of an LMNtal ruleset R is defined by $\forall r.\ r \in R \Leftrightarrow r^{\mathrm{inv}} \in R^{\mathrm{inv}}$.

It is important to note that an inverted rule is also a well-formed rule in LMNtal, and this plays a key role in our type checking method. Actually, reduction using an inverted rule is equivalent to following the reduction relation of the original rule in an opposite direction[4].

Proposition 1. For an LMNtal rule $r = T :- U$, $G' \xrightarrow{r} G \Leftrightarrow G \xrightarrow{r^{\mathrm{inv}}} G'$.

Hence the reduction relation can be followed backward by executing the inversion of the LMNtal rule. This is called *reverse execution*.

4 LMNtal ShapeType

A typed language comes with a type description (sub)language, which is called LMNtal ShapeType in our setting. This section defines LMNtal ShapeType and describes its basic type checking algorithms. First, we give a formal definition. Following the terminology of formal language theory, LMNtal functors are hereinafter referred to as *symbols* also.

Definition 2. A *type* in LMNtal ShapeType (simply called *ShapeType*) is a triplet (S, P, N), where

- $S = t/m$ is a functor called the *start symbol*,
- P is a finite set of rules called *production rules*, and
- N is a finite set of functors called *nonterminal symbols*.

4.1 Syntax

The triplet of Definition 2 is written by the syntax of Fig. 7, where the start symbol S is given as an atom $p(X_1, \ldots, X_m)$[5], the production rules P as a ruleset R in Fig. 3, and nonterminal symbols N as a graph G in Fig. 3. *The LHS of each production rule must consist only of one or more nonterminal atoms which must not include connectors*. Abbreviations allowed for LMNtal atoms (Section 3.1) are also allowed.

ShapeType ::= `defshape` S `{` P `}` [`nonterminal {` N `}`]

Figure 7: Syntax of ShapeType.

[4] Proofs of all theorems, propositions and lemmas can be found in Appendix.

[5] We allow the case where $m = 0$, i.e., graphs with no roots, and non-connected graphs, for which our algorithms described in this section work as well.

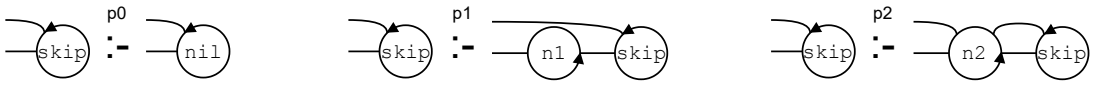


Figure 8: Production rules of the skip-list type `skip`.

By abuse of notation, functors in S and N are written as atoms in which link names are insignificant. Note also that S is always considered to be included in the set N of nonterminals (including the case where the definition of N is omitted).

An important feature of LMNtal ShapeType is that *a graph to be typed may have two or more roots (as in skip lists) whose order is significant*. Accordingly, when discussing whether a graph is of a specific type, a type is referred to as $t(L_1, \ldots, L_m)$ by explicitly mentioning roots. If the link names are not important, it is called type t/m (or type t if t is a unique start symbol name).

4.2 Semantics

We define the typing relation ":" using an auxiliary relation "$\lhd$" called the production relation. Hereinafter, a graph with free links $L_1, \ldots, L_m$ is written as $G[L_1, \ldots, L_m]$ and the set of all functors used in graph G is written as $\mathrm{Funct}(G)$.

Definition 3 (Production Relation)**.** For a type $(t/m, P, N)$, a graph $G[L_1, \ldots, L_m]$ is *generated by the type* $t(L_1, \ldots, L_m)$, written $G[L_1, \ldots, L_m] \lhd t(L_1, \ldots, L_m)$, iff $t(L_1, \ldots, L_m) \xrightarrow{P}{}^{*} G[L_1, \ldots, L_m]$.

Definition 4 (Typing Relation)**.** For a type $(t/m, P, N)$, a graph $G[L_1, \ldots, L_m]$ *has the type* $t(L_1, \ldots, L_m)$, written $G[L_1, \ldots, L_m] : t(L_1, \ldots, L_m)$, iff $G[L_1, \ldots, L_m] \lhd t(L_1, \ldots, L_m) \wedge \mathrm{Funct}(G[L_1, \ldots, L_m]) \cap N = \varnothing$.

Intuitively, only graphs to which the start symbol of type t can transition in zero or more steps by the production rules are said to be generated by type t, and only graphs with no nonterminal symbols are said to have the type t. In these definitions, free links of graphs and types are both explicitly written because their names and ordering are significant. For instance, given a type

```
defshape t(X,Y) { t(X,Y) :- a(X,Y) },
```

`a(X,Y)` $\lhd$ `t(X,Y)` holds but `a(Y,X)` $\lhd$ `t(X,Y)` does not hold.

The type of skip lists shown in Section 1 can be described as follows.

```
defshape skip(List2,List1){
  p0@@ skip(L2,L1) :- nil(L2,L1).
  p1@@ skip(L2,L1) :- n1(X1,L1), skip(L2,X1).
  p2@@ skip(L2,L1) :- n2(X1,X2,L2,L1), skip(X2,X1).
}
```

Note that elements in the skip list are omitted in order to focus on the structure of the graph, though it is possible to include elements and specify their types. The production rules can be visualized as in Fig. 8.

4.3 Graph Type Checking

Graph type checking is to check if an LMNtal graph X has a type t. The algorithm is shown in Fig. 9.

GCHECK is to check if the graph X has the type $(t/m, P, N)$. To check that X does not include a nonterminal symbol, we only have to check all the atoms because the number of atoms in X is finite. Then, it suffices to check that X is generated by $(t/m, P, N)$. GGCHECK checks that X can transition

```
    // Checks that the LMNtal graph X has the type (t/m,P,N)
1:  function GCHECK(X,(t/m,P,N))
2:      if ∃f ∈ Funct(X). f ∈ N then return false
3:      else return GGCHECK(X,(t/m,P,N),∅)
4:  end function

    // Checks that the LMNtal graph X can be generated by the type (t/m,P,N)
1:  function GGCHECK(X,(t/m,P,N),G)
2:      if X ≡ t(L1 , ... ,Lm) then return true
3:      for each Y s.t. X --P^inv--> Y ∧ ∄Y' ∈ G. Y ≡ Y' do
4:          G ← G ∪ {Y}
5:          if GGCHECK(Y,(t/m,P,N),G ∪ {X}) then return true
6:      end for
7:      return false
8:  end function
```

Figure 9: Algorithm of graph type checking.

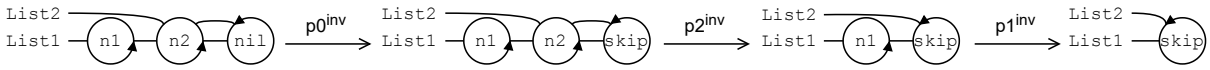


Figure 10: Reverse execution path of a skip list.

back to the start symbol by reverse execution with production rules P. For example, by reverse execution shown in Fig. 10, we can verify that the leftmost graph in Fig. 10 has the `skip` type. For a certain class of types that will be detailed in Section 5, the graph type checking algorithm satisfies soundness, completeness and termination for any rule.

4.4 Rule Type Checking

First, we define the type preservation property of an LMNtal rule r for the type t.

Definition 5 (Type Preservation)**.** Let r be an LMNtal rule, t be a type, and $L_1,\ldots,L_m$ be a sequence of links. We say that *r preserves t* iff

$$\forall G : t(L_1 , \ldots , L_m).\quad G \xrightarrow{r} G' \Rightarrow G' : t(L_1 , \ldots , L_m).$$

Checking the type preservation property is called *rule type checking*. The algorithm is shown in Fig. 11.

Intuitively, the algorithm checks if each generation path of L can be transformed to a generation path of R by structural induction on the production rules used last. RCHECK ensures that both sides of the given rule consist only of the terminal symbols and then calls RCHECKSUB. RCHECKSUB recursively follows the production rule of t backwards from the LHS L, supplying a 'deficient' graph C to both sides (line 8), until the LHS reaches the start symbol (line 2) or a graph that appeared before (line 5). If it detects an LHS that appeared before, the algorithm backtracks to the point where the LHS appeared first. Positive return values are used to inform how many times the function should return by backtracking. Then REDUCE verifies that the resulting graph augmented with the supplied graphs accumulated in the previous phase can transition to the RHS. REDUCE shows this by reverse execution

```
// Checks that the rule L :- R preserves the type (t/m,P,N)
1: function RCHECK(L :- R, (t/m,P,N))
2:     return (Funct(L) ∪ Funct(R)) ∩ N = ∅ ∧ RCHECKSUB(L :- R, (t/m,P,N), []) = 0
3: end function

// Traverses the state space generated by reverse execution
// (L :- R): current state, [v1,...,vn]: stack of visited states in depth first search
1: function RCHECKSUB(L :- R, (t/m,P,N), [v1,...,vn])
2:     if L consists only of one t/m atom then
3:         if REDUCE(R,L,P,∅) then return 0          ▷ Returns 0 if the rule preserves the type
4:         else return −1                            ▷ Returns −1 if the rule may destroy the type
5:     if ∃i s.t. 1 ≤ i ≤ n ∧ v_i = L_i :- R_i ∧ L_i ≡ L then
6:         if REDUCE(R_i,L_i,P,∅) then return n − i  ▷ Backtracks n − i steps when a cycle is detected
7:         else return −1
8:     for each (L',C) s.t. L,C --P^inv--> L' do     ▷ Explore all paths
9:         S ← RCHECKSUB(L' :- (R,C), (t/m,P,N), [v1,...,vn,L :- R])
10:        if S > 0 then return S − 1                ▷ Continues backtracking
11:        if S = −1 then return −1                  ▷ The rule may destroy the type if one or more calls return −1
12:    return 0
13: end function

// Checks that there exists a reverse execution path from X to Y, i.e., X --P^inv-->* Y
// G: visited graphs
1: function REDUCE(X,Y,P,G)
2:     if X ≡ Y then return true
3:     for each X' s.t. X --P^inv--> X' ∧ ∄X'' ∈ G. X'' ≡ X' do
4:         G ← G ∪ {X'}
5:         if REDUCE(X',Y,P,G ∪ {X}) then return true
6:     return false
7: end function
```

Figure 11: Algorithm of rule type checking.

in order to prevent divergence. Finally, RCHECKSUB returns 0 if there is a state $L :- R$ on every path such that $\textsc{Reduce}(R,L,P,\varnothing)$ returns **true** and returns -1 otherwise. Note that each graph C enumerated in line 8 of RCHECK must be the minimum one that enables matching with the RHS of rules and lets the reverse execution proceed.

For example, to verify that the rule in Fig. 12 preserves the `skip` type, we first go back from the LHS, supplying and accumulating necessary graphs (Fig. 13, upper), and then check if the resulting graph (`skip` in this example) can transition to the RHS with the supplied graphs (Fig. 13, lower).

This algorithm is inspired by that of Structured Gamma [6]. While the algorithm of Structured Gamma generates the entire state space first and the correspondence of free links in the supplied graphs was recorded and managed separately, our algorithm avoids this inconvenience by rewriting the target rule itself to generate the state space, using rewrite rules to represent individual states. This idea is similar to that of the sequent calculus in that it treats the pair of premise and conclusion as a single object.

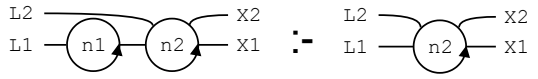


Figure 12: An example rule preserving the `skip` type.

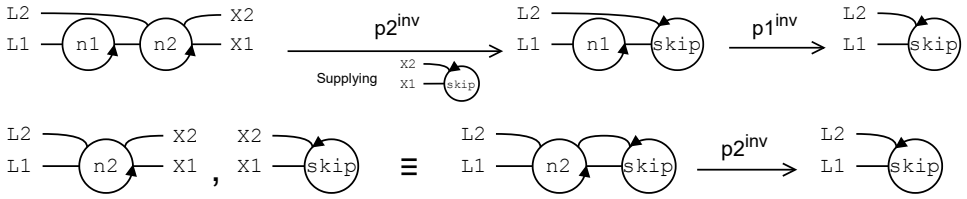


Figure 13: Reverse execution from the LHS (upper) and from the RHS (lower) of Fig. 12.

5 Properties of LMNtal ShapeType

This section describes some important properties of LMNtal ShapeType.

5.1 Extensive Types

The type checking algorithms introduced in [5] and [6] handle types with context-free production rules to ensure termination of graph type checking. *Context-free* types in our setting are defined as follows:

Definition 6. A production rule $\alpha :- \beta$ is *context-free* iff α consists of a single atom and β contains one or more atoms other than connectors. A type is context-free iff all of its production rules are context-free.

However, to ensure termination of graph type checking, it is sufficient if there is some measure for graphs whose values will not decrease by production rules, and this extension opens up versatile applications. Thus we propose *graph weighting* as a new measure for graphs.

Definition 7. For a type τ, a *weighting function* $w : \mathrm{Funct}(\tau) \to \mathbb{N} \setminus \{0\}$ weights each functor occurring in τ with a positive integer. As an exception, connectors' weight $w(=/2)$ must be zero because connectors can be arbitrarily absorbed or emitted by the structural congruence rule (E9).

Definition 8. Let w be a weighting function for a type τ and G be a graph which consists only of atoms with functors contained in $\mathrm{Funct}(\tau)$. The *weight of the graph* G, denoted $w(G)$, is defined by

$$w(G) = \sum_{p(L_1,\ldots,L_m)\in G} w(p/m).$$

Note that graph weighting generalizes the concept of the number of atoms.

Definition 9. A type $(t/m, P, N)$ is *extensive* iff $\exists w.\ \forall(\alpha :- \beta) \in P.\ w(\alpha) \leq w(\beta)$.

Context-free types are obviously extensive. An extensive type corresponds to a length-increasing grammar in formal language theory, which is equivalent to a context-sensitive grammar. This extension is motivated by the need to handle a variety of types with non-context-free constraints as explained below[6].

For instance, the type of red-black trees can be defined as in Fig. 14. Here, the nonterminal symbol

[6]However, since LMNtal ShapeType handles graphs, care must be taken when discussing its expressive power. For instance, the list version of the typical context-sensitive language $\{a^n b^n c^n\}$ can be expressed by a simple context-free type in our setting.

```
defshape rbtree(R){
   s@@ rbtree(R) :- btree(nat,R).
  bz@@ btree(z,R) :- leaf(R).
  tz@@  tree(z,R) :- leaf(R).
  bs@@ btree(s(N),R) :- b(tree(N1),tree(N2),R), cp(N,N1,N2).
  ts@@  tree(s(N),R) :- b(tree(N1),tree(N2),R), cp(N,N1,N2).
   r@@  tree(N,R) :- r(btree(N1),btree(N2),R), cp(N,N1,N2).
  ns@@ nat(R) :- s(nat,R).
  nz@@ nat(R) :- z(R).
  cs@@ cp(s(N),N1,N2) :- cp(N,M1,M2), s(M1,N1), s(M2,N2).
  cz@@ cp(z,N1,N2) :- z(N1), z(N2).
} nonterminal {
  rbtree(R), btree(N,R), tree(N,R),
  cp(N,N1,N2), nat(R), s(N,R), z(R)
}
```

Figure 14: Type definition of red-black trees

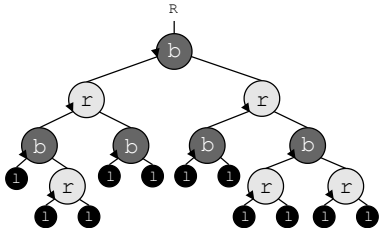


Figure 15: An example graph of the `rbtree` type (`l` stands for `leaf`).

`btree`/2 stands for a red-black tree with a black root and `tree`/2 stands for a red-black tree with a black or red root. In this definition, the black height of a tree to be generated is expressed using `z`/1 (zero) and `s`/2 (successor) atoms as X=`s(s(`$\ldots$`s(z)`$\ldots$`))` (as done in Prolog). The atom `cp`/3 copies and distributes the numeral connected to the first argument to the second and third arguments, with the production rules `cs` and `cz`. This definition expresses the balance of red-black trees by constraints on black heights distributed properly to subtrees. For instance, a graph in Fig. 15 has the `rbtree` type. This `rbtree` type is extensive with a weighting function $w(x)$ which returns 2 if $x =$ `leaf`$/2$ and returns 1 otherwise. Note that it is easy to find an appropriate weighting since the constraint of extensive types (Def. 9) reduces to a system of linear inequalities.

The technique that employs `z`/1 and `s`/2 atoms can be used also for defining, e.g., skip lists with constraints on the number of stops that an "express" link can skip. Also, the type of the graph representation of λ-terms needs to manage a list (of unbounded length) of free links of subterms, which can be represented using an additional nonterminal symbol representing a list constructor.

For extensive types, the algorithm of graph type checking (Fig. 9) satisfies soundness, completeness, and termination.

Theorem 1 (Termination of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType τ, GC$\textsc{heck}(X,\tau)$ terminates.

Theorem 2 (Soundness of graph type checking)**.** For any LMNtal graph X and a ShapeType $(t/m,P,N)$, if GC$\textsc{heck}(X,(t/m,P,N))$ returns **true**, $X : t(L_1, \ldots, L_m)$ holds.

Theorem 3 (Completeness of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType $(t/m,P,N)$, if $X : t(L_1, \ldots, L_m)$ holds, GC$\textsc{heck}(X,(t/m,P,N))$ returns **true**.

5.2 Production rules with connectors

The data structure called *difference lists* (*d-lists* for short), commonly used in Prolog programming since 1970's, represents a list with two variables representing the two ends. It enables constant-time concatenation by virtue of logic variables. Difference lists are extremely useful also in LMNtal programming where links are a special use of logic variables.

A type of d-lists can be written as in Fig. 16 (left). Since `p1` (for an empty d-list) has no atoms in the RHS except for a connector, there is no increasing weighting function for this type. The inversion of `p1`

```
defshape dlist(X,Y){
  p1@@ dlist(X,Y) :- X=Y.
  p2@@ dlist(X,Y) :- c(dlist(X),Y).
}
```

```
defshape dlist(X,Y){
  p1@@ dlist(X,Y) :- X=Y.
  p2@@ dlist(X,Y) :- c(dlist(X),Y).
  p3@@ dlist(X,Y) :- c(X,Y).
}
```

Figure 16: Type definition of d-lists (left) and its normalized definition (right).

is `X=Y :- dlist(X,Y)`, which matches *any* link, so reverse execution with this rule will not terminate. However, we cannot describe an *empty* d-list without such rules.

Let the target graph to be type-checked be G. If two free links of G are connected to each other, G cannot be expressed without connectors. A connector connecting two free links of G is called *global*. Now, we put a restriction that the inversion of a production rule fires only if all connectors in the RHS of the original production rule are global in G. With this restriction, a connector in the RHS of a production rule cannot match an arbitrary link.

Although this seems to decrease the flexibility of the types, the type description with this restriction retains the same expressiveness as the case where connectors can be freely used in the RHS of production rules. Actually, non-global connectors can be removed by a procedure like *the elimination of ε-rules* in the conversion of a context-free grammar to Chomsky normal form. Figure 16 (right) shows the result of such normalization.

Although we defined that connectors must have zero weight in Definition 7, global connectors may be weighted in the same way as ordinary functors. This is because only finitely many global connectors can appear in a graph and they cannot be absorbed or emitted. Note that a global connector cannot be a nonterminal symbol.

We go back to the `dlist` example. When all functors including the global connector are weighted 1, the production rules will not decrease the weight. Furthermore, graph type checking ensures that an empty d-list has the type `dlist` since `X=Y` can make reverse transition to `dlist(X,Y)` by the rule `p1`.

5.3 Rule type checking

The algorithm of rule type checking (Fig. 11) is sound in the following sense:

Theorem 4. For an LMNtal rule α :- β, a ShapeType $(t/m,P,N)$, and a sequence of links $L_1,\ldots,L_m$, if RCHECK$(\alpha$:- $\beta,(t/m,P,N))$ returns **true**, the following formula (the type preservation property) holds:

$$\forall G : t(L_1,\ldots,L_m).\quad G \xrightarrow{\alpha :- \beta} G' \;\Rightarrow\; G' : t(L_1,\ldots,L_m)$$

For context-free types, the rule type checking algorithm terminates for any rule because the state space of our algorithm is the same as that of Structured Gamma [6] if we focus only on the LHS. However, it may not terminate for extensive types. For example, we can use the production rule `cz` of `rbtree` type (Fig. 14) backwards to transition from one `z` atom to `cp` and `z` atoms. This causes infinitely many `cp` atoms to be generated from a single `z` atom, so the rule type checking does not terminate.

As with other static type checking methods, completeness of rule type checking may not hold in general [6]. The proof of soundness assumes that there exists a transition path from the start symbol to a resulting graph of reverse execution from the LHS[7], but if there is no such path, the completeness does not hold. In this sense the algorithm is conservative.

[7] which is α_0 in the proof in Appendix.

```
defshape rbtree3(R){
  init@@ rbtree3(R) :- btree3(R).
  bb3@@ btree3(R) :- b(tree2,tree2,R).
  bb2@@ btree2(R) :- b(tree1,tree1,R).
  bb1@@ btree1(R) :- b(tree0,tree0,R).
  bl@@ btree0(R) :- leaf(R).
  tb2@@ tree2(R) :- b(tree1,tree1,R).
  tb1@@ tree1(R) :- b(tree0,tree0,R).
  tr2@@ tree2(R) :- r(btree2,btree2,R).
  tr1@@ tree1(R) :- r(btree1,btree1,R).
  tr0@@ tree0(R) :- r(btree0,btree0,R).
  tl@@ tree0(R) :- leaf(R).
} nonterminal {
  rbtree3(R), btree3(R), btree2(R),
  btree1(R), btree0(R),
  tree2(R), tree1(R), tree0(R)
}
```

```
defshape rbtree(R){
  init@@ rbtree(R) :- btree($n,R).
  bb@@ btree($n,R) :- $m = $n-1
    | b(tree($m),tree($m),R).
  bl@@ btree(0,R) :- leaf(R).
  tb@@ tree($n,R) :- $m = $n-1
    | b(tree($m),tree($m),R).
  tr@@ tree($n,R)
    :- r(btree($n),btree($n),R).
  tl@@ tree(0,R) :- leaf(R).
} nonterminal {
  rbtree(R), btree($n,R), tree($n,R)
}
```

Figure 17: Type definition of red-black trees as a context-free type (left) and as an indexed type (right).

5.4 Indexed Types

Types with numerical constraints, such as complete binary trees or (balanced) red-black trees of height n, or lists of length n, cannot be represented as context-free types. We could use extensive types (Section 5.1), for which the graph type checking works well, but the rule type checking for such types may not terminate.

On the other hand, types with constant numerical constraints, such as red-black trees with a height of exactly 3, can be represented by a context-free type as in Fig. 17 (left). The meanings of symbols in this definition are the same as those in Fig. 14 except that the names of nonterminal symbols are followed by indices representing the black height. In this definition, the production rules `bb3`–`bb1`, `tb2`–`tb1`, `tr2`–`tr0` are quite similar, respectively, so we can simplify them as in Fig. 17 (right), where we introduced a notation like '`$n`' to represent integer variables. This notation is borrowed from the *typed process context* [16], an extension of LMNtal. While the original typed process context is a mechanism to match any graph that satisfies the constraints specified in the *guard* (the part between `:-` and `|`), here we assume that all typed process contexts match only natural numbers[8]. This extension allows natural numbers and typed process contexts to appear in the LHS of the production rule in addition to one nonterminal symbol. Although this goes beyond the context-freeness assumption, the production rules are still essentially context-free in the sense that this natural number, which originally played the role of an index of a non-terminal symbol, can be regarded as an index of the non-terminal symbol. This idea of indexed nonterminal symbols was derived from indexed grammars in formal language theory.

Since these process contexts represent variables that may take any natural numbers, the rule type checking can cause state space explosion. Therefore, we ignore the difference of the indices of nonterminal symbols and consider them as the same state in the state space construction.

In this setting, the algorithm terminates because the state space of an indexed type is isomorphic to that of the type without indices. Since multiple states are represented by a single state, this may affect the completeness of type checking compared to the context-free version with a fixed size (Fig. 17, left). However, introducing indices is important not only for the simplicity of description but also for the

[8] In LMNtal, a number is represented as a unary atom with the atom name of that number.

expressiveness of the types since we cannot represent red-black trees of *any* height as a context-free type without indices.

6 Functional Atoms

A *functional atom* is a design pattern commonly used in LMNtal programming, which behaves like a function in functional languages. For instance, the atom `append/3` with the following rules, depicted in Fig. 18 (upper), appends two lists like an `append` function in functional languages. The `append/3` atom expects two lists connected to the first and second arguments as input, handles them by the rules, and returns the concatenated list through the third argument as in Fig. 18 (lower).

```
a1@@ R=append(c(L1),L2) :- R=c(append(L1,L2)).
a2@@ R=append(n,L)      :- R=L.
```

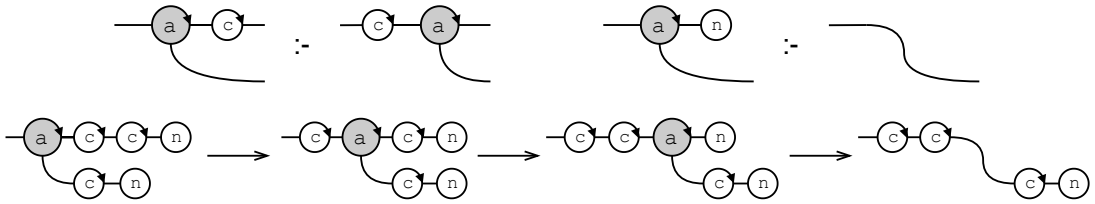


Figure 18: Rules of the `append/3` atom (upper) and progress of computation (lower).

A functional atom may receive and/or return a value of a type with multiple roots. Consider the following functional atom `t2l/3` (for `tree-to-list`):

```
t1@@ t2l(n(N,L,R),T,P) :- t2l(L,c(N,t2l(R,T)),P).
t2@@ t2l(l,T,P) :- T=P.
```

These rules are depicted in Fig. 19 (upper). The atom `t2l`$/3$ receives a binary tree, traverses it in-order, and returns a difference list as in Fig. 19 (lower).

By generalizing them, we formalize functional atoms as follows:

Definition 10 (Functional property)**.** For types $t_1,\ldots,t_n$ and T, a graph F consisting of a single f/m atom, and a ruleset R consisting of rules each of which has just one f/m atom in the LHS, F is *functional* iff for all graphs $G_{t_1},\ldots,G_{t_n}$ which have types $t_1,\ldots,t_n$ respectively,

$$\forall G.\ \left(F, G_{t_1}, \ldots, G_{t_n} \xrightarrow{R}{}^{*}\ G\right) \wedge \left(f/m \notin \mathrm{Funct}(G)\right)\ \Rightarrow\ G : T.$$

This property is written[9] as $t_1,\ldots,t_n \vdash_R F : T$, where $t_1,\ldots,t_n$ are called the *input types*, T the *output type*, and R the *functional ruleset for* F. With this notation, the properties of `append` and `t2l` are expressed as:

$$\texttt{list(L1),list(L2)} \vdash_{\{\texttt{a1,a2}\}} \texttt{append(L1,L2,R)} : \texttt{list(R)},$$
$$\texttt{tree(T)} \vdash_{\{\texttt{t1,t2}\}} \texttt{t2l(T,X,Y)} : \texttt{dlist(X,Y)}.$$

The functional property states that, when a functional atom F is given graphs with types $t_1,\ldots,t_n$ as inputs, it returns a graph with type T upon termination.

The functional property is verified with rule type checking. Here, S_t, P_t, and N_t stand for the start symbol, the set of production rules, and the set of nonterminal symbols of type t, respectively.

[9] In this section, we assume that each type is written as an atom with the functor of its start symbol in order to clarify which argument of the functional atom is connected to which one of the types.

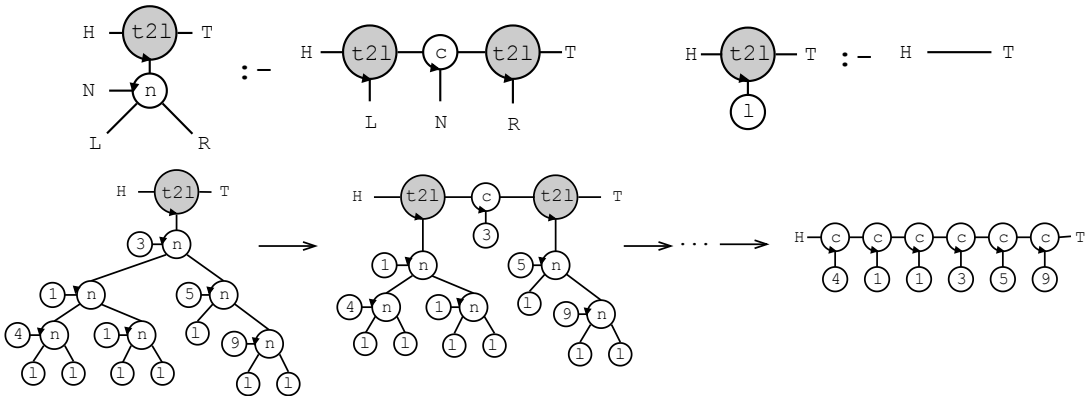


Figure 19: Rules of the `t2l`/3 atom (upper) and progress of computation (lower).

Theorem 5. For a set of production rules $P = P_T \cup P_{t_1} \cup \cdots \cup P_{t_n} \cup \{T :- F, t_1, \ldots, t_n\}$ and a set of nonterminal symbols[10] $N = N_T \cup N_{t_1} \cup \cdots \cup N_{t_n}$, if every rule $r \in R$ preserves type (S_T, P, N), then $t_1, \ldots, t_n \vdash_R F : T$ holds.

For example, to check the functional property of `t2l`, we add a production rule `dlist(X,Y) :- t2l(T,X,Y),tree(T)` and the production rules of the type `tree` to the type `dlist` and and execute the rule type checking of `t1` and `t2`. Note that the functional property does not ensure that the computation with functional atoms will not get stuck because it assumes that the computation successfully terminates and no functional atom remains in the graph.

7 Implementation

While graph type checking can be easily done using the model checking features of SLIM, rule type checking is much harder to implement. Since the rule type checking algorithm requires special features to construct a state space in which states are represented as rewrite rules and states with the same LHSs are considered identical, we implemented this algorithm with the ideas and features of the LMNtal meta-interpreter [14].

To implement indexed types (Section 5.4), we need to perform operations on graphs containing numbers whose concrete values are not necessarily fixed but constrained. Since the current implementation of LMNtal cannot handle such indefinite numbers symbolically, we extended the data structure used in the above implementation on the LMNtal meta-interpreter to handle numerical constraints. These constraints belong to Presburger arithmetic, in which all formulas are decidable. We used the Z3 solver [4] as a backend to solve them. We have tested the implementation by typechecking reasonably complex operations on the graph structures exemplified in the paper, including the insertion operation into red-black trees (formulated using an indexed type) that requires rotation of trees, using functional atoms.

8 Conclusion and Future Work

We studied LMNtal ShapeType, a static type checking framework for a graph rewriting language LMNtal, and proposed extensions to enhance its expressiveness. First, we gave a formalization of the types

[10] Here we assume that there is no duplication among the nonterminal symbols.

and type checking algorithms for both graphs and rewrite rules, addressing various subtleties including those arising from connectors representing link fusion, a key construct in graph rewriting. Second, we proposed *extensive types* and *indexed types* as broader classes of graph types, which enabled us to handle constraints such as the balancing of trees. Third, we proposed a unified approach for checking the type safety of *functional atoms*, i.e., atoms interpreted as functions (as opposed to data constructors) in other languages, so that we could handle multi-step operations which might result in graphs of different types. The expressive power of our rather simple framework was demonstrated by defining a variety of graph types including skip lists with and without constraints, red-black trees, λ-terms represented as graphs, and rectangular grids with arbitrary or specified size, all available from the website given in Section 1.

For future work, it is important to expand the target language. We focused on Flat LMNtal without membranes or guards or hyperlinks [18], but they proved to be useful in computing with complicated graph structures. The challenge here is the handling of process contexts (a wildcard construct in graph matching) and guards (for expressing operations and constraints over built-in types), which is straightforward in forward execution but is challenging in backward execution both in theory and practice. Also, it is interesting to extend the framework to handle infinite graphs as input of functional atoms because the conception of LMNtal was the unified modeling of data structures and network of concurrent processes that evolve by exchanging messages. Functions modeled in LMNtal naturally allow concurrent and nonterminating execution, cooperating with each other by dataflow synchronization, and a unified modeling and reasoning framework of graph types with and without base cases is important future work. Finally, it is necessary to optimize the type checking algorithms. The algorithms worked well as a proof of concept and are simple enough to implement within the existing framework of LMNtal, but the exhaustive search can be too costly when target graphs or rules become complicated. Effective techniques for pruning search is an important topic of future work.

Acknowledgments. The authors would like to thank anonymous reviewers for their useful comments. This work was partially supported by Grant-in-Aid for Scientific Research (B) JP18H03223, JSPS, Japan, and Waseda University Grant for Special Research Projects (2021C-142).

References

[1] Christopher Bak (2015): *GP 2: efficient implementation of a graph programming language*. Ph.D. thesis, Department of Computer Science, The University of York. Available at `https://etheses.whiterose.ac.uk/12586/`.

[2] Adam Bakewell, Detlef Plump & Colin Runciman (2004): *Checking the Shape Safety of Pointer Manipulations*. In: *Proc. RelMiCS 2003*, *LNCS* 3051, Springer, pp. 48–61. Available at `https://doi.org/10.1007/978-3-540-24771-5_5`.

[3] W.-N. Chin & S.-C. Khoo (2001): *Calculating sized types*. *Higher-Order and Symbolic Computation* 14(2–3), pp. 260–300. Available at `https://doi.org/10.1145/328690.328893`.

[4] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An efficient SMT solver*. In: *Proc. TACAS 2008*, *LNCS* 4963, Springer, pp. 337–340. Available at `https://doi.org/10.1007/978-3-540-78800-3_24`.

[5] Pascal Fradet & Daniel Le Métayer (1997): *Shape types*. In: *Proc. POPL'97*, ACM, pp. 27–39. Available at `https://doi.org/10.1145/263699.263706`.

[6] Pascal Fradet & Daniel Le Métayer (1998): *Structured Gamma*. *Science of Computer Programming* 31(2), pp. 263–289. Available at `https://doi.org/10.1016/S0167-6423(97)00023-3`.

[7] A.H. Ghamarian, M. de Mol, A. Rensink, E. Zambon & M. Zimakova (2012): *Modelling and analysis using GROOVE*. *STTT* 14(1), pp. 15–40. Available at `https://doi.org/10.1007/s10009-011-0186-x`.

[8] Masato Gocho, Taisuke Hori & Kazunori Ueda (2011): *Evolution of the LMNtal Runtime to a Parallel Model Checker*. *Computer Software* 28(4), pp. 4_137–4_157. Available at https://doi.org/10.11309/jssst.28.4_137.

[9] Nils Klarlund & Michael I. Schwartzbach (1993): *Graph Types*. In: *Proc. POPL'93*, ACM, pp. 196–205. Available at https://doi.org/10.1145/158511.158628.

[10] William Pugh (1990): *Skip lists: A probabilistic alternative to balanced trees*. *Commun. ACM* 33(6), pp. 668–676. Available at https://doi.org/10.1145/78973.78977.

[11] J.C. Reynolds (2002): *Separation logic: a logic for shared mutable data structures*. In: *Proc. LICS 2002*, IEEE, pp. 55–74. Available at https://doi.org/10.1109/LICS.2002.1029817.

[12] Patrick M. Rondon, Ming Kawaguci & Ranjit Jhala (2008): *Liquid Types*. In: *Proc. PLDI 2008*, ACM, pp. 159–169. Available at https://doi.org/10.1145/1375581.1375602.

[13] Grzegorz Rozenberg (1997): *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific. Available at https://doi.org/10.1142/3303.

[14] Yutaro Tsunekawa, Taichi Tomioka & Kazunori Ueda (2018): *Implementation of LMNtal Model Checkers: A Metaprogramming Approach*. *Journal of Object Technology* 17(1). Available at https://doi.org/10.5381/jot.2018.17.1.a1.

[15] Kazunori Ueda (2008): *Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting*. In: *Proc. RTA2008*, *LNCS* 5117, Springer, pp. 392–408. Available at https://doi.org/10.1007/978-3-540-70590-1_27.

[16] Kazunori Ueda (2009): *LMNtal as a hierarchical logic programming language*. *Theoretical Computer Science* 410(46), pp. 4784–4800. Available at https://doi.org/10.1016/j.tcs.2009.07.043.

[17] Kazunori Ueda (2014): *Towards a Substrate Framework of Computation*. In: *Concurrent Objects and Beyond*, *LNCS* 8665, Springer, pp. 341–366. Available at https://doi.org/10.1007/978-3-662-44471-9_15.

[18] Kazunori Ueda & Seiji Ogawa (2012): *HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model*. *KI - Künstliche Intelligenz* 26(1), pp. 27–36. Available at https://doi.org/10.1007/s13218-011-0162-3.

[19] Yusuke Yoshimoto & Kazunori Ueda (2015): *Static type checking in graph rewriting system*. In: *Proceedings of the 32nd JSSST Annual Conference*. Available at http://jssst.or.jp/files/user/taikai/2015/PPL/ppl3-3.pdf.

Appendix

Hereinafter, for a set of functors F, Graph(F) stands for the set of every graph G s.t. Funct$(G) \subseteq F$.

Proposition 1. For LMNtal rule $r = T :\!- U,\ G' \xrightarrow{r} G \Leftrightarrow G \xrightarrow{r^{\mathrm{inv}}} G'$

Proof. ($\Rightarrow$) We will prove by structural induction on the last-used reduction relation rule.

- Case (R1): We assume that $G' = P, Q,\ G = P', Q$ and $P \xrightarrow{r} P'$. By the induction hypothesis, $P' \xrightarrow{r^{\mathrm{inv}}} P$. By (R1), $P', Q \xrightarrow{r^{\mathrm{inv}}} P, Q$. Therefore, $G \xrightarrow{r^{\mathrm{inv}}} G'$.
- Case (R3): We assume that $G' \equiv P,\ P' \equiv G,\ P \xrightarrow{r} P'$. By the induction hypothesis, $P' \xrightarrow{r^{\mathrm{inv}}} P$. By (R3) and $G' \equiv P,\ P' \equiv G$, we obtain $G \xrightarrow{r^{\mathrm{inv}}} G'$.
- Case (R6) is self-evident by (R6).

Hence $G' \xrightarrow{r} G \Rightarrow G \xrightarrow{r^{\mathrm{inv}}} G'$.

($\Leftarrow$) follows from $(r^{\mathrm{inv}})^{\mathrm{inv}} = r$ and ($\Rightarrow$). □

Lemma 1. For an extensive ShapeType $\tau = (t/m, P, N)$, its weighting function w, and an LMNtal graph $G, G' \in$ Graph(Funct(τ)),

$$G' \xrightarrow{P} G \Rightarrow w(G) \geq w(G')$$

Proof. By $G' \xrightarrow{P} G$, let $p = \alpha :\!- \beta$ be a rule s.t. $G' \xrightarrow{p} G$. We will show $w(G) \geq w(G')$ by structural induction on the last-used reduction relation rule.

- Case (R1): We assume that $G' = G'_1, G_2,\ G = G_1, G_2$ and $G'_1 \xrightarrow{p} G_1$. Then $w(G_1) \geq w(G'_1)$ by the induction hypothesis. We have $w(G') = w(G'_1) + w(G_2),\ w(G) = w(G_1) + w(G_2)$, therefore $w(G) \geq w(G')$ holds.
- Case (R3): We assume that $G'_1 \equiv G',\ G \equiv G_1,\ G'_1 \xrightarrow{p} G_1$. Then $w(G_1) \geq w(G'_1)$ by the induction hypothesis. We have $w(G_1) = w(G),\ w(G'_1) = w(G')$, therefore $w(G) \geq w(G')$ holds.
- Case (R6): We assume that $G' = \alpha,\ G = \beta$. Since the type $(t/m, P, N)$ is extensive, we have $w(\alpha) \leq w(\beta)$ i.e. $w(G) \geq w(G')$. □

Lemma 2. For an extensive ShapeType $\tau = (t/m, P, N)$, its weighting function w, a non-negative integer n, and a finite set of links L, the number of LMNtal graph G satisfying following formula is *finite* regarding structurally congruent graphs as the same.

$$G \in \text{Graph}(\text{Funct}(\tau)) \wedge w(G) = n \wedge \text{FLink}(G) = L$$

Proof. The number of functors occurring in G is finite and also the number of atoms (excluding non-global connectors) occurring in G is less than n because of $w(G) = n$. Since the number of graphs consisting of finite kinds of functors and finite atoms is finite, the number of G is finite. □

Lemma 3. For any LMNtal graph X, an extensive ShapeType $(t/m, P, N)$, and a finite set of LMNtal graphs G, GGCHECK$(X, (t/m, P, N, G))$ terminates.

Proof. Let the weighting function of type $(t/m,P,N)$ be w. Y given to the first argument of recursive call at line 5 of GGCHECK satisfies $Y \xrightarrow{P}{}^{*} X$. By Lemma 1, $w(X) \geq w(Y)$. By this and Lemma 2, the number of possible Y is finite (regarding structurally congruent graphs are the same). Also, it is verified at line 3 of GGCHECK that structurally congruent graphs cannot be given again to the argument of recursive calls, so that recursive calls occur finite times at most. Therefore $\mathrm{GGCHECK}(X,(t/m,P,N),G)$ terminates. □

Theorem 1 (Termination of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType τ, $\mathrm{GCHECK}(X,\tau)$ terminates.

Proof. This follows by Lemma 3 and the fact that $\mathrm{Funct}(X)$ and N are finite. □

Lemma 4. For any LMNtal graph X, a ShapeType $(t/m,P,N)$, and a finite set of LMNtal graphs G, if $\mathrm{GGCHECK}(X,(t/m,P,N),G)$ returns **true**, $X \lhd t(L_1,\ldots,L_m)$ holds.

Proof. We will show by induction on the maximum number n of times of recursive calls (i.e. the depth of recursion) of GGCHECK.

- If $n=0$, **true** is returned at line 2. Also, we have $X \equiv t(L_1,\ldots,L_m)$. Then it is clear by the definition that $X \lhd t(L_1,\ldots,L_m)$.
- If $n=k+1$ $(k\geq 0)$, **true** is returned at line 5 since recursive calls occur one or more times. We have $Y \xrightarrow{P} X$ by the line 3 and $Y \lhd t(L_1,\ldots,L_m)$ by the induction hypothesis. Then $t(L_1,\ldots,L_m) \xrightarrow{P}{}^{*} Y$ follows by the definition of production relations. By $Y \xrightarrow{P} X$, we have $Y \xrightarrow{P} X$, so that $t(L_1,\ldots,L_m) \xrightarrow{P}{}^{*} X$. Therefore $X \lhd t(L_1,\ldots,L_m)$ holds. □

Theorem 2 (Soundness of graph type checking)**.** For any LMNtal graph X and a ShapeType $(t/m,P,N)$, if $\mathrm{GCHECK}(X,(t/m,P,N))$ returns **true**, $X : t(L_1,\ldots,L_m)$ holds.

Proof. $\mathrm{GCHECK}(X,(t/m,P,N))$ returns **true** only when $\exists f \in \mathrm{Funct}(X).\ f \in N$ does not hold and then it just returns the returned value from $\mathrm{GGCHECK}(X,(t/m,P,N),\varnothing)$, so that $\mathrm{GGCHECK}(X,(t/m,P,N),\varnothing)$ returns **true**. By Lemma 4, we have $X \lhd t(L_1,\ldots,L_m)$. Since $\neg\exists f \in \mathrm{Funct}(X).\ f \in N$ holds, we have $\forall f \in \mathrm{Funct}(X).\ f \notin N$. Therefore $\mathrm{Funct}(X) \cap N = \varnothing$ holds. Hence we have $X : t(L_1,\ldots,L_m)$. □

Lemma 5. For an LMNtal graph X and a ShapeType $(t/m,P,N)$, if $X \lhd t(L_1,\ldots,L_m)$, $\mathrm{GGCHECK}(X,(t/m,P,N),\varnothing)$ returns **true**.

Proof. By $X \lhd t(L_1,\ldots,L_m)$, for certain $X_0,\ldots,X_n$ $(n\geq 0)$, the following holds:

$$t(L_1,\ldots,L_m) = X_n \xrightarrow{P} \ldots \xrightarrow{P} X_1 \xrightarrow{P} X_0 = X$$

Note that X_i is not the start symbol for every i $(i<n)$ and $i \neq j \Rightarrow X_i \not\equiv X_j$ holds (i.e. no loops in the path). Consider the case when $\mathrm{GGCHECK}(Y_i,(t/m,P,N),G_i)$ is called for i $(i<n)$, Y_i s.t. $Y_i \equiv X_i$, and a certain G_i. Since X_i is not the start symbol, Y_i is also not the start symbol, so that the condition of the if statement at line 2 does not hold. Then we have $X_{i+1} \xrightarrow{P} Y_i$ by $X_{i+1} \xrightarrow{P} X_i$ and (R3).

- If $\nexists Y_{i+1} \in G_i.\ X_{i+1} \equiv Y_{i+1}$, the for-loop from the line 3 is executed for $Y \leftarrow X_{i+1}$, and then $\mathrm{GGCHECK}(X_{i+1},(t/m,P,N),G_{i+1})$ is called for a certain G_{i+1} at line 5.
- If $\exists Y_{i+1} \in G_i.\ X_{i+1} \equiv Y_{i+1}$, $\mathrm{GGCHECK}(Y_{i+1},(t/m,P,N),G_{i+1})$ has been called for a certain G_{i+1} elsewhere.

Therefore GGCHECK$(Y_{i+1},(t/m,P,N),G_{i+1})$ is called for Y_{i+1} s.t. $Y_{i+1}\equiv X_{i+1}$ and a certain G_{i+1} somewhere in the recursive calls.

From the above reasons, when GGCHECK$(X,(t/m,P,N),\varnothing)$ is called, GGCHECK$(Y_n,(t/m,P,N),G_n)$ is also called for Y_n s.t. $Y_n\equiv X_n$ and a certain G_n. This call returns **true** at line 2 because of $Y_n\equiv X_n = t(L_1,\ldots,L_m)$.

Therefore GGCHECK$(X,(t/m,P,N),\varnothing)$ returns **true** since GGCHECK returns **true** if one or more recursive calls in it return **true**. □

Theorem 3 (Completeness of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType $(t/m,P,N)$, if $X:t(L_1,\ldots,L_m)$ holds, GCHECK$(X,(t/m,P,N))$ returns **true**.

Proof. By $X:t(L_1,\ldots,L_m)$, we have $X\lhd t(L_1,\ldots,L_m)$ and $\mathrm{Funct}(X)\cap N=\varnothing$. Therefore $\forall f\in Funct(X).\ f\notin X$, so that the condition of the if statement at line 2 of GCHECK does not hold. Then GCHECK$(X,(t/m,P,N))$ just returns the returned value from GGCHECK$(X,(t/m,P,N),\varnothing)$. By $X\lhd t(L_1,\ldots,L_m)$ and Lemma 5, GGCHECK$(X,(t/m,P,N),G)$ returns **true**. Therefore GCHECK$(X,(t/m,P,N))$ returns **true**. □

Definition 11. A transition relation $\mathscr{T}$ between LMNtal rules $\alpha_1 :- \beta_1$ and $\alpha_2 :- \beta_2$ is defined as follows:

$$\begin{aligned}&\alpha_1 :- \beta_1 \xrightarrow{\mathscr{T}} \alpha_2 :- \beta_2\\ &\quad\text{iff}\quad \exists \alpha_p :- \beta_p \in P.\ \exists \gamma,\gamma'.\\ &\qquad \alpha_2 \xrightarrow{\alpha_p :- \beta_p} \alpha_1,\gamma \wedge \beta_2 \equiv \beta_1,\gamma\\ &\qquad\quad \wedge\ \beta_p \equiv \gamma,\gamma' \wedge \gamma' \not\equiv \mathbf{0}\end{aligned}$$

Next, we define a labeling function $\mathscr{L}:\mathscr{W}\to 2^{\{\mathtt{s},\mathtt{r}\}}$ as follows, where $\mathscr{W}$ is the whole set of LMNtal rules:

$$\begin{aligned}\mathtt{s}\in\mathscr{L}(\alpha :- \beta) \quad&\text{iff}\quad \alpha\equiv T\\ \mathtt{r}\in\mathscr{L}(\alpha :- \beta) \quad&\text{iff}\quad \alpha \xrightarrow{P}{}^{*}\ \beta\end{aligned}$$

If $\mathtt{r}\in\mathscr{L}(\alpha :- \beta)$, we say $\alpha :- \beta$ is *reducible*. Then we consider a Kripke structure $\mathscr{S}=(\mathscr{W},\mathscr{T},\mathscr{L})$ which represents the state space of the rule type checking algorithm.

Lemma 6. If $\alpha :- \beta$ is reducible and $\alpha :- \beta \xrightarrow{\mathscr{T}} \alpha' :- \beta'$, then $\alpha' :- \beta'$ is reducible.

Proof. By the assumption, we have $\alpha \xrightarrow{P}{}^{*}\ \beta$. By the definition of $\mathscr{T}$, we have $\exists\gamma.\ \alpha' \xrightarrow{P} \alpha,\gamma\beta'\equiv\beta,\gamma$. Then we have $\alpha' \xrightarrow{P} \alpha,\gamma \xrightarrow{P}{}^{*}\ \beta,\gamma\equiv\beta'$, and $\alpha' \xrightarrow{P}{}^{*}\ \beta'$ holds. □

Lemma 7. If $P,Q\equiv R,S$ holds, $P\equiv A_1,A_2,\ Q\equiv A_3,A_4,\ R\equiv A_1,A_3,\ S\equiv A_2,A_4$ holds for certain graphs A_1,A_2,A_3,A_4.

Proof. This follows by the rules of structural congruence. □

Lemma 8. If $P \xrightarrow{\alpha :- \beta} Q$ holds, there exists a graph C that satisfies $P\equiv C,\alpha,\ Q\equiv C,\beta$.

Proof. This follows by the rules of structural congruence and reduction relation. □

Lemma 9. If $X \xrightarrow{p} Y\equiv\alpha,C\ (p=\alpha_p :- \beta_p\in P)$ holds, one of the followings holds:

- $\forall\beta.\ \exists\alpha',\beta',C_1,C_2.\ \alpha :\!- \beta \xrightarrow{\mathscr{T}} \alpha' :\!- \beta' \ \wedge\ C \equiv C_1\,,C_2 \wedge X \equiv \alpha'\,,C_2 \wedge \beta' \equiv \beta\,,C_1$
- $\exists C'.\ X \equiv \alpha\,,C' \wedge C' \xrightarrow{p} C$

Proof. By $X \xrightarrow{p} Y$ and Lemma 8, there exists C_p s.t. $X \equiv \alpha_p\,,C_p,\ Y \equiv \beta_p\,,C_p$. Then we have $Y \equiv \beta_p\,,C_p \equiv \alpha\,,C$ and, by Lemma 7, there exist C_1,C_2,C_3,C_4 s.t. $C \equiv C_1\,,C_2,\ \alpha \equiv C_3\,,C_4,\ \beta_p \equiv C_1\,,C_3,\ C_p \equiv C_2\,,C_4$.

Case 1: $C_3 \not\equiv \mathbf{0}$

Let $\alpha' = \alpha_p\,,C_4,\ \beta' = \beta\,,C_1$. Then we have $\alpha' \equiv \alpha_p\,,C_4 \xrightarrow{p} \beta_p\,,C_4 \equiv C_1\,,C_3\,,C_4 \equiv \alpha\,,C_1$. Here we consider C_1,C_3 as γ,γ' in the definition $\mathscr{T}$ respectively, and we have $\alpha :\!- \beta \xrightarrow{\mathscr{T}} \alpha' :\!- \beta'$. Also, we have $X \equiv \alpha_p\,,C_p \equiv \alpha_p\,,C_2\,,C_4 \equiv \alpha'\,,C_2$.

Case 2: $C_3 \equiv \mathbf{0}$

By $C_1 \equiv \beta_p$, we have $C \equiv \beta_p\,,C_2$. Therefore $\alpha_p\,,C_2 \xrightarrow{p} C$ holds. Here we consider C' s.t. $C' \equiv \alpha_p\,,C_2$, then $C' \xrightarrow{p} C$ holds. Besides, by $\alpha \equiv C_4$, we have $C_p \equiv C_2\,,\alpha$. Then we have $X \equiv \alpha_p\,,C_p \equiv \alpha_p\,,C_2\,,\alpha \equiv \alpha\,,C'$. □

Lemma 10. If $\mathscr{S},r \models \neg\mathtt{s}\,\mathsf{W}\,\mathtt{r}$ holds, r preserves the type T.

Proof. We assume $G \lhd T$, $G \xrightarrow{r} G'$, and $\mathscr{S},r \models \neg\mathtt{s}\,\mathsf{W}\,\mathtt{r}$, and we will prove $G' \lhd T$.

By $G \lhd T$, we have $\forall i < n.\ X_{i+1} \xrightarrow{p_i} X_i$ for a certain non-negative integer n, graphs $X_0,\ldots,X_n$ ($X_0 = G, X_n = T$), and $p_0,\ldots,p_{n-1} \in P$. By $G \xrightarrow{r} G'$, there exists $C \in \mathscr{G}(N \cup \Sigma)$ s.t. $G \equiv \alpha\,,C,\ G' \equiv \beta\,,C$ where $r = \alpha :\!- \beta$.

Next, we will show that, if $X_i \equiv \alpha_i\,,C_i$ holds, there exist $\alpha_{i+1},\beta_{i+1},C_{i+1}$ such that:

$$\alpha_i :\!- \beta_i \xrightarrow{\mathscr{T}}{}^{*}\ \alpha_{i+1} :\!- \beta_{i+1}\ \wedge\ X_{i+1} \equiv \alpha_{i+1}\,,C_{i+1}\ \wedge\ \beta_{i+1}\,,C_{i+1} \xrightarrow{P}{}^{*}\ \beta_i\,,C_i$$

By $X_{i+1} \xrightarrow{p_i} X_i$ and Lemma 9, one of the following holds:

1. $\exists\alpha_{i+1},\beta_{i+1},C_i',C_{i+1}.\ \alpha_i :\!- \beta_i \xrightarrow{\mathscr{T}} \alpha_{i+1} :\!- \beta_{i+1} \wedge C_i \equiv C_i'\,,C_{i+1} \wedge X_{i+1} \equiv \alpha_{i+1}\,,C_{i+1} \wedge \beta_{i+1} \equiv \beta_i\,,C_i'$
2. $\exists C_{i+1}.\ X_{i+1} \equiv \alpha_i\,,C_{i+1} \wedge C_{i+1} \xrightarrow{p_i} C_i$

If 1. holds, it is obvious since we have $\beta_{i+1}\,,C_{i+1} \equiv \beta_i\,,C_i'\,,C_{i+1} \equiv \beta_i\,,C_i$. On the other hand, if 2. holds, it is obvious when we consider $\alpha_{i+1} = \alpha_i,\ \beta_{i+1} = \beta_i$.

Thus, there exist α_n,β_n,C_n s.t. $\alpha :\!- \beta \xrightarrow{\mathscr{T}}{}^{*}\ \alpha_n :\!- \beta_n,\ T \equiv \alpha_n\,,C_n$, and $\beta_n\,,C_n \xrightarrow{P}{}^{*}\ \beta\,,C$. Since T consists only of one atom of the start symbol (with no self loops), we have $T \equiv \alpha_n$. Then $\mathtt{s} \in \mathscr{L}(\alpha_n :\!- \beta_n)$ holds.

By $r \xrightarrow{\mathscr{T}}{}^{*}\ \alpha_n :\!- \beta_n$, $\mathscr{S},r \models \neg\mathtt{s}\,\mathsf{W}\,\mathtt{r}$, and Lemma 6, we also have $\mathtt{r} \in \mathscr{L}(\alpha_n :\!- \beta_n)$. Therefore we have $\alpha_n \xrightarrow{P}{}^{*}\ \beta_n$. Thus we have $T \equiv \alpha_n \xrightarrow{P}{}^{*}\ \beta_n \xrightarrow{P}{}^{*}\ \beta\,,C \equiv G'$, that is, $G' \lhd T$. □

Theorem 4 (Soundness of rule type checking)**.** For an LMNtal rule $\alpha :\!- \beta$, a ShapeType $(t/m,P,N)$, and a sequence of links $L_1,\ldots,L_m$, if RCHECK$(\alpha :\!- \beta,(t/m,P,N))$ returns **true**, the following formula (the rule preserving property) holds:

$$\forall G : t\,(L_1\,,\ldots,L_m).\quad G \xrightarrow{\alpha :\!- \beta} G' \Rightarrow G' : t\,(L_1\,,\ldots,L_m)$$

Proof. Since RCHECK$(\alpha :\!- \beta,(t/m,P,N))$ returns **true**, on all the paths from the target rule to the start symbol, there exists a state $L :\!- R$ such that REDUCE$(R,L,P,\varnothing)$ returns **true**. Therefore we have $\mathscr{S},r \models \neg\mathtt{s}\,\mathsf{W}\,\mathtt{r}$, and the rule preserving property holds by Lemma 10. □

Theorem 5. For a set of production rules $P = P_T \cup P_{t_1} \cup \cdots \cup P_{t_n} \cup \{T :- F, t_1, \ldots, t_n\}$, and a set of nonterminal symbols $N = N_T \cup N_{t_1} \cup \cdots \cup N_{t_n}$, if every rule $r \in R$ preserves type (S_T, P, N), $t_1, \ldots, t_n \vdash_R F : T$ holds.

Proof. $F, G_{t_1}, \ldots, G_{t_n}$ has the type (S_T, P, N) because P includes the production rule $T :- F, t_1, \ldots, t_n$ and T is the start symbol. Let G be a graph s.t. $F, G_{t_1}, \ldots, G_{t_n} \xrightarrow{R}{}^* G$. Then G has the type (S_T, P, N) because every rule $r \in R$ preserves the type. Here we assume that G contains no f/m atoms. By $G : (S_T, P, N)$, there exists a production path s.t. $S_T \xrightarrow{P}{}^* G$. Since G contains no f/m atoms, the production rule $T :- F, t_1, \ldots, t_n$ has not been applied in the production path. Also the nonterminal symbols of the types $t_1, \ldots, t_n$ do not appear in the production path because they can appear only after the production rule $T :- F, t_1, \ldots, t_n$ is applied. Therefore the production rules of the types $t_1, \ldots, t_n$ have not been applied in the production path, so that the nonterminal symbols $N_{t_1}, \ldots, N_{t_n}$ and the production rules $P_{t_1}, \ldots, P_{t_n}$ are redundant in the production path. Hence we have $G : (S_T, P_T, N_T) = T$. By Definition 10, $t_1, \ldots, t_n \vdash_R F : T$ holds. □

A Small-Step Operational Semantics for GP 2

Brian Courtehoute and Detlef Plump
Department of Computer Science, University of York, York, UK
{bc956,detlef.plump}@york.ac.uk

The operational semantics of a programming language is said to be small-step if each transition step is an atomic computation step in the language. A semantics with this property faithfully corresponds to the implementation of the language. The previous semantics of the graph programming language GP 2 is not fully small-step because the loop and branching commands are defined in big-step style. In this paper, we present a truly small-step operational semantics for GP 2 which, in particular, accurately models diverging computations. To obtain small-step definitions of all commands, we equip the transition relation with a stack of host graphs and associated operations. We prove that the new semantics is non-blocking in that every computation either diverges or eventually produces a result graph or the failure state. We also show the finite nondeterminism property, viz. that each configuration has only a finite number of direct successors. The previous semantics of GP 2 is neither non-blocking nor does it have the finite nondeterminism property.

1 Introduction

GP 2 is a nondeterministic programming language based on graph transformation rules. The previous semantics of GP 2 is defined by both small-step and big-step inference rules [17]. An operational semantics is *small-step* if transition steps are atomic computation steps in the language, meaning that the implementation of the language faithfully corresponds to the semantics. In this paper, we present a truly small-step operational semantics for GP 2 which, in particular, accurately models diverging computations.

While the previous semantics (Figure 3) has small-step elements, the branching and loop constructs are not small-step. This can lead to the semantic transition sequence *blocking* or *getting stuck* [14], i.e. reaching a configuration which is neither a graph nor the failure state, such that no inference rule is applicable.

To illustrate this situation, consider the program `P = try (r1!) then skip else skip`, with the rule `r1` : $_1$● ⇒ $_1$● ○, applied to the host graph ●. The statement `r1!` means that the rule `r1` is called until it is no longer applicable. The `try` statement attempts to evaluate `r1!` but will neither branch to the `then` nor the `else` part because the loop `r1!` diverges on ●. In the previous semantics, `try` statements are handled with the following inference rules :

$$[\text{try}'_1]\ \frac{\langle C, G\rangle \rightsquigarrow^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightsquigarrow \langle P, H\rangle} \qquad [\text{try}'_2]\ \frac{\langle C, G\rangle \rightsquigarrow^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightsquigarrow \langle Q, G\rangle}$$

The premises of these inference rules are that the conditional part C of a `try` statement applied to host graph G results in either a graph H or failure, which determines whether P or Q is called. If $\langle C, G\rangle$ diverges (does not terminate) however, neither rule applies. Since there are no other `try` rules, the transition sequence gets stuck.

The new semantics we introduce in this paper handles `try` statements with the following rules:

$$[\text{try}_1]\ \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, S\rangle \to \langle \text{TRY}(C, P, Q), \text{push}(\text{top}(S), S)\rangle$$

$$[\text{try}_2]\ \frac{\langle C, S\rangle \to \langle C', S'\rangle}{\langle \text{TRY}(C,P,Q), S\rangle \to \langle \text{TRY}(C',P,Q), S'\rangle} \qquad [\text{try}_3]\ \frac{\langle C, S\rangle \to S'}{\langle \text{TRY}(C,P,Q), S\rangle \to \langle P, \text{pop2}(S')\rangle}$$

$$[\text{try}_4]\ \frac{\langle C, S\rangle \to \text{fail}}{\langle \text{TRY}(C,P,Q), S\rangle \to \langle Q, \text{pop}(S)\rangle}$$

Here S and S' are stacks of graphs. The rule $[\text{try}_1]$ duplicates the top of the stack, and the TRY construct signals that the copy operation has happened. Repeated applications of the inference rule $[\text{try}_2]$ model the evaluation of the condition in a small-step fashion. If the condition loops, $[\text{try}_2]$ can be applied indefinitely, and we get an infinite transition sequence.

In the implementation of GP 2, `P` indeed loops. In the previous semantics however, `P` gets stuck because `r1!` diverges, which means that we cannot apply either of the inference rules $[\text{try}'_1]$ or $[\text{try}'_2]$ to resolve the `try` statement.

The previous semantics tries to remedy this issue in the *semantic function* which associates to a program `P` and host graph G the set $[\![\texttt{P}]\!]G$ of all possible outcomes of the execution of `P` on G. These outcomes can be a graph, the element fail, or $\bot$ which represents an infinite transition sequence. The previous semantic function uses $\bot$ as an outcome if the transition sequence gets stuck. However, there are problems with this approach.

Consider the program `P = try Loop then skip else skip`, where `Loop = {r1,r2}!`, `r1` is as previously defined, and `r2` : ● ⇒ ∅. The command `{r1,r2}` is a *rule set call*, meaning that rules `r1` and `r2` are selected nondeterministically. When `P` is executed on the host graph ●, an application of `r2` causes the loop to terminate since it removes the marked node which is necessary for either rule to be applicable. Hence `r1` may be applied a number of times, and then `r2` is applied once. But it should also be possible that `r2` is never called, resulting in a diverging computation. Hence the set of outcomes we want is $\{\bot, \emptyset, ○, ○○, ○○○, \dots\}$. According to the previous semantics, however, the execution of `P` on ● cannot get stuck since `Loop` *can* always transition to a graph; and by the rules $[\text{try}'_1]$ and $[\text{try}'_2]$, the execution cannot diverge either. So $\bot \notin [\![\texttt{P}]\!]● = \{\emptyset, ○, ○○, ○○○, \dots\}$.

This may also lead to two programs being semantically equivalent, even though they should not be. Programs P and P' are *semantically equivalent* if $[\![P]\!] = [\![P']\!]$, i.e. they have the same outcomes for all host graphs. Consider the program `P = try ({r3,r2}!) then skip else skip`, where `r3` : $_1● \Rightarrow {}_1●$. It can diverge but is semantically equivalent to `Q = try r2 then skip else skip` since the previous semantics cannot detect that divergence. For instance, $[\![\texttt{P}]\!]● = [\![\texttt{Q}]\!]● = \{\emptyset\}$, but $[\![\texttt{P}]\!]●$ should include $\bot$.

The aforementioned issues can also happen with `if` statements, which work similarly to `try` statements, except that the changes the condition made to the host graph are reversed, even if the evaluation of the condition succeeds. Nested loops such as `Loop!` can get stuck as well since their inference rules also assume that the loop body either results in a graph or fails.

Diverging computations not being modelled properly entails an undesirable property, namely *infinite nondeterminism*, i.e. there can be infinitely many configurations reachable in a single transition step. Consider the program `P = try Loop then skip else skip`, where `Loop = {r1,r2}!`, and the rules are as previously defined. We have $[\![\texttt{Loop}]\!]● = \{\emptyset, ○, ○○, ○○○, \dots, \bot\}$. In a transition sequence starting with $\langle \texttt{P}, ●\rangle$, since the `try` statement is resolved within a single step, it only takes one step to transition to either of the graphs in the set $\{\emptyset, ○, ○○, ○○○, \dots\}$, of which there are infinitely many.

The semantics we introduce in this paper is truly small-step and as such, it accurately models looping computations with diverging transition sequences. When starting with a valid GP 2 program, it cannot get stuck, which is a property we call *non-blocking*. As a consequence of the small-step approach, we get *finite nondeterminism*, meaning we can only reach a finite number of configurations within a single

transition step.

In Section 2, we give a brief overview of the rule-based graph programming language GP 2 along with the previous semantics. We propose the new semantics in Section 3 and give examples of transition sequences. In Section 4 we prove several properties of the new semantics, including non-blocking as well as finite nondeterminism, and define the semantic function along with semantic equivalence.

2 The Graph Programming Language GP 2

This section provides a brief introduction to GP 2 [16], a nondeterministic graph programming language based on transformation rules. We show the abstract syntax of GP 2 programs below, and refer to [3] for the full syntax. The language is implemented by a compiler generating C code [4].

GP 2 programs transform input graphs into output graphs, where graphs are labelled and directed and may contain parallel edges and loops.

The principal programming construct in GP 2 are conditional graph transformation rules labelled with expressions. For example, Figure 1 shows a program recognising graphs that contain cycles and the declaration of its rules. The rule `delete` which has three formal parameters, a left-hand graph and a right-hand graph which are specified graphically, and a textual condition starting with the keyword `where`. The small numbers attached to nodes are identifiers, all other text in the graphs are labels.

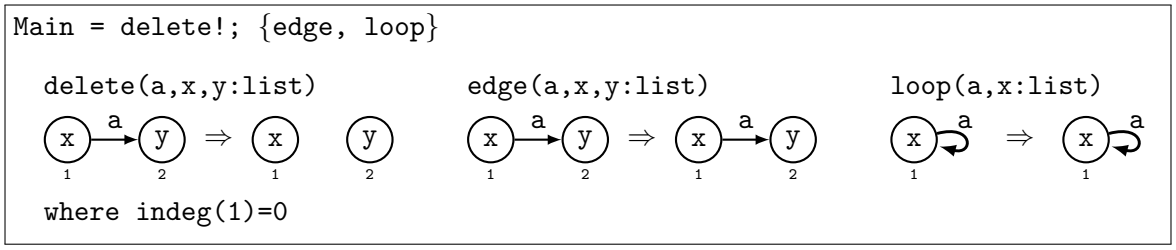


Figure 1: GP 2 program recognising cyclic graphs

GP 2 labels consist of an expression and an optional mark (explained below). Expressions are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`, and `list` is the type of a (possibly empty) list of atoms. Lists of length one are equated with their entries and hence every expression can be considered as a list.

The concatenation of two lists x and y is written $x{:}y$, the empty list is denoted by `empty`. Character strings are enclosed in double quotes. Composite arithmetic expressions such as `n*n` must not occur in the left-hand graph, and all variables occurring in the right-hand graph or the condition must also occur in the left-hand graph.

Besides carrying list expressions, nodes and edges can be *marked*. For example, one of the nodes in rule r1 in the introduction is marked by a grey shading. Marks are convenient to highlight items in input or output graphs, and to record visited items during a graph traversal. For instance, a graph can be checked for connectedness by propagating marks along edges as long as possible and subsequently testing whether any unmarked nodes remain. Note that conventional graph algorithms are often described by using marks as a visual aid [7].

Additionally, nodes in rules and host graphs can be *rooted*. If such a node appears in the left-hand side of a rule, it can only be matched with a root node in the host graph. Their use restricts matching to the neighbourhoods of root nodes, which can greatly increase efficiency [6].

```
Program         ::= Declaration { Declaration }
Declaration     ::= MainDecl | ProcedureDecl | RuleDecl
MainDecl        ::= Main '=' CommandSeq
ProcedureDecl   ::= ProcedureID '=' [ '[' LocalDecl ']' ] CommandSeq
LocalDecl       ::= ( RuleDecl | ProcedureDecl ) { LocalDecl }
CommandSeq      ::= Command {';'Command}
Command         ::= Block
                  | if Block then Block [ else Block ]
                  | try Block [ then Block ] [ else Block ]
Block           ::= '(' CommandSeq ')' ['!']
                  | SimpleCommand
                  | Block or Block
SimpleCommand   ::= RuleSetCall ['!']
                  | ProcedureCall ['!']
                  | break
                  | skip
                  | fail
RuleSetCall     ::= RuleID | '{' [ RuleID { ',' RuleID } ] '}'
ProcedureCall   ::= ProcedureID
```

Figure 2: GP 2 Program Syntax

We do not elaborate any further on features such as marks or roots because the GP 2 semantics does not depend on them.

Rules operate on *host graphs* which are labelled with constant values (lists containing integer and string constants). Applying a rule $L \Rightarrow R$ to a host graph G works roughly as follows: (1) Replace the variables in L and R with constant values and evaluate the expressions in L and R, to obtain an instantiated rule $\hat{L} \Rightarrow \hat{R}$. (2) Choose a subgraph S of G isomorphic to $\hat{L}$ such that the dangling condition and the rule's application condition are satisfied (see below). (3) Replace S with $\hat{R}$ as follows: numbered nodes stay in place (possibly relabelled), edges and unnumbered nodes of $\hat{L}$ are deleted, and edges and unnumbered nodes of $\hat{R}$ are inserted.

In this construction, the *dangling condition* requires that nodes in S corresponding to unnumbered nodes in $\hat{L}$ (which should be deleted) must not be incident with edges outside S. The rule's application condition is evaluated after variables have been replaced with the corresponding values of $\hat{L}$, and node identifiers of L with the corresponding identifiers of S. For example, the term $\texttt{indeg}(\texttt{1}) = \texttt{0}$ in the condition of `delete` in Figure 1 forbids the node $g(\texttt{1})$ to have incoming edges, where $g(\texttt{1})$ is the node in S corresponding to `1`.

Formally, GP 2 is based on a form of attributed graph transformation according to the so-called double-pushout approach [11, 9]. The grammar in Figure 2 gives the abstract syntax of GP 2 programs. A program consists of declarations of conditional rules and procedures, and exactly one declaration of a main command sequence. The category RuleID refers to declarations of conditional rules in RuleDecl (whose syntax is omitted). Procedures must be non-recursive, they can be seen as macros with local declarations.

The call of a rule set $\{r_1, \ldots, r_n\}$ nondeterministically applies one of the rules whose left-hand graph

$$[\text{call}_1]\ \frac{G \Rightarrow_R H}{\langle R, G\rangle \rightsquigarrow H} \qquad [\text{call}_2]\ \frac{G \not\Rightarrow_R}{\langle R, G\rangle \rightsquigarrow \text{fail}}$$

$$[\text{seq}_1]\ \frac{\langle P, G\rangle \rightsquigarrow \langle P', H\rangle}{\langle P;Q, G\rangle \rightsquigarrow \langle P';Q, H\rangle} \qquad [\text{seq}_2]\ \frac{\langle P, G\rangle \rightsquigarrow H}{\langle P;Q, G\rangle \rightsquigarrow \langle Q, H\rangle}$$

$$[\text{seq}_3]\ \frac{\langle P, G\rangle \rightsquigarrow \text{fail}}{\langle P;Q, G\rangle \rightsquigarrow \text{fail}}$$

$$[\text{if}_1]\ \frac{\langle C, G\rangle \rightsquigarrow^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightsquigarrow \langle P, G\rangle} \qquad [\text{if}_2]\ \frac{\langle C, G\rangle \rightsquigarrow^+ \text{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightsquigarrow \langle Q, G\rangle}$$

$$[\text{try}_1]\ \frac{\langle C, G\rangle \rightsquigarrow^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightsquigarrow \langle P, H\rangle} \qquad [\text{try}_2]\ \frac{\langle C, G\rangle \rightsquigarrow^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightsquigarrow \langle Q, G\rangle}$$

$$[\text{alap}_1]\ \frac{\langle P, G\rangle \rightsquigarrow^+ H}{\langle P!, G\rangle \rightsquigarrow \langle P!, H\rangle} \qquad [\text{alap}_2]\ \frac{\langle P, G\rangle \rightsquigarrow^+ \text{fail}}{\langle P!, G\rangle \rightsquigarrow G}$$

$$[\text{alap}_3]\ \frac{\langle P, G\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle}{\langle P!, G\rangle \rightsquigarrow H} \qquad [\text{break}]\ \langle \texttt{break};P, G\rangle \rightsquigarrow \langle \texttt{break}, G\rangle$$

(a) Inference rules for core commands

$$[\text{or}_1]\quad \langle P\ \texttt{or}\ Q, G\rangle \rightsquigarrow \langle P, G\rangle \qquad [\text{or}_2]\quad \langle P\ \texttt{or}\ Q, G\rangle \rightsquigarrow \langle Q, G\rangle$$

$$[\text{skip}]\quad \langle \texttt{skip}, G\rangle \rightsquigarrow G \qquad [\text{fail}]\quad \langle \texttt{fail}, G\rangle \rightsquigarrow \text{fail}$$

$$[\text{if}_3]\quad \langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \rightsquigarrow \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}_3]\quad \langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}_4]\quad \langle \texttt{try}\ C\ \texttt{else}\ P, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then skip else}\ P, G\rangle$$

$$[\text{try}_5]\quad \langle \texttt{try}\ C, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then skip else skip}, G\rangle$$

(b) Inference rules for derived commands

Figure 3: Previous GP 2 Semantics

matches a subgraph of the host graph such that the dangling condition and the rule's application condition are satisfied. The call *fails* if none of the rules is applicable to the host graph.

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution in case C succeeds.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop with the current graph and transfers control to the command following the loop.

A program P `or` Q non-deterministically chooses to execute either P or Q, which can be simulated by a rule-set call and the other commands [16]. The commands `skip` and `fail` can also be expressed by the other commands: `skip` is equivalent to an application of the rule $\emptyset \Rightarrow \emptyset$ (where $\emptyset$ is the empty graph) and `fail` is equivalent to an application of $\{\}$ (the empty rule set).

Like Plotkin's structural operational semantics [15], the previous GP 2 semantics is given by inference rules. The rules in Figure 3 define the transition relation $\rightsquigarrow$ over the following set:

$$(\text{ComSeq} \times \mathscr{G}) \times ((\text{ComSeq} \times \mathscr{G}) \cup \mathscr{G} \cup \{\text{fail}\}).$$

Here $\mathscr{G}$ is the set of all GP 2 host graphs and ComSeq is the set of command sequences as defined in the syntax (Figure 2), and fail is an element representing the program resulting in a failure state. The inference rules contain universally quantified variables, namely host graphs G and H, command sequences in ComSeq C, P, P', and Q, and rule set call R. The transitive closure of $\rightsquigarrow$ is denoted by $\rightsquigarrow^+$, and the reflexive transitive closure by $\rightsquigarrow^*$.

In general, the execution of a program on a host graph may result in another graph, fail, or diverge. Also, executions can get *stuck* in that they reach a non-terminal configuration (neither a graph nor fail) to which no inference rule is applicable. Let $\mathscr{G}$ be the set of all host graphs and $\mathscr{G}^{\oplus} = \mathscr{G} \cup \{\bot, \text{fail}\}$. These outcomes are described by the semantic function $[\![\_]\!] : \text{ComSeq} \to (\mathscr{G} \to 2^{\mathscr{G}^{\oplus}})$ which, for a command sequence P and a host graph G, is defined as

$$[\![P]\!]G = \{X \in \mathscr{G} \cup \{\text{fail}\} \mid \langle P, G\rangle \rightsquigarrow^+ X\} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}.$$

3 The Small-Step Semantics

In this section, we introduce an improved semantics defined by inference rules, and give examples of transition sequences.

Due to additional constructs, the new semantics needs to distinguish between command sequences that are valid GP 2 programs, and command sequences that are intermediary. The former are members of CommandSeq from the syntax in Figure 2, and are called *command sequences*. They have to satisfy the context conditions specified in Appendix A.6 of Bak's thesis [3]. The following condition is particularly relevant to this paper: "A `break` can only be called inside a loop. If a `break` is in the condition of a branching statement, the enclosing loop must be within the same condition." This constraint is not specific to graph programs: Java, C, and Python have similar restrictions on the use of `break` statements.

We define *extended command sequences* (set ExtComSeq) to be command sequences with additional auxiliary constructs ITE and TRY. They do not follow context conditions since we may want a `break` outside of a loop in an intermediary transition step. The ITE and TRY statements serve to advance

the command sequence in the condition in a small-step fashion, as well as to maintain the stack of host graphs. When we enter an ITE or TRY statement, the top of the stack (and current host graph) is duplicated in order to keep a backup. When exiting these statements we either pop the top, modified graph, or the second graph on the stack which is the unmodified backup copy depending on the outcome of the condition. The stack structure is needed because `if` and `try` statements may be nested. Whenever we enter an ITE or TRY construct, we push a graph, and whenever we exit one, we pop a graph. This ensures that the stack always contains enough graphs to pop.

The rules in Figure 4 inductively define a transition relation $\rightarrow$ over the following set:

$$(\text{ExtComSeq} \times \mathscr{S}) \times ((\text{ExtComSeq} \times \mathscr{S}) \cup \mathscr{S} \cup \{\text{fail}\}),$$

where $\mathscr{S}$ is the set of all stacks of GP 2 host graphs (explained below). We call an element of the set $(\text{ExtComSeq} \times \mathscr{S}) \cup \mathscr{S} \cup \{\text{fail}\}$ an *extended configuration*, whereas $(\text{CommandSeq} \times \mathscr{S}) \cup \mathscr{S} \cup \{\text{fail}\}$ is the set of *configurations*. A configuration (or extended configuration) C is *terminal* if $C = \text{fail}$ or $C = S$ for some graph stack S.

The set $\mathscr{S}$ is the set of all non-empty stacks of GP 2 host graphs where the top element is the current host graph, and where the other elements are backup copies to revert to or discard after the resolution of conditions of branching statements. Such a stack $S = [G_1, G_2, G_3, \ldots, G_n]$ is a finite ordered list of GP 2 host graphs with unary operations $\text{top}(S) = G_1$, $\text{pop}(S) = [G_2, G_3, \ldots, G_n]$ and $\text{pop2}(S) = [G_1, G_3, \ldots, G_n]$, as well as the binary operation $\text{push}(G, S) = [G, G_1, G_2, \ldots, G_n]$, where G is a GP 2 host graph.

Most of the inference rules in Figure 4 have a horizontal bar. These rules consist of a *premise* above the bar and a *conclusion* below. The conclusion defines a transition step provided that the premise holds. A rule without a bar is called an *axiom* and can be applied to a configuration without any precondition.

There are several universally quantified meta-variables within the inference rules. P, P', Q, Q', C, and C' stand for extended command sequences in ExtComSeq, S stands for a graph stack in $\mathscr{S}$, G represents a host graph, and R represents a rule set. We denote the transitive closure of $\rightarrow$ by $\rightarrow^+$, and the reflexive transitive closure by $\rightarrow^*$.

The inference rules inductively define the transition relation $\rightarrow$. The rules $[\text{call}_1]$ and $[\text{call}_2]$ are base cases. Their premises are GP 2 derivations. Which of the two premises is satisfied depends on whether $\text{top}(S) \Rightarrow_R G$ or $\text{top}(S) \not\Rightarrow_R$, i.e. whether a rule in the rule set can be applied to the current host graph or not. The `if` and `try` statements are modelled by the $[\text{if}_\text{i}]$ and $[\text{try}_\text{i}]$ rules.

Sequential composition of commands is covered by $[\text{seq}_1]$, $[\text{seq}_2]$, and $[\text{seq}_3]$, covering the cases of whether the first command called on a host graph results in a configuration, a graph stack, or fail.

Loops are semantically described as a try statement in $[\text{alap}_1]$. Calling a command sequence as long as possible is modelled by trying to apply the command sequence, and if it succeeds, keep applying it as long as possible. Breaking from a loop is handled by [break], which makes sure commands following the break are discarded, and $[\text{alap}_2]$, which terminates the loop if there is an isolated break in the TRY condition.

Figure 4a shows the inference rules for the core commands of GP 2, while Figure 4b gives the inference rules for derived commands such as `or`, `skip`, and `fail`, as well as some `if` and `try` statements with omitted `then` and `else` clauses. These commands are referred to as *derived* commands because they can be defined by the core commands [16].

Let us look at a couple of examples of transition sequences in Figure 5, the first to illustrate loops, and the second to illustrate `if` and `try` statements. For each transition, we note the applied inference

$$[\text{call}_1]\ \frac{\text{top}(S) \Rightarrow_R G}{\langle R, S\rangle \to \text{push}(G, \text{pop}(S))} \qquad [\text{call}_2]\ \frac{\text{top}(S) \not\Rightarrow_R}{\langle R, S\rangle \to \text{fail}}$$

$$[\text{seq}_1]\ \frac{\langle P, S\rangle \to \langle P', S'\rangle}{\langle P; Q, S\rangle \to \langle P'; Q, S'\rangle} \qquad [\text{seq}_2]\ \frac{\langle P, S\rangle \to S'}{\langle P; Q, S\rangle \to \langle Q, S'\rangle}$$

$$[\text{seq}_3]\ \frac{\langle P, S\rangle \to \text{fail}}{\langle P; Q, S\rangle \to \text{fail}} \qquad [\text{break}]\ \langle \texttt{break}; P, S\rangle \to \langle \texttt{break}, S\rangle$$

$$[\text{alap}_1]\ \langle P!, S\rangle \to \langle \texttt{try}\ P\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip}, S\rangle \qquad [\text{alap}_2]\ \langle \text{TRY}(\texttt{break}, P!, \texttt{skip}), S\rangle \to \text{pop2}(S)$$

$$[\text{if}_1]\ \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, S\rangle \to \langle \text{ITE}(C, P, Q), \text{push}(\text{top}(S), S)\rangle$$

$$[\text{try}_1]\ \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, S\rangle \to \langle \text{TRY}(C, P, Q), \text{push}(\text{top}(S), S)\rangle$$

$$[\text{if}_2]\ \frac{\langle C, S\rangle \to \langle C', S'\rangle}{\langle \text{ITE}(C, P, Q), S\rangle \to \langle \text{ITE}(C', P, Q), S'\rangle} \qquad [\text{try}_2]\ \frac{\langle C, S\rangle \to \langle C', S'\rangle}{\langle \text{TRY}(C, P, Q), S\rangle \to \langle \text{TRY}(C', P, Q), S'\rangle}$$

$$[\text{if}_3]\ \frac{\langle C, S\rangle \to S'}{\langle \text{ITE}(C, P, Q), S\rangle \to \langle P, \text{pop}(S')\rangle} \qquad [\text{try}_3]\ \frac{\langle C, S\rangle \to S'}{\langle \text{TRY}(C, P, Q), S\rangle \to \langle P, \text{pop2}(S')\rangle}$$

$$[\text{if}_4]\ \frac{\langle C, S\rangle \to \text{fail}}{\langle \text{ITE}(C, P, Q), S\rangle \to \langle Q, \text{pop}(S)\rangle} \qquad [\text{try}_4]\ \frac{\langle C, S\rangle \to \text{fail}}{\langle \text{TRY}(C, P, Q), S\rangle \to \langle Q, \text{pop}(S)\rangle}$$

(a) Inference rules for core commands

$$[\text{or}_1]\quad \langle P\ \texttt{or}\ Q, S\rangle \to \langle P, S\rangle \qquad [\text{or}_2]\quad \langle P\ \texttt{or}\ Q, S\rangle \to \langle Q, S\rangle$$

$$[\text{skip}]\quad \langle \texttt{skip}, S\rangle \to S \qquad [\text{fail}]\quad \langle \texttt{fail}, S\rangle \to \text{fail}$$

$$[\text{if}_5]\quad \langle \texttt{if}\ C\ \texttt{then}\ P, S\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip}, S\rangle$$

$$[\text{try}_5]\quad \langle \texttt{try}\ C\ \texttt{then}\ P, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip}, S\rangle$$

$$[\text{try}_6]\quad \langle \texttt{try}\ C\ \texttt{else}\ P, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ P, S\rangle$$

$$[\text{try}_7]\quad \langle \texttt{try}\ C, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip}, S\rangle$$

(b) Inference rules for derived commands

Figure 4: Improved GP 2 Semantics

rule as a subscript. If the conclusion of $[\text{rule}_1]$ is used as a premise for $[\text{rule}_2]$, we denote it by $\frac{[\text{rule}_2]}{[\text{rule}_1]}$.

Example 3.1. Consider the program `P=r!` and the rule `r` : $_1 \circ\!\!\rightarrow\!\!\circ \Rightarrow {}_1\circ$. Let us examine a transition sequence of `P` applied to the graph $\circ\!\!\rightarrow\!\!\circ\!\!\rightarrow\!\!\circ$, as seen in Figure 5a.

We start by applying $[\text{alap}_1]$ which turns the loop into a `try` statement. Unlike in the previous semantics, we model a loop by trying to apply its body, and if it is successful, we call the loop again.

The inference rule $[\text{try}_1]$ transforms the `try` statement into the auxiliary TRY construct, which advances the program in a small-step fashion, unlike the previous semantics. There is a similar ITE construct which models `if` statements. The top of the graph stack is duplicated since the changes made by the condition of the `try` may be discarded.

We then apply `r` to the current host graph (top of the stack) so $[\text{call}_1]$ can be applied. This serves as a premise for $[\text{try}_3]$, which ends the TRY statement, pops the second element of the stack, and moves on to the `then` part which is the original loop.

We repeat this process until `r` is no longer applicable to the host graph. At this point, $[\text{call}_2]$ serves as the premise for $[\text{try}_4]$ which exits the TRY statement. This time, the condition results in fail, so we move on to the `else` part which is `skip` and the loop terminates.

Now consider program `P'=try(if (r1;r1) then (r1;r1))` and the rule `r1` : $_1 \circ\!\!\rightarrow\!\!\circ \Rightarrow {}_1\circ$. A transition sequence of `P'` applied to host graph $\circ\!\!\leftarrow\!\!\circ\!\!\rightarrow\!\!\circ$ can be found in Figure 5b.

Since the `try` statement does not have a `then` or `else` part, we first apply $[\text{try}_7]$, which adds `skip` as both the `then` and `else` parts.

The inference rule $[\text{try}_1]$ turns the `try` statement into the auxiliary TRY statement and duplicates the top of the stack. For most of the remaining transition sequence, we apply $[\text{try}_2]$ under various premises to advance the condition.

Since the `if` has no `else` part, $[\text{if}_5]$ completes it with a `skip`. The `if` statement is then turned into the auxiliary ITE statement, duplicating the top of the stack once again.

The rule `r1` is applied to the host graph which advances the concatenation with $[\text{seq}_2]$, the ITE with $[\text{if}_2]$, and the TRY with $[\text{try}_2]$. Calling `r1` a second time resolves the ITE, and the top of the stack is popped since changes made by the conditions of `if` statements are reversed.

We keep applying the condition of the TRY, until we resolve it with $[\text{try}_3]$. This time the second graph on the stack is popped since changes made by the condition of a `try` that did not result in fail are preserved. □

4 Properties of the Semantics

In this section, we show that the semantics is non-blocking, i.e. if a transition sequence ends in an extended configuration, we can always apply an inference rule (Proposition 4.3). Note that we can only guarantee the non-blocking property for extended configuration that are part of a transition sequence originating in a valid GP 2 program. We call those *reachable* extended configurations. This is reasonable because there can be no other types of configurations in a transition sequence modelling a GP 2 program.

Furthermore, we will describe the outcomes of a transition sequence starting with a valid GP 2 program (Proposition 4.5), and show that we have finite nondeterminism (Proposition 4.7), i.e. there are only finitely many one-step transitions starting from any configuration, and what it means for the semantic function.

Let us first look at a lemma that guarantees we can make a transition step from extended configurations that do not contain a `break`, which is the first step towards showing the non-blocking property.

$\langle$ `r!`, [O→O→O]$\rangle$

$\to_{[\text{alap}_1]}$ $\langle$ `try r then r! else skip`, [O→O→O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY (`r`, `r!`, `skip`), [O→O→O, O→O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_3]}}$ $\langle$ `r!`, [O→O]$\rangle$

$\to_{[\text{alap}_1]}$ $\langle$ `try r then r! else skip`, [O→O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY (`r`, `r!`, `skip`), [O→O, O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_3]}}$ $\langle$ `r!`, [O]$\rangle$

$\to_{[\text{alap}_1]}$ $\langle$ `try r then r! else skip`, [O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY (`r`, `r!`, `skip`), [O, O]$\rangle$

$\to_{\substack{[\text{call}_2]\\[\text{try}_4]}}$ $\langle$ `skip`, [O]$\rangle$

$\to_{[\text{skip}]}$ [O]

(a) Transition sequence of program `P` applied to graph O→O→O.

$\langle$ `try(if(r1;r1) then(r1;r1))`, [O←O→O]$\rangle$

$\to_{[\text{try}_7]}$ $\langle$ `try (if (r1;r1) then (r1;r1)) then skip else skip`, [O←O→O]$\rangle$

$\to_{[\text{try}_1]}$ $\langle$TRY(`if (r1;r1) then (r1;r1)`, `skip`, `skip`), [O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{if}_5]\\[\text{try}_2]}}$ $\langle$TRY(`if (r1;r1) then (r1;r1) else skip`, `skip`, `skip`), [O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{if}_1]\\[\text{try}_2]}}$ $\langle$TRY(ITE(`r1;r1`, `r1;r1`, `skip`), `skip`, `skip`), [O←O→O, O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{seq}_2]\\[\text{if}_2]\\[\text{try}_2]}}$ $\langle$TRY(ITE(`r1`, `r1;r1`, `skip`), `skip`, `skip`), [O→O, O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{if}_3]\\[\text{try}_2]}}$ $\langle$TRY(`r1;r1`, `skip`, `skip`), [O←O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_2]}}$ $\langle$TRY(`r1`, `skip`, `skip`), [O→O, O←O→O]$\rangle$

$\to_{\substack{[\text{call}_1]\\[\text{try}_3]}}$ $\langle$ `skip`, [O]$\rangle$

$\to_{[\text{skip}]}$ [O]

(b) Transition sequence of program `P'` applied to graph O←O→O.

Figure 5: Examples of transition sequences

Lemma 4.1 (Progress from Extended Configurations)**.** Let $\langle P, S\rangle$ be an extended configuration in ExtComSeq $\times$ $\mathscr{S}$. Then one of the following applies:

- $\langle P, S\rangle \to \langle P', S'\rangle$ for some extended configuration $\langle P', S'\rangle$.
- $\langle P, S\rangle \to S'$ for some graph stack $S' \in \mathscr{S}$.
- $\langle P, S\rangle \to$ fail.
- P is not a command sequence and contains a `break`.

Proof. We shall prove this lemma by going through what P could be according to the syntax and the semantics.

Case 1: P is a rule set call. Then either $\text{top}(S) \Rightarrow_P G$ or $\text{top}(S) \not\Rightarrow_P$. So either $[\text{call}_1]$ or $[\text{call}_2]$ can be applied.

Case 2: P is a loop. If P is a loop, $[\text{alap}_1]$ can be applied.

Case 3: P is fail*,* `skip` *or an* `or` *statement.* Then [fail], [skip], or $[\text{or}_1]$ can be applied respectively.

Case 4: P is of the form `if` P_1 `then` P_2 `else` P_3 *or* `try` P_1 `then` P_2 `else` P_3. Then $[\text{if}_1]$ or $[\text{try}_1]$ can be applied. If any then-clause or else-clause is omitted as specified by the syntax, $[\text{if}_5]$, $[\text{try}_5]$, $[\text{try}_6]$, or $[\text{try}_7]$ can be applied.

Case 5: P is of the form ITE(P_1, P_2, P_3) or TRY(P_1, P_2, P_3). If P contains a `break`, the fourth point of the lemma is satisfied, as containing ITE or TRY statements makes P not a command sequence. So for the remainder of this case, assume P does not contain a `break`. If P_1 is a sequential composition, let $P_1 = P_1'; P_1''$ where P_1' is atomic. Otherwise let $P_1 = P_1'$. We shall show the lemma's statement by induction on how many ITE or TRY statements are nested in P_1' via the first atomic command of the condition.

- For the base case, assume P_1' is not an ITE or TRY statement. Then P_1' is atomic and either covered by cases 1 to 4 (P_1' cannot be `break` since P contains no `break`).
- Now for the induction step, assume that P_1' is an ITE or TRY statement. Then P_1' does derive either a configuration $\langle P_1''', S'\rangle$, a graph stack S' or fail by the induction hypothesis. Hence one of $[\text{if}_2]$, $[\text{if}_3]$, $[\text{if}_4]$, $[\text{try}_2]$, $[\text{try}_3]$, or $[\text{try}_4]$ can be applied to $\langle P, S\rangle$.

Case 6: P is a sequential composition. Then we can decompose P into $P = P_1; P_2$ where P_1 is atomic, i.e. not a sequential composition. We can apply $[\text{seq}_1]$, $[\text{seq}_2]$, or $[\text{seq}_3]$ since $\langle P_1, S\rangle \to \langle P_1', S'\rangle$, $\langle P_1, S\rangle \to S'$, or $\langle P_1, S\rangle \to$ fail respectively by cases 1 to 5. □

Lemma 4.1 has a case where the extended command sequence contains a `break`. This is because for a transition sequence not to get stuck on a `break`, we need to start with a command sequence where the `break` is within a loop, which we cannot guarantee if we consider a single transition step like in Lemma 4.1. In order to deal with this case, we prove that we can construct a transition sequence that leads to a state with no `break` in the following lemma. However, we need to restrict it to extended configurations reachable from a valid GP 2 program. We say that an extended configuration C is *reachable* if there is a configuration $\langle P, [G]\rangle$ such that $\langle P, [G]\rangle \to^* C$. This will still allow us to work towards non-blocking, since we only care about transition sequences that start with valid GP 2 programs.

Lemma 4.2 (Removing the `break` Statement)**.** Let $\langle P, S\rangle$ be an extended configuration in ExtComSeq $\times$ $\mathscr{S}$ that is reachable and non-terminal. Suppose that P contains `break`. Then one of the following applies.

- There is an extended configuration $\langle P', S'\rangle$ containing no `break` statement such that $\langle P, S\rangle \to^* \langle P', S'\rangle$.

- There is a graph stack S' such that $\langle P,S\rangle \to^+ S'$.
- $\langle P,S\rangle \to^+$ fail

Proof. First assume that $\langle P,S\rangle$ satisfies context conditions, i.e. the `break` is contained within a loop, and if the `break` is in the condition of an `if` or `try` statement, the enclosing loop must be in the same condition.

We will apply various inference rules to construct a transition sequence starting in $\langle P,S\rangle$. Remember that whenever we apply such an inference rule, it results in either a non-terminal extended configuration, a graph stack, or fail. If it results in a graph stack or fail, the second or third point of the lemma is satisfied. So at each step of the transition sequence we construct, we only need to consider the case where an inference rule results in a non-terminal extended command sequence.

If there are multiple loops with `break` statements, they are either in different sequential composition components, or nested. So let us show this lemma by induction on nesting and sequential composition.

As a base case, assume P contains a single loop with a `break`, and want to show we can apply a sequence of inference rules that ultimately removes the `break`. So P is of the form $Q_0\,;Q_1\,!\,;Q_2$, where Q_1 contains a `break`, and neither Q_0 nor Q_2 do. (What follows also applies if P is of the form $Q_1\,!\,;Q_2$, $Q_0\,;Q_1\,!$, or $Q_1\,!$.) We can repeatedly apply Lemma 4.1 to transition to $Q_1\,!\,;Q_2$. Then we apply $[\text{alap}_1]$ followed by $[\text{try}_1]$ to get $\text{TRY}(Q_1, Q_1\,!, \texttt{skip})$. We can then use Lemma 4.1 repeatedly as a premise for $[\text{try}_2]$ until we get $\text{TRY}(Q_3\,;Q_4, Q_1\,!, \texttt{skip})$, where Q_3 contains `break` and is atomic (i.e. not a sequential composition). We know Q_3 cannot be a loop since we assumed $Q_1\,!$ is the enclosing loop of the `break`. So Q_3 is either an `or`, `if`, or `try` statement. If it is an `or` statement, we can apply $[\text{or}_1]$ or $[\text{or}_2]$ to either remove the `break` or lead to $\text{TRY}(\texttt{break;}Q_5, Q_1\,!, \texttt{skip})$. Similarly, if Q_3 is an `if` or `try` statement, the `break` must be in the `then` or `else` part due to context conditions, and we can use inference rules to either remove the `break` or lead to $\text{TRY}(\texttt{break;}Q_5, Q_1\,!, \texttt{skip})$. We can now apply $[\text{try}_2]$ under the premise of [break] to get $\text{TRY}(\texttt{break}, Q_1\,!, \texttt{skip})$. To this, we can apply $[\text{alap}_2]$, which gets rid of the `break`.

For the induction step, let us first consider the case of nesting. Assume that P is of the form $Q_0;(Q_1;Q_2;Q_3)!;\ Q_4$, where Q_2 satisfies the lemma statement, and either Q_1 or Q_2 contain a single `break`. We can use the same arguments as in the base case in addition to $[\text{seq}_1]$ under the premise of the induction hypothesis to get rid of the `break`.

Now consider sequential composition. As an induction step, assume that P is of the form $Q_0;Q_1\,!\,;Q_2;$ $Q_3\,!\,;Q_4$, where one of Q_1 or Q_3 satisfies the lemma statement, and the other contains a single `break`. Again, we can use the arguments from the base case as well as the induction hypothesis in conjunction with $[\text{seq}_1]$ to remove the `break`.

Finally, assume that $\langle P,S\rangle$ does not satisfy context conditions, i.e. `break` statements must appear within a loop, and if one appears in the condition of a branching statement, the enclosing loop must be within the same condition. Since $\langle P,S\rangle$ is reachable, the latter condition is verified since the transition relation preserves it (loops can only be removed by inference rules, they cannot be "moved"). Suppose there is a `break` without an enclosing loop. This must be because $[\text{alap}_1]$ is applied earlier in the transition sequence, so it must be within the condition of a `try` or TRY. So we can use the same arguments as earlier in the proof, except that we need not argue that some of the inference rule, such as $[\text{alap}_1]$ or $[\text{try}_1]$ need to be applied. □

Now that we have Lemmata 4.1 and 4.2, we can prove that the non-blocking property holds.

Proposition 4.3 (Non-Blocking Property)**.** Let $\langle P,S\rangle$ be an extended configuration that is reachable and non-terminal. Then there is a transition step $\langle P,S\rangle \to C$ for some extended configuration C.

Proof. If P does not contain a `break`, this proposition follows from Lemma 4.1. Otherwise, it follows from Lemma 4.2. □

Let us now introduce a lemma that makes various statements about the size of host graph stacks in order to ensure that the inference rules are well-defined. Since we defined stacks to be nonempty, we want to make sure that if a transition sequence starts with a nonempty stack, it cannot lead to an empty stack, which the following lemma shows. Furthermore, when a transition sequence terminates in a graph stack, we want that stack to only contain one host graph.

For this lemma, we want to start from a valid GP 2 program, not extended command sequences in general (since they may contain auxiliary constructs like ITE and TRY). So we consider configurations in CommandSeq $\times \mathscr{S}$. These follow the context conditions on where the `break` statement can appear as specified in [3].

Lemma 4.4 (Stack Size)**.** Let $\langle P, [G] \rangle$ be a configuration in CommandSeq $\times \mathscr{S}$.

(a) If $\langle P, [G] \rangle \to^* \langle P', S \rangle$, where $\langle P', S \rangle$ is an extended configuration, then $|S| \geq 1$.

(b) If $\langle P, [G] \rangle \to^+ S$, where S is a graph stack, then $|S| = 1$.

Proof. The statement in (a), is satisfied for zero transition steps. So let us examine the inference rules that contain push, pop, and pop2. The rule $[\text{call}_1]$ contains both push and pop, but preserves the size of the stack. The rules that push a graph onto the stack are $[\text{if}_1]$ and $[\text{try}_1]$ which are exactly the rules that introduce an ITE or a TRY. The rules that pop a graph from the stack are $[\text{alap}_2]$, $[\text{if}_3]$, $[\text{if}_4]$, $[\text{try}_3]$, and $[\text{try}_4]$. These are exactly rules that remove an ITE or TRY from the extended command sequence. Since $\langle P, [G] \rangle$ contains no ITE or TRY statements and only one host graph, we have $|S| = \#(P') + 1$, where # counts the combined number of ITE and TRY statements in an extended command sequence. Since $|S| = \#(P') + 1$, we have $|S| \geq 1$.

Now in case (b), we can break down the transition sequence into $\langle P, [G] \rangle \to^* \langle P', S' \rangle \to S$. Like in the proof of (a), the formula $|S'| = \#(P') + 1$ applies. Let us examine which inference rules can be applied in the final step of the transition. It can only be either [skip], $[\text{call}_1]$, or $[\text{alap}_2]$. To apply [skip], P' must be `skip` and $\#(\texttt{skip}) = 0$, so $|S| = |S'| = 1$. To apply $[\text{call}_1]$, P' must be rule set call, and hence cannot contain ITE or TRY, so $|S| = |S'| = 1$. To apply $[\text{alap}_2]$, P' must be of the form $\text{TRY}(\texttt{break}, P''!, \texttt{skip})$, where P'' is an extended command sequence. We know P'' cannot contain an ITE or TRY statement because they can only be nested in their first argument. Indeed, if an extended command sequence already starts with an ITE or TRY, no inference rule allows for said ITE or TRY statement to be nested within another one. So the only way to nest statements is via the rule $[\text{try}_2]$, which modifies the first argument. But the first argument of P' is `break`, which contains no ITE or TRY statements. So $\#(P') = 1$ and $|S'| = 2$. Since we apply $[\text{alap}_2]$, we have $S = \text{pop2}(S')$, so $|S| = |S'| - 1 = 1$. □

We also want to make sure that if we call pop2 on a stack to pop its second element, the stack does indeed contain at least two elements. More precisely, under the premise of Lemma 4.4, if $\langle P, [G] \rangle \to^+ \langle P', \text{pop2}(S) \rangle$ (an extended configuration) or $\langle P, [G] \rangle \to^+ \text{pop2}(S)$ (a graph stack), then $|S| \geq 2$. This follows directly from Lemma 4.4 since $|\text{pop2}(S)| = |S| - 1$.

Let us now use Lemmata 4.1, 4.2, and 4.4 is to describe what the possible outcomes of a transition sequence starting in a valid GP 2 program are.

Proposition 4.5 (Outcomes of Transition Sequences)**.** Let $\langle P, [G] \rangle$ be a configuration in CommandSeq $\times$ $\mathscr{S}$. Then one of the following applies:

- There is an infinite transition sequence $\langle P, [G]\rangle \to \langle P_1, S_1\rangle \to \langle P_2, S_2\rangle \to \dots$ where $\langle P_i, S_i\rangle$ is an extended configuration for all $i \geq 1$.
- $\langle P, [G]\rangle \to^+ [G']$ for some host graph G'.
- $\langle P, [G]\rangle \to^+$ fail.

Proof. Lemma 4.4 guarantees that if a transition sequence starts in $\langle P, [G]\rangle$ and ends in a stack, that stack only contains one graph. So for this proposition, it is enough to show that transition sequences end in a stack in the relevant cases.

In order to get rid of a potential `break` statement in P, we can apply Lemma 4.2 to $\langle P, [G]\rangle$. If we get a graph stack or fail, we fulfil the second or third case of this proposition. Otherwise we get an extended configuration $\langle P', S\rangle$ that contains no `break`.

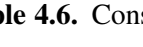
Since there is now no `break` in either $\langle P, [G]\rangle$ or $\langle P', S\rangle$, we can apply the first, second, and third cases of Lemma 4.1 either indefinitely to get an infinite transition sequence, or until we get a graph stack or fail. □

Now that we know the possible outcomes of a transition sequence, we can define the *semantic function* $[\![\_]\!] : \text{CommandSeq} \to (\mathscr{G} \to 2^{\mathscr{G}^\oplus})$, where $\mathscr{G}$ is the set of host graphs, $[\mathscr{G}]$ the set of stacks consisting of exactly one host graph (which we can identify with single host graphs), and $\mathscr{G}^\oplus = [\mathscr{G}] \cup \{\text{fail}, \bot\}$. The symbol $\bot$ is used to represent an infinite transition sequence, i.e. divergence. The function is defined as

$$[\![P]\!]G = \{X \in [\mathscr{G}] \cup \{\text{fail}\} \mid \langle P, G\rangle \to^+ X\} \cup \{\bot \mid P \text{ can diverge from } G\}.$$

This functions differs from the one presented in [17] and Section 2 since $\bot$ is only used when P diverges, not when P gets stuck, i.e. there is a non-terminal configuration.

Let us now examine the property of *finite nondeterminism* as specified by Apt in Section 4.1 of [1], i.e. the set of elements reachable from a configuration in one transition step is finite. A related concept is *bounded nondeterminism*, where the cardinality of the aforementioned set depends on the program only (and not on the size of the input). GP 2 has finite nondeterminism, but not bounded nondeterminism, which the following example illustrates. It also shows that bounded nondeterminism is a stronger property than finite nondeterminism.

Example 4.6. Consider the rule `r` :  and the *comb graph* G_4 as shown in Figure 6. There

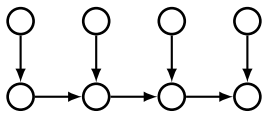


Figure 6: The comb graph G_4

are four possible matches for the left-hand side of rule `r` in graph G_4, so applying the rule can result in four different non-isomorphic graphs, which is a finite amount. When applying `r` to comb graph G_k, we get k non-isomorphic graphs, which depends on the size of the host graph and hence is not bounded. □

Proposition 4.7 (Finite Nondeterminism)**.** Let $\gamma \in \text{ExtComSeq} \times \mathscr{S}$ be an extended configuration, and $T_\gamma = \{\gamma' \mid \gamma \to \gamma' \in \text{ExtCommSeq} \times \mathscr{S}\}$. Then $|T_\gamma|$ is finite.

Proof. The only inference rules that cause nondeterminism are $[\text{or}_1]$, $[\text{or}_2]$, and $[\text{call}_1]$. If the rules $[\text{or}_1]$ and $[\text{or}_2]$ are applicable to γ then there are exactly two configurations reachable from γ. In $[\text{call}_1]$, the nondeterminism comes from several GP 2 rules being called non-deterministically as part of a rule set, as well as from all the ways these rules can be matched in the host graph. Since rule sets and host graphs are finite, the number of configurations reachable from γ in one step via the inference rule $[\text{call}_1]$ is finite as well. □

Reynolds [18] defines this kind of nondeterminism using the semantic function instead of the set of configurations reachable in one step. The following corollary shows that this semantics fulfils that definition as well.

Corollary 4.8. Let $P \in \text{CommandSeq}$ and $G \in \mathscr{G}$ such that $[\![P]\!]G$ is infinite. Then $\bot \in [\![P]\!]G$.

Proof. Let $\gamma_0 = \langle P, [G] \rangle$. Then $T^*_{\gamma_0} = \{\gamma \mid \gamma_0 \to^* \gamma \in \text{ExtCommSeq} \times \mathscr{S}\}$ is infinite as well since it contains all elements of $[\![P]\!]G$ except perhaps fail or $\bot$. The set $T^*_{\gamma_0}$ can be seen as a tree whose nodes are configurations and whose edges are defined by transition relations. Since T_γ is finite for all configurations γ by Proposition 4.7, each node in the tree only has finitely many adjacent nodes. By König's Lemma [12], the tree contains an infinite path. Since every node of the tree is reachable from the root γ_0, there is an infinite path starting from γ_0. By definition of the tree, this means there's an infinite transition sequence starting with γ_0. By definition of the semantic function, we can conclude that $\bot \in [\![P]\!]G$. □

Many references [8, 10, 18, 19] equate the concepts of finite and bounded nondeterminism and call it "bounded nondeterminism". This is likely because imperative programming languages, unlike GP 2, usually satisfy both properties, so there is no need to distinguish between them.

5 Conclusion

We have introduced a new small-step operational semantics for the graph programming language GP 2. Unlike the previous semantics, this one is entirely small-step and non-blocking. As a consequence, it accurately models diverging computations. In particular, the new semantic function correctly lists $\bot$ as an outcome when there is a computation in which the condition of a branching statement or the body of a loop diverges. We also obtain finite nondeterminism, meaning that for every configuration there are only finitely many choices for the next transition step.

In future work, the new semantics should serve as a solid underpinning for setting up a time and space complexity theory for GP 2. Its small-step nature is crucial to defining atomic computation steps. Such a theory could possibly be automated akin to the resource analysis in [13].

Another aspect of the GP 2 semantics is that it is orthogonal to the definition of the transformation rules. The inference rules that depend on the domain of graph transformation only need the definition of a rule application ($[\text{call}_1]$) and the information when such an application fails ($[\text{call}_2]$). Hence this semantics could be used as a foundation for GP 2-like programming languages over other rule-based domains, such as string rewriting [5] or term rewriting [2].

References

[1] Krzysztof R. Apt (1984): *Ten Years of Hoare's Logic: A Survey. Part II: Nondeterminism*. *Theoretical Computer Science* 28, pp. 83–109, doi:10.1016/0304-3975(83)90066-X.

[2] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1017/cbo9781139172752.

[3] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, University of York, UK. Available at `https://etheses.whiterose.ac.uk/12586/`.

[4] Christopher Bak & Detlef Plump (2016): *Compiling Graph Programs to C*. In: *Proc. International Conference on Graph Transformation (ICGT 2016)*, *LNCS* 9761, Springer, pp. 102–117, doi:10.1007/978-3-319-40530-8_7.

[5] Ronald V. Book & Friedrich Otto (1993): *String-Rewriting Systems*. Springer, doi:10.1007/978-1-4613-9771-7.

[6] Graham Campbell, Brian Courtehoute & Detlef Plump (2020): *Fast Rule-Based Graph Programs*. *ArXiv e-prints* arXiv:2012.11394. Available at `https://arxiv.org/abs/2012.11394`. 47 pages.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, 3 edition. The MIT Press.

[8] Edsger W. Dijkstra (1997): *A Discipline of Programming*, 1st edition. Prentice Hall PTR.

[9] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer, doi:10.1007/3-540-31188-2˙3.

[10] Wan Fokkink & Thuy Vu (2003): *Structural Operational Semantics and Bounded Nondeterminism*. *Acta Informatica* 39, pp. 501–516, doi:10.1007/s00236-003-0111-1.

[11] Ivaylo Hristakiev & Detlef Plump (2016): *Attributed Graph Transformation via Rule Schemata: Church-Rosser Theorem*. In: *Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers*, 9946, Springer, pp. 145–160, doi:10.1007/978-3-319-50230-4˙11.

[12] Dénes König (1927): *Über eine Schlussweise aus dem Endlichen ins Unendliche*. *Acta Sci. Math.(Szeged)* 3(2-3), pp. 121–130.

[13] Georg Moser & Manuel Schneckenreither (2020): *Automated amortised resource analysis for term rewrite systems*. *Science of Computer Programming* 185, p. 102306, doi:10.1016/j.scico.2019.102306.

[14] Hanne Riis Nielson & Flemming Nielson (2007): *Semantics with Applications: An Appetizer*. Springer, doi:10.1007/978-1-84628-692-6.

[15] Gordon D. Plotkin (2004): *A Structural Approach to Operational Semantics*. *Journal of Logic and Algebraic Programming* 60–61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001.

[16] Detlef Plump (2012): *The Design of GP 2*. In: *Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, *Electronic Proceedings in Theoretical Computer Science* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[17] Detlef Plump (2017): *From Imperative to Rule-based Graph Programs*. *Journal of Logical and Algebraic Methods in Programming* 88, pp. 154–173, doi:10.1016/j.jlamp.2016.12.001.

[18] John C Reynolds (1998): *Theories of Programming Languages*. Cambridge University Press, doi:10.1017/cbo9780511626364.

[19] H. Søndergaard & P. Sestoft (1992): *Non-determinism in Functional Languages*. *The Computer Journal* 35(5), pp. 514–523, doi:10.1093/comjnl/35.5.514.