Expressiveness and Complexity of Graph Logic

Anuj Dawar
University of Cambridge Computer Lab., Cambridge, UK.
anuj.dawar@cl.cam.ac.uk

Philippa Gardner
Department of Computing, Imperial College, London, UK.
pg@doc.ic.ac.uk

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
ghelli@di.unipi.it

May 29, 2006

Abstract

We investigate the complexity and expressive power of a spatial logic for reasoning about graphs. This logic was previously introduced by Cardelli, Gardner and Ghelli, and provides the simplest setting in which to explore such results for spatial logics. We study several forms of the logic: the logic with and without recursion, and with either an exponential or a linear version of the basic composition operator. We study the combined complexity and the expressive power of the four combinations. We prove that, without recursion, the linear and exponential versions of the logic correspond to significant fragments of First-Order (FO) and Monadic Second-Order (MSO) Logics; the two versions are actually equivalent to FO and MSO on graphs representing strings. However, when the two versions are enriched with μ-style recursion, their expressive power is sharply increased. Both are able to express PSPACE-complete problems, although their combined complexity and data complexity still belong to PSPACE.

1 Introduction

Spatial logics have been used to specify properties of graphs, memory heaps, and trees. They are characterized by the presence of the composition operator (sometimes called separation operator) for reasoning locally about disjoint substructures. Such logics have been used to reason about code with side effects [21], mobile processes [8], and semi-structured data [7, 6]. Our motivation for studying expressivity results arose from two pieces of work: Cardelli, Gardner and Ghelli's work [6] on using a spatial logic for querying graphs, which postulates the connections with MSO and refers to some of the results reported here, and Cardelli and Ghelli's work [8] on the TQL language for querying XML data which, although it has been implemented and tested [11], had no formal analysis of its expressive power until now. In this paper, we concentrate on the spatial logic for reasoning about graphs, since this is the simplest setting in which to study the combination of first-order logic connectives with spatial connectives. In Section 2, we describe how our results transfer to the TQL project.

The complexity and expressive power of a query language based on a logic L are typically characterized by the following problems:

- the *combined complexity* of the model-checking problem of the underlying logic L: that is, the complexity class that contains the set $\text{Truths}_L = \{(G, \phi) \ : \ G \vDash \phi\}$;
- the *expressivity* of L: that is, the collection of all sets $\mathcal{G}_\phi = \{G \ : \ G \vDash \phi\}$, for each sentence ϕ in L;
- the *data complexity* of model-checking L, that is, the complexity class that contains all the sets $\mathcal{G}_\phi$.

Model-checking is directly related to n-ary query-answering: that is, given G and ϕ with n free variables, finding all substitutions σ such that $G \vDash \phi\sigma$ holds. By enumerating all substitutions over the domain of G, query-answering can be reduced to model-checking modulo a polynomial factor (for n fixed). Moreover, the evaluation of Boolean queries coincides with model-checking. Hence, the combined complexity of model-checking is a measure of query evaluation cost. However, the data complexity problem recognizes the fact that databases are usually much bigger than queries. It therefore makes sense to study how the complexity of $G \vDash \phi$ depends on G when ϕ is fixed. Moreover, data complexity is an upper bound for the set of the sets of graphs $\mathcal{G}_\phi$ that can be expressed in the logic, and hence is a way to describe its expressive power.

In this paper, we explore expressivity and complexity results for four versions of the spatial logic for reasoning about graphs: the basic logic GL without recursion, the logic GL_μ with least fixed-points for positive formulae studied in [6], and the 'linear' sublogics LGL and LGL_μ with a limited form of the composition operator which we introduce below. We show that GL and GL_μ have a high combined complexity and expressive power, both in the realm of PSPACE. This high complexity derives from the fact that the composition operator $\phi|\psi$ specifies that a graph can be split into two parts: the part satisfying ϕ and the part satisfying ψ. It therefore represents an existential quantification over the 2^n subgraphs of a graph of size n. Ghelli's experience with TQL, however, shows that in practice the vast majority of queries use a very limited 'linear' version of the composition operator, denoted by $\phi\rfloor\psi$, where the quantification only ranges over the n one-edge subgraphs of the graph. Indeed, the TQL implementation supports the $\rfloor$ operator as primitive, and a core task of the TQL optimizer is to transform every instance of the full composition into the linear composition, where possible. Full composition rarely survives the optimization and, when it does, the query produced is often unusable even on very small databases. For this reason, the TQL designers seriously considered the possibility of replacing $|$ by $\rfloor$ altogether. The logics LGL and LGL_μ use this linear composition. We found that the analysis of their complexity and expressive power is not only interesting in the context of the TQL project, but also for the strict relationship of LGL with first-order logic; this relationship is discussed later.

Our main technical contributions are the following.

- **Relation between GL and MSO** We provide a translation of GL into monadic second-order logic for graphs (MSO), which illustrates that GL is intermediate between first-order logic (FO) and MSO for graphs. We provide examples of graph properties which are well-known to be expressible in MSO, which are not expressible in FO, and which we show are expressible in GL. Despite this, we conjecture that GL is strictly less expressive than MSO, as it seems unlikely that one could express graph properties such as Hamiltonicity and 3-colourability in GL. Interestingly, we are able to show that GL is as expressive as MSO on strings over a finite alphabet. We establish this result using a translation of regular expressions into GL, with the translation of the Kleene star operation requiring a creative use of the composition operator.
- **Complexity of GL** The translation of GL into MSO establishes that $\text{Truths}_{\text{GL}}$ is in PSPACE. We also establish PSPACE-completeness by a reduction from the validity of quantified Boolean formulas. The result is not surprising, since we have already established that GL lies between FO and MSO which are both PSPACE-complete; we include it for completeness.

- **Expressivity of GL** A simple consequence of the translation of GL to MSO is that, for every $\phi \in$ GL, the set of graphs $\mathcal{G}_\phi$ is in the polynomial hierarchy. This does not imply that *all* problems in the polynomial hierarchy can be expressed in GL. We show that this expressivity upper-bound is tight, in that GL can express complete problems at all levels of the hierarchy. This is achieved by encoding quantified Boolean formulas as graphs.

- **Complexity and expressivity of GL$_\mu$** We show that the combined complexity of GL$_\mu$ is also in PSPACE. The upper bound is established by studying the complexity of an abstract version of the algorithm used in the TQL system, which is similar to Winskel's algorithm for model-checking the μ-calculus [27].

  We also compare the expressive power of GL$_\mu$ to that of MSO. The former can define properties that are not definable in MSO such as the parity of the set of edges. We also show that GL$_\mu$ can express a problem that is PSPACE-complete, and therefore unlikely to be expressible in second-order logic (unless PSPACE collapses to the polynomial hierarchy), let alone MSO. The ability to express a PSPACE-complete problem also gives us a very good characterization of the expressive power of GL$_\mu$, because of the PSPACE upper bound. The proof is again through an encoding of quantified Boolean formulas as graphs. As far as we are aware, this is the first result about the expressive power of a spatial logic with recursion. On string graphs, GL$_\mu$ is at least as expressive as conjunctive context-free grammars, as defined in [22].

- **Complexity and expressivity of LGL** We show how the translation of GL into MSO can be modified to produce an encoding of LGL into FO. We show that the combined complexity of this logic is still PSPACE-complete. The expressive power is however much lower, being included in LogSpace. On string graphs, the expressive power corresponds precisely to that of FO, since LGL can express all and only the star-free languages.

- **Complexity and expressivity of LGL$_\mu$** Whilst LGL can be mapped to FO, we show that LGL$_\mu$ is very different from FO+LFP. In particular, LGL$_\mu$ can specify evenness, which is the standard example of a graph property which is not expressible in FO+LFP. We also show that LGL$_\mu$ goes well beyond the PTime limit of FO+LFP, by exhibiting a sentence that expresses a PSPACE-complete property. On string graphs, LGL$_\mu$ can express at least every conjunctive linear context-free language, which is much richer than the class of regular languages.

Many of our results are obtained by adapting standard techniques, though not always in a straightforward fashion. However, we found some results to be of particular interest and quite surprising: the expressive power of GL is the same as MSO when we only consider strings; the expressive power of GL$_\mu$ (and even of LGL$_\mu$) goes considerably beyond that of MSO, and the expressive power of LGL$_\mu$ goes beyond that of FO+LFP.

2 Graph Logic and Models

2.1 Graphs

In [6], graphs are defined as the denotations of terms of the grammar $G ::= \mathbf{0} \mid \mathbf{a}(\mathbf{x}_1, \mathbf{x}_2) \mid G \,|\, G$. The term $\mathbf{0}$ denotes the empty graph, term $\mathbf{a}(\mathbf{x}_1, \mathbf{x}_2)$ denotes the graph consisting of just one edge from node named $\mathbf{x}_1$ to node $\mathbf{x}_2$ with label $\mathbf{a}$, and the composition $G \mid G'$ denotes the disjoint union of the edges of G and G' and the set-union of their nodes. Names $\mathbf{x}$ and labels $\mathbf{a}$ are *observable* in the logic, which means that two graphs $\mathbf{a}_1(\mathbf{x}_1, \mathbf{x}_1)$ and $\mathbf{a}_2(\mathbf{x}_2, \mathbf{x}_2)$ can be distinguished by logical formulas, unless $\mathbf{a}_1 = \mathbf{a}_2$ and $\mathbf{x}_1 = \mathbf{x}_2$. While a node label $\mathbf{x}$ uniquely identifies a node, edge labels do not identify

edges, even on a specific pair of nodes, so that a graph $\mathbf{a}(\mathbf{x}_1, \mathbf{x}_2)|\mathbf{a}(\mathbf{x}_1, \mathbf{x}_2)$ contains *two* edges, with the same label, source, and destination. In the logic we can observe that the graph contains two edges, and we can observe their label, source, and destination, but we have no way to observe the edge itself, i.e. to distinguish the first edge from the second. Isolated nodes are not specified in the graph syntax This style of presentation is not standard in graph theory[1], but it arose from the field of process algebra, and is reflected in the logic.

In the same paper [6], an equivalent presentation of graphs as relational structures is given. We present here a streamlined version of these structures. We fix an infinite set of node names $\mathcal{X}$ and an infinite set of edge labels $\mathcal{A}$. A graph is then defined as a pair $G = (E, \mathsf{edge})$ where E is a *finite* set of edges and $\mathsf{edge} : E \to \mathcal{A} \times \mathcal{X} \times \mathcal{X}$ is a function that associates to each edge a label, a source node and a target node. Two graphs $G = (E, \mathsf{edge})$ and $G' = (E', \mathsf{edge'})$ are *graph-equal*, denoted $G \equiv G'$, whenever there exists a bijection $\sigma : E \to E'$ such that $\mathsf{edge} = \mathsf{edge'} \circ \sigma$. Notice that the ranges of edge and $\mathsf{edge'}$ are required to be identical in this definition, reflecting the fact that the node names and edge labels can be observed.[2] In informal reasoning, we will often identify two graphs which are graph-equal.

We introduce some notation. The set of nodes of a graph is given by

$$\mathrm{nam}(G) =_{def} \Pi_2(\mathrm{range}(\mathsf{edge})) \cup \Pi_3(\mathrm{range}(\mathsf{edge}))$$

and the set of labels by

$$\mathrm{lab}(G) =_{def} \Pi_1(\mathrm{range}(\mathsf{edge})),$$

where Π_i denotes the ith projection. As mentioned above, the graph composition $G_1 \mid G_2$ is the disjoint union of the edges and the set-union of the nodes: that is,

$$(E_1, \mathsf{edge}_1) \mid (E_2, \mathsf{edge}_2) =_{def} (E'_1 \uplus E'_2, \mathsf{edge}'_1 \uplus \mathsf{edge}'_2),$$

where (E'_1, edge'_1) and (E'_2, edge'_2) are two arbitrary graphs such that $(E'_1, \mathsf{edge}'_1) \equiv (E_1, \mathsf{edge}_1)$, $(E'_2, \mathsf{edge}'_2) \equiv (E_2, \mathsf{edge}_2)$ and $E'_1 \cap E'_2 = \emptyset$. We use $\uplus$ to denote the union of two functions with disjoint domain, and the union of two disjoint sets. This definition of graph composition is well-defined up to graph equality.

A subgraph is, essentially, a subset of the edges of a graph. Formally, a graph (E', edge') is a subgraph of (E, edge) if and only if, for some graph (E_1, edge_1) which is graph-equal to (E, edge), $E' \subseteq E_1$ and edge' is edge_1 restricted to E'.

2.2 The Logics GL and GL$_\mu$

In this section, we define the μ-Recursive Graph Logic GL$_\mu$. The basic Graph Logic GL is the obvious fragment of GL$_\mu$ without the recursion variables and recursion operator. GL$_\mu$ extends first-order logic with a composition operator and μ-recursion.

[1] According to standard terminology, every triple $\mathbf{a}(\mathbf{x}_1, \mathbf{x}_2)$ is actually a labeled three-vertex directed hyperedge, hence our "graphs" would be described as vertex-labeled 3-uniform hypergraphs, or, more precisely, multi-directed-hypergraphs. Each hypergraph is also bipartite, since the set of vertices is partitioned into those labelled in $\mathcal{A}$ and those labelled in $\mathcal{X}$, and every directed hyperedge is associated with a triple of vertices labelled in $\mathcal{A} \times \mathcal{X} \times \mathcal{X}$. We prefer to call these structures just *graphs*, and to define them from the ground up, with no reference to standard terminology.

[2] In [6], we consider a name hiding operator on graphs. This means that the graphs are defined using vertices and an explicit function src mapping names to some of the vertices in the graph. All the results in this paper easily apply to this more complex setting. It is only the translation of Graph Logic to monadic second-order logic presented in Section 3.1 that requires some modification to cope with hiding, as we shall observe.

Definition 2.1 *The GL_μ-formulas are built from a set $\mathcal{X}$ of node names, a set $\mathcal{A}$ of label names, and the countably infinite sets $V_\mathcal{X}$, $V_\mathcal{A}$, and $V_\mathcal{R}$, of node variables, label variables, and recursive formula variables respectively. A formula is one of*

$\mathsf{0}$	
T	
$\alpha(\xi_1, \xi_2)$	*where* $\alpha \in \mathcal{A} \cup V_\mathcal{A}$ *and* $\xi_i \in \mathcal{X} \cup V_\mathcal{X}$
$\xi_1 = \xi_2$ *or* $\alpha_1 = \alpha_2$	*where* $\alpha_i \in \mathcal{A} \cup V_\mathcal{A}$ *and* $\xi_i \in \mathcal{X} \cup V_\mathcal{X}$
$\phi \mid \psi$ *or* $\phi \wedge \psi$ *or* $\neg\phi$	*where* ϕ *and* ψ *are formulas*
$\exists x.\phi$ *or* $\exists a.\phi$	*where* $x \in V_\mathcal{X}, a \in V_\mathcal{A}$ *and* ϕ *is a formula*
R	*where* $R \in V_\mathcal{R}$
$\mu R.\phi$	*where* $R \in V_\mathcal{R}, \phi$ *is a formula, and* R *only appears positively in* ϕ

As usual, a variable occurs positively in a formula iff it only appears under an even number of negations.

We use bold face for constants ($\mathbf{a}$, $\mathbf{x}$, ...), italics for variables (a, x, ...), and Greek letters for terms (α, ξ...) which are either constants or variables. The sets of the free and bound variables of a formula, fv(ϕ) and bv(ϕ), are defined as usual. We use nam(ϕ) to denote the set of all node-name constants $\mathbf{x}_i$ that appear in ϕ, and lab(ϕ) to denote the label-name constants $\mathbf{a}_i$. For example, if $\phi = \exists x.\mathbf{a}(x, y) | a(\mathbf{x}, \mathbf{y})$, then fv($\phi$) $= \{y, a\}$, bv(ϕ) $= \{x\}$, nam(ϕ) $= \{\mathbf{x}, \mathbf{y}\}$, and lab($\phi$) $= \{\mathbf{a}\}$. We use nam(G, ϕ) to denote the union of the name constants in G and in ϕ; similarly, lab(G, ϕ) denotes the label constants in G plus those in ϕ.

To define the semantics, let $\mathcal{G}$ denote the set of all graphs over $\mathcal{X}$, $\mathcal{A}$. Let ϕ be a formula and $\sigma : V_\mathcal{X} \cup V_\mathcal{A} \to \mathcal{X} \cup \mathcal{A}$ be an assignment of node and label names to the node and label variables respectively. Extend σ in a canonical fashion to the domain $V_\mathcal{X} \cup V_\mathcal{A} \cup \mathcal{X} \cup \mathcal{A}$, by letting $\sigma(\mathbf{z}) = \mathbf{z}$ for $\mathbf{z} \in \mathcal{X} \cup \mathcal{A}$. Finally, let ρ map recursion variables to elements of $\mathcal{P}(\mathcal{G})$. We now define $[\![\phi]\!]_{\sigma;\rho}$, the set of graphs satisfying ϕ under the assignments σ and ρ, as follows:

$$
\begin{array}{lcl}
[\![\mathsf{0}]\!]_{\sigma;\rho} & = & \{(\emptyset, \emptyset)\} \text{ (the singleton containing the empty graph)} \\
[\![\mathsf{T}]\!]_{\sigma;\rho} & = & \mathcal{G} \\
[\![\alpha(\xi_1, \xi_2)]\!]_{\sigma;\rho} & = & \{G \; : \; G \equiv (\{\mathbf{e}\}, \mathbf{e} \mapsto (\sigma\alpha, \sigma\xi_1, \sigma\xi_2))\} \\
[\![\xi_1 = \xi_2]\!]_{\sigma;\rho} & = & \mathcal{G}, \text{ if } \sigma\xi_1 = \sigma\xi_2; \; \emptyset \text{ otherwise} \\
[\![\alpha_1 = \alpha_2]\!]_{\sigma;\rho} & = & \mathcal{G}, \text{ if } \sigma\alpha_1 = \sigma\alpha_2; \; \emptyset \text{ otherwise} \\
[\![\phi \mid \psi]\!]_{\sigma;\rho} & = & \{G \; : \; G \equiv (G_1 \mid G_2) \text{ for some } G_1 \in [\![\phi]\!]_{\sigma;\rho} \text{ and } G_2 \in [\![\psi]\!]_{\sigma;\rho}\} \\
[\![\phi \wedge \psi]\!]_{\sigma;\rho} & = & [\![\phi]\!]_{\sigma;\rho} \cap [\![\psi]\!]_{\sigma;\rho} \\
[\![\neg\phi]\!]_{\sigma;\rho} & = & \mathcal{G} \setminus [\![\phi]\!]_{\sigma;\rho} \\
[\![\exists x. \, \phi]\!]_{\sigma;\rho} & = & \bigcup_{\mathbf{x} \in \mathcal{X}} [\![\phi]\!]_{\sigma, x \mapsto \mathbf{x}; \rho} \\
[\![\exists a. \, \phi]\!]_{\sigma;\rho} & = & \bigcup_{\mathbf{a} \in \mathcal{A}} [\![\phi]\!]_{\sigma, a \mapsto \mathbf{a}; \rho} \\
[\![R]\!]_{\sigma;\rho} & = & R\rho \\
[\![\mu R.\phi]\!]_{\sigma;\rho} & = & \bigcap\{S \in \mathcal{P}(\mathcal{G}) \; : \; [\![\phi]\!]_{\sigma;\rho, R \mapsto S} \subseteq S\}
\end{array}
$$

This definition can be shown to be well-defined by structural induction on formulas. For the recursive case, observe that the set $\mathcal{P}(\mathcal{G})$ is a complete lattice, and that $\lambda S. \, [\![\phi]\!]_{\sigma;\rho,R \mapsto S}$ is a monotone mapping. As a conquence, the intersection of the pre-fixpoints of the satisfaction interpretation is its least fixpoint (Lemma 2.2).

We say that a formula is μ-closed if it contains no free recursion variable. For a μ-closed formula, $[\![\phi]\!]_{\sigma;\rho}$ does not depend on ρ. Hence, for such formulas, we use $[\![\phi]\!]_\sigma$ as an abbreviation for $[\![\phi]\!]_{\sigma;()}$, and observe that it is also equal to $[\![\phi]\!]_{\sigma;\rho}$ for any other ρ. We use the notation $G \vDash^{\sigma;\rho} \phi$ to denote that $G \in [\![\phi]\!]_{\sigma;\rho}$, and, for μ-closed formulas, write $G \vDash^\sigma \phi$ to denote $G \in [\![\phi]\!]_\sigma$; we call $G \vDash^\sigma \phi$ the

satisfaction relation of GL_μ. When ϕ is a sentence, its interpretation does not depend on either σ or ρ, and in this case we write $G \vDash \phi$.

Lemma 2.2 *For μ-closed formulas, the satisfaction relation $G \vDash^\sigma \phi$ satisfies the properties:*

$$
\begin{array}{lcl}
\mathbf{0} \vDash^\sigma G & \Leftrightarrow & G = (\emptyset, \emptyset) \\
G \vDash^\sigma \mathsf{T} & & \\
G \vDash^\sigma \alpha(\xi_1, \xi_2) & \Leftrightarrow & G \equiv (\{\mathbf{e}\}, \mathbf{e} \mapsto (\sigma\alpha, \sigma\xi_1, \sigma\xi_2)) \\
G \vDash^\sigma \xi_1 = \xi_2 & \Leftrightarrow & \sigma\xi_1 = \sigma\xi_2 \\
G \vDash^\sigma \alpha_1 = \alpha_2 & \Leftrightarrow & \sigma\alpha_1 = \sigma\alpha_2 \\
G \vDash^\sigma \phi \mid \psi & \Leftrightarrow & \exists G_1, G_2.\ G \equiv G_1 \mid G_2 \ \mathit{and}\ G_1 \vDash^\sigma \phi \ \mathit{and}\ G_2 \vDash^\sigma \psi \\
G \vDash^\sigma \phi \wedge \psi & \Leftrightarrow & G \vDash^\sigma \phi \ \mathit{and}\ G \vDash^\sigma \psi \\
G \vDash^\sigma \neg\phi & \Leftrightarrow & G \nvDash^\sigma \phi \\
G \vDash^\sigma \exists x.\ \phi & \Leftrightarrow & \exists \mathbf{x} \in \mathcal{X}.\ G \vDash^{\sigma, x \rightarrow \mathbf{x}} \phi \\
G \vDash^\sigma \exists a.\ \phi & \Leftrightarrow & \exists \mathbf{a} \in \mathcal{A}.\ G \vDash^{\sigma, a \rightarrow \mathbf{a}} \phi \\
G \vDash^\sigma \mu R.\phi & \Leftrightarrow & G \vDash^\sigma \phi\{R \leftarrow (\mu R.\phi)\}
\end{array}
$$

We also use derived operators: disjunction $\phi \vee \psi$, implication $\phi \Rightarrow \psi$ and universal quantifiers $\forall x.\phi$ and $\forall a.\phi$ are defined as usual. We also use the derived operator $\phi \mid\Rightarrow \psi =_{def} \neg(\phi \mid \neg\psi)$. By this definition, $G \vDash^\sigma \phi \mid\Rightarrow \psi$ if, and only if, for every partition of G into G', G'', if $G' \vDash^\sigma \phi$, then $G'' \vDash^\sigma \psi$.

The logic cannot distinguish graphs which are graph-equal (Lemma 2.3). On the other hand, for any single finite graph there is a sentence that characterises that, and only that, graph up to graph-equality (Lemma 2.4). Hence, logical equivalence coincides with graph-equality.

Lemma 2.3 *For any two graphs $G \equiv G'$, for any sentence ϕ of GL_μ, $\quad G \vDash \phi \quad \Leftrightarrow \quad G' \vDash \phi$. The analogous result for GL is an immediate corollary.*

Lemma 2.4 *Every finite graph G has a characteristic sentence $\underline{G}$ in GL, and hence in GL_μ. Formally, for any finite graph G, let $\underline{G}$ be the sentence that contains one $\mathbf{a}(\mathbf{x}, \mathbf{y})$ literal, separated by $|$, for each edge in the graph that is labelled $(\mathbf{a}, \mathbf{x}, \mathbf{y})$. Then: $\ G' \vDash \underline{G} \quad \Leftrightarrow \quad G' \equiv G$.*

2.3 The Range of Quantifiers

In GL_μ, quantifiers are interpreted as ranging over the infinite sets of names $\mathcal{X}$ and $\mathcal{A}$, even though the number of edges in the graph G is finite. In MSO however, each graph carries the set of its own nodes, connected or isolated, which is a finite subset X of $\mathcal{X}$, and quantification only ranges over this finite set of nodes, and similarly for labels $\mathbf{a}$. In this section, we show that the two approaches are basically equivalent. To this end, we define a variant $G \vDash^\sigma_{X,A} \phi$ of the satisfaction relation, where quantification ranges over $X \subseteq \mathcal{X}$ and $A \subseteq \mathcal{A}$, and prove that $\quad G \vDash^\sigma_{X,A} \phi \Leftrightarrow G \vDash^\sigma \phi \quad$ for 'big enough' X and A.

Definition 2.5 *The set of graphs $[\![\phi]\!]^{X,A}_{\sigma;\rho}$ is defined as in the definition of $[\![\phi]\!]_{\sigma;\rho}$, apart from the existential cases, where quantification ranges over X and A instead of $\mathcal{X}$ and $\mathcal{A}$:*

$$
[\![\exists x.\ \phi]\!]^{X,A}_{\sigma;\rho} \;=\; \bigcup_{\mathbf{x} \in X} [\![\phi]\!]^{X,A}_{\sigma, x \mapsto \mathbf{x};\rho} \qquad\qquad [\![\exists a.\ \phi]\!]^{X,A}_{\sigma;\rho} \;=\; \bigcup_{\mathbf{a} \in A} [\![\phi]\!]^{X,A}_{\sigma, a \mapsto \mathbf{a};\rho}
$$

For μ-closed formulas, we use $G \vDash^\sigma_{X,A} \phi$ to denote $G \in [\![\phi]\!]^{X,A}_{\sigma;()}$, which satisfies the properties:

$$
G \vDash^\sigma_{X,A} \exists x.\ \phi \quad \Leftrightarrow \quad \exists \mathbf{x} \in X.\ G \vDash^{\sigma, x \rightarrow \mathbf{x}}_{X,A} \phi \qquad\qquad G \vDash^\sigma_{X,A} \exists a.\ \phi \quad \Leftrightarrow \quad \exists \mathbf{a} \in A.\ G \vDash^{\sigma, a \rightarrow \mathbf{a}}_{X,A} \phi
$$

The standard GL_μ interpretation is obtained in the special case when $X = \mathcal{X}$ and $A = \mathcal{A}$.

We first prove a technical result that names and labels can be freely exchanged in the relativized version of the logic (hence, in the standard version as well). We first define name exchange, where $\mathbf{x}$ and $\mathbf{x}'$ are two arbitrary node names. Label exchange $_\{\mathbf{a} \leftrightarrow \mathbf{a}'\}$ is defined the same way, apart from the obvious modification of the first two lines; this is the reason why, in the $\alpha(\xi_1, \xi_2)$ case, we write $\alpha\{\mathbf{x} \leftrightarrow \mathbf{x}'\}$ instead of just α. Name exchange is defined by:

Terms: $\mathbf{x}\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \mathbf{x}'$, $\quad \mathbf{x}'\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \mathbf{x}$, $\quad \mathbf{x}''\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \mathbf{x}''$ if $\mathbf{x}'' \notin \{\mathbf{x}, \mathbf{x}'\}$
$\mathbf{a}\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \mathbf{a}$
$x\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} x$, $\quad a\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} a$,

Formulas: $(\xi_1 = \xi_2)\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} (\xi_1\{\mathbf{x} \leftrightarrow \mathbf{x}'\}) = (\xi_2\{\mathbf{x} \leftrightarrow \mathbf{x}'\})$
$(\alpha(\xi_1, \xi_2))\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} (\alpha\{\mathbf{x} \leftrightarrow \mathbf{x}'\})((\xi_1\{\mathbf{x} \leftrightarrow \mathbf{x}'\}), (\xi_2\{\mathbf{x} \leftrightarrow \mathbf{x}'\}))$
$(\exists x.\phi)\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \exists x.(\phi\{\mathbf{x} \leftrightarrow \mathbf{x}'\})$
$\ldots$

Graphs: $(E, \mathsf{edge})\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} (E, (\mathsf{edge}\{\mathbf{x} \leftrightarrow \mathbf{x}'\}))$, where:
$\mathsf{edge}(e) = (\mathbf{a}, \mathbf{x}_1, \mathbf{x}_2)$
$\Rightarrow (\mathsf{edge}\{\mathbf{x} \leftrightarrow \mathbf{x}'\})(e) = (\mathbf{a}\{\mathbf{x} \leftrightarrow \mathbf{x}'\}, \mathbf{x}_1\{\mathbf{x} \leftrightarrow \mathbf{x}'\}, \mathbf{x}_2\{\mathbf{x} \leftrightarrow \mathbf{x}'\}) \ldots$

Sets: $T\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \{t\{\mathbf{x} \leftrightarrow \mathbf{x}'\} \; : \; t \in T\}$

Substitutions: $()\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} ()$
$(\sigma, y \to \mathbf{y})\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \sigma\{\mathbf{x} \leftrightarrow \mathbf{x}'\}, y \to (\mathbf{y}\{\mathbf{x} \leftrightarrow \mathbf{x}'\})$
$(\sigma, a \to \mathbf{a})\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \sigma\{\mathbf{x} \leftrightarrow \mathbf{x}'\}, a \to (\mathbf{a}\{\mathbf{x} \leftrightarrow \mathbf{x}'\})$
$(\rho, R \to S)\{\mathbf{x} \leftrightarrow \mathbf{x}'\} =_{def} \rho\{\mathbf{x} \leftrightarrow \mathbf{x}'\}, R \to (S\{\mathbf{x} \leftrightarrow \mathbf{x}'\})$

Satisfaction is preserved by name and label exchange. We prove this result for formulas which may contain free recursion variables, since that allows us to reason by induction on the formula size in the $\mu R.\,\phi$ case.

Proposition 2.6 *For arbitrary node names $\mathbf{x}$ and $\mathbf{x}'$, labels $\mathbf{a}$ and $\mathbf{a}'$, graph G, sets X and A, formula ϕ in GL_μ, and substitutions σ and ρ:*

$$G \in [\![\phi]\!]^{X,A}_{\sigma;\rho} \quad \Leftrightarrow \quad G\{\mathbf{x} \leftrightarrow \mathbf{x}'\} \in [\![\phi\{\mathbf{x} \leftrightarrow \mathbf{x}'\}]\!]^{X\{\mathbf{x}\leftrightarrow\mathbf{x}'\}, A\{\mathbf{x}\leftrightarrow\mathbf{x}'\}}_{\sigma\{\mathbf{x}\leftrightarrow\mathbf{x}'\};\rho\{\mathbf{x}\leftrightarrow\mathbf{x}'\}}$$

$$G \in [\![\phi]\!]^{X,A}_{\sigma;\rho} \quad \Leftrightarrow \quad G\{\mathbf{a} \leftrightarrow \mathbf{a}'\} \in [\![\phi\{\mathbf{a} \leftrightarrow \mathbf{a}'\}]\!]^{X\{\mathbf{a}\leftrightarrow\mathbf{a}'\}, A\{\mathbf{a}\leftrightarrow\mathbf{a}'\}}_{\sigma\{\mathbf{a}\leftrightarrow\mathbf{a}'\};\rho\{\mathbf{a}\leftrightarrow\mathbf{a}'\}}$$

Proof Since the double application of $_\{\mathbf{x} \leftrightarrow \mathbf{x}'\}$ is the identity, we can just prove the ($\Rightarrow$) direction. The proof is by induction and by cases. It is long, but quite trivial. We present here the base case of equality and the inductive case for recursion, for name exchange only. For readability, we abbreviate $t\{\mathbf{x} \leftrightarrow \mathbf{x}'\}$ to $t^{\leftrightarrow}$, where t may be a term, a formula, a graph, a substitution, or a set of such things.

When ϕ is $\xi_1 = \xi_2$, we must prove that $\sigma(\xi_1) = \sigma(\xi_2) \;\Leftrightarrow\; \sigma^{\leftrightarrow}(\xi_1{}^{\leftrightarrow}) = \sigma^{\leftrightarrow}(\xi_2{}^{\leftrightarrow})$. By injectivity of $_^{\leftrightarrow}$ on constants, we have $\sigma(\xi_1) = \sigma(\xi_2)$ iff $(\sigma(\xi_1))^{\leftrightarrow} = (\sigma(\xi_2))^{\leftrightarrow}$. The result follows, since $(\sigma(\xi))^{\leftrightarrow} = \sigma^{\leftrightarrow}(\xi^{\leftrightarrow})$.

In the case $\phi = \mu R.\psi$, we have to prove that $G \in [\![\mu R.\psi]\!]^{X,A}_{\sigma;\rho} \;\Rightarrow\; G^{\leftrightarrow} \in [\![(\mu R.\psi)^{\leftrightarrow}]\!]^{X^{\leftrightarrow},A^{\leftrightarrow}}_{\sigma^{\leftrightarrow};\rho^{\leftrightarrow}}$. By definition, we have to prove that, for any G, (a) $\forall S.\ ([\![\psi]\!]^{X,A}_{\sigma;\rho,R\mapsto S} \subseteq S \;\Rightarrow\; G \in S)$ implies (b) $\forall S.\ ([\![\psi^{\leftrightarrow}]\!]^{X^{\leftrightarrow},A^{\leftrightarrow}}_{\sigma^{\leftrightarrow};\rho^{\leftrightarrow},R\mapsto S} \subseteq S \;\Rightarrow\; G^{\leftrightarrow} \in S)$. Let us assume (a) and (b') $[\![\psi^{\leftrightarrow}]\!]^{X^{\leftrightarrow},A^{\leftrightarrow}}_{\sigma^{\leftrightarrow};\rho^{\leftrightarrow},R\mapsto S} \subseteq S$ for an arbitrary set S. If we can prove that (c) $[\![\psi]\!]^{X,A}_{\sigma;\rho,R\mapsto S^{\leftrightarrow}} \subseteq S^{\leftrightarrow}$, then $G \in S^{\leftrightarrow}$ by (a), and hence $G^{\leftrightarrow} \in S$ follows immediately. So we only have to prove that (b') $[\![\psi^{\leftrightarrow}]\!]^{X^{\leftrightarrow},A^{\leftrightarrow}}_{\sigma^{\leftrightarrow};\rho^{\leftrightarrow},R\mapsto S} \subseteq S$ implies (c) $[\![\psi]\!]^{X,A}_{\sigma;\rho,R\mapsto S^{\leftrightarrow}} \subseteq S^{\leftrightarrow}$. Assume (b') and assume that a generic G belongs to $[\![\psi]\!]^{X,A}_{\sigma;\rho,R\mapsto S^{\leftrightarrow}}$; we have to show that $G \in S^{\leftrightarrow}$:

$$
\begin{array}{lll}
G \in [\![\psi]\!]^{X,A}_{\sigma;(\rho,R\mapsto S^{\leftrightarrow})} \Rightarrow \text{(induction on } \psi) & G^{\leftrightarrow} \in [\![\psi^{\leftrightarrow}]\!]^{X^{\leftrightarrow},A^{\leftrightarrow}}_{\sigma^{\leftrightarrow};(\rho,R\mapsto S^{\leftrightarrow})^{\leftrightarrow}} & \Rightarrow G^{\leftrightarrow} \in [\![\psi^{\leftrightarrow}]\!]^{X^{\leftrightarrow},A^{\leftrightarrow}}_{\sigma^{\leftrightarrow};\rho^{\leftrightarrow},R\mapsto S} \\
\qquad\qquad \Rightarrow \text{(by (b'))} & G^{\leftrightarrow} \in S & \Rightarrow G \in S^{\leftrightarrow}
\end{array}
$$

□

We are now ready to prove that, for any μ-closed GL$_\mu$ formula ϕ, in order to check $G \vDash^{\sigma} \phi$, we can restrict quantification to any finite subset X of $\mathcal{X}$ provided it is 'big enough'. We say that $X \subseteq \mathcal{X}$ is big enough for (G, ϕ, σ), written $X \sqsupseteq (G, \phi, \sigma)$, if $X \supseteq (\text{nam}(G, \phi) \cup (\text{range}(\sigma) \cap \mathcal{X}))$ and $X \setminus (\text{nam}(G, \phi) \cup \text{range}(\sigma))$ contains k distinct names, where k is the nesting level of name quantifiers in ϕ. The definition that $A \subseteq \mathcal{A}$ is big enough for (G, ϕ, σ), written $A \sqsupseteq (G, \phi, \sigma)$, is similar.

Proposition 2.7 *For any graph G, μ-closed GL$_\mu$ formula ϕ, substitution σ, name set X and label set A, such that dom(σ) $\supseteq$ fv(ϕ), $X \sqsupseteq (G, \phi, \sigma)$, and $A \sqsupseteq (G, \phi, \sigma)$, we have*

$$G \vDash^{\sigma} \phi \;\Leftrightarrow\; G \vDash^{\sigma}_{X,A} \phi$$

Proof The implication from right to left is immediate since $X \subseteq \mathcal{X}$. To prove the other direction, we use Lemma 2.2 and reason by cases on the shape of ϕ and by induction on its size, exploiting the fact that, if ψ is a subformula of ϕ, then $X \sqsupseteq (G, \phi, \sigma)$ implies $X \sqsupseteq (G, \psi, \sigma)$, and similarly for $A \sqsupseteq (G, \phi, \sigma)$. All cases apart from existential quantification are trivial.
Case $\phi = \exists x.\ \psi$. By definition, we know that $G \vDash^{\sigma} \exists x.\ \psi \Leftrightarrow \exists \mathbf{x} \in \mathcal{X}.\ G \vDash^{\sigma,x\to\mathbf{x}} \psi$. We have now to prove that $\exists \mathbf{x}' \in X.\ G \vDash^{\sigma,x\to\mathbf{x}'} \psi$. Consider a witness $\mathbf{x}$ for $\exists \mathbf{x} \in \mathcal{X}.\ G \vDash^{\sigma,x\to\mathbf{x}} \psi$. If $\mathbf{x} \in X$ we are done. If $\mathbf{x} \notin X$, then it can be substituted by any of the k fresh names in X specifically given for this purpose. Formally, let $\mathbf{x}'$ be one of these fresh names. By Proposition 2.6, $G \vDash^{\sigma,x\to\mathbf{x}} \psi$ implies (a) $G\{\mathbf{x} \leftrightarrow \mathbf{x}'\} \vDash^{(\sigma,x\to\mathbf{x})\{\mathbf{x}\leftrightarrow\mathbf{x}'\}} \psi\{\mathbf{x} \leftrightarrow \mathbf{x}'\}$. Since $\mathbf{x} \notin X$, and by freshness of $\mathbf{x}'$, neither $\mathbf{x}$ nor $\mathbf{x}'$ appear in nam(G, ϕ) nor in range(σ), hence (a) can be rewritten as $G \vDash^{\sigma,x\to\mathbf{x}'} \psi$. Observe now that $X \sqsupseteq (G, \psi, (\sigma, x \to \mathbf{x}'))$, since $X \supseteq (\text{nam}(G, \psi) \cup (\text{range}(\sigma, x \to \mathbf{x}') \cap \mathcal{X}))$ holds since $\mathbf{x}' \in X$ and $X \setminus (\text{nam}(G, \psi) \cup \text{range}(\sigma, x \to \mathbf{x}'))$ has one name less than $X \setminus (\text{nam}(G, \phi) \cup \text{range}(\sigma))$ but ψ has one quantification nesting level less than ϕ. Hence, by induction, from $\exists \mathbf{x}' \in X.\ G \vDash^{\sigma,x\to\mathbf{x}'} \psi$, we get $\exists \mathbf{x}' \in X.\ G \vDash^{\sigma,x\to\mathbf{x}'}_{X,A} \psi$ and hence $G \vDash^{\sigma}_{X,A} \phi$.

□

Corollary 2.8 *For any sentence ϕ of GL$_\mu$, for any X and A such that $X \sqsupseteq (G, \phi, \emptyset)$ and $A \sqsupseteq (G, \phi, \emptyset)$, $G \vDash \phi$ if, and only if, $G \vDash_{X,A} \phi$.*

2.4 Graph Logic and TQL Logic

The logic underlying the TQL query language, which we denote TL, is essentially the static fragment of Cardelli and Gordon's Ambient Logic without adjoints. It is a *generalization* of GL$_\mu$ that is defined on trees rather than graphs. This may sound strange, since trees are usually defined as a special case of graphs, rather than a generalization of graphs. However, TQL trees are labelled over an infinite set of labels, and hence three-level flat trees are already rich enough to represent our graphs, and, under this representation, TL and GL have the same expressive power.

TQL trees are described by the grammar

T	::=	$\mathbf{0}$	empty rooted tree
		$\mathbf{a}[T]$	one edge labelled $\mathbf{a}$, root as the source node, target node the root of subtree T
		$T \mid T$	composition of two trees, joining the roots

For example, the term $\mathbf{a}[\mathbf{b}[\mathbf{0}] \mid \mathbf{b}[\mathbf{0}]]$ represents a tree with an edge labelled $\mathbf{a}$ joined to the root, and two children just consisting of edges labelled by $\mathbf{b}$. TL therefore has formula $\alpha[\phi]$ instad of $\alpha(\xi, \chi)$.

Any GL graph $\mathbf{a}_1(\mathbf{x}_1, \mathbf{x}'_1) \mid \ldots \mid \mathbf{a}_n(\mathbf{x}_n, \mathbf{x}'_n)$ can be faithfully represented by the three-level tree $\mathbf{a}_1[\mathbf{x}_1[\mathbf{x}'_1[\mathbf{0}]]] \mid \ldots \mid \mathbf{a}_n[\mathbf{x}_n[\mathbf{x}'_n[\mathbf{0}]]]$. In the same way, any GL formula can be transformed into an equivalent TL formula by substituting any atom $\alpha_i(\xi_i, \xi'_i)$ with $\alpha_i[\xi_i[\xi'_i[\mathbf{0}]]]$, modulo some technical work to enforce the disjointness of names from edge labels. Hence, TL can be described as a generalization of GL to a set of models where the edge constructor $\mathbf{a}(\mathbf{x}, \mathbf{y})$ is substituted with the subtree constructor $\mathbf{a}[T]$. We decided to focus our study on GL, rather than TL, since the generalization of $\alpha(\xi, \chi)$ to $\alpha[\phi]$ adds no interesting issues to model-checking complexity.

3 Graph Logic and MSO

We wish to compare the expressive power of GL to that of standard logics such as first- and second-order logic, whose expressive power on finite graphs has been extensively studied (see [16]). Among these, the logic that is closest to GL is monadic second-order logic (MSO) which is widely used as a standard of expressivity on strings, trees and graphs. We show (in Section 3.1) that every formula of GL is equivalent to a formula of MSO, subject to a minor qualification with regard to the range of quantification. In terms of expressivity, this implies that the class of graphs defined by any formula of GL is in the polynomial hierarchy. Whilst it seems unlikely that every property definable in MSO is expressible in GL, we do illustrate the richness of GL by showing (in Section 3.2) how to express properties such as graph connectivity, 2-colourability and the existence of two node-disjoint paths. None of these properties is definable in first-order logic. The richness of GL is further illustrated in Section 4, where we show that complete problems at every level of the polynomial hierarchy are expressible. There is one important class of graphs for which we are able to show that GL is as expressive as MSO. This is the class of *string graphs*, i.e. graphs encoding words over a given alphabet. It is known that on this class MSO defines exactly the regular languages. We show that the same is true for GL in Section 3.3.

3.1 Monadic Second-Order Logic

We show that, for every formula of GL, there is an equivalent formula of monadic second-order logic (MSO). We will use a version of MSO with three sorts of variables, corresponding to vertices, edges and labels in the graphs. The logic will be interpreted in a three-sorted structure accordingly. Moreover, we treat edge syntactically as a 4-ary relation, rather than a function. To be precise, the formulas of MSO are built up from the set of names $\mathcal{X}$, the labels $\mathcal{A}$, a countable set of edges $\mathcal{E}$, three sorts of first-order variables $V_{\mathcal{X}}$, $V_{\mathcal{A}}$ and $V_{\mathcal{E}}$, and a countable collection $V_{\mathcal{S}}$ of set variables. A formula is one of:

$\mathsf{edge}(e, \alpha, \xi_1, \xi_2)$	where $e \in V_{\mathcal{E}}, \alpha \in \mathcal{A} \cup V_{\mathcal{A}}$ and $\xi_i \in \mathcal{X} \cup V_{\mathcal{X}}$
$e_1 = e_2$ or $\alpha_1 = \alpha_2$ or $\xi_1 = \xi_2$	where $e_i \in V_{\mathcal{E}}, \alpha_i \in \mathcal{A} \cup V_{\mathcal{A}}, \xi_i \in \mathcal{X} \cup V_{\mathcal{X}}$
$e \in S$	where $e \in V_{\mathcal{E}}$ and $S \in V_{\mathcal{S}}$
$\phi \wedge \psi$ or $\neg\phi$	where ϕ and ψ are formulas
$\exists e.\phi$ or $\exists x.\phi$ or $\exists a.\phi$	where $e \in V_{\mathcal{E}}, x \in V_{\mathcal{X}}, a \in V_{\mathcal{A}}$ and ϕ is a formula
$\exists S.\phi$	where $S \in V_{\mathcal{S}}$ and ϕ is a formula.

Formulas are interpreted over three-sorted structures (A, X, E, edge) such that $A \subseteq \mathcal{A}$, $X \subseteq \mathcal{X}$, $E \subseteq \mathcal{E}$, and $\mathsf{edge} : E \to A \times X \times X$. The semantics is standard except that set variables are interpreted as ranging over *sets of edges*. In this sense, this version of monadic second-order logic is akin to the logic MS_2 of [12].

In order to define a translation of GL-formulas to MSO-formulas, there is one issue that needs to be addressed carefully. The node and label quantifiers in GL are interpreted over the infinite sets $\mathcal{X}$ and $\mathcal{A}$. However, in the standard semantics of MSO, the range of the first-order quantifiers is the set of nodes

X and the set of labels A that are *in* the graph. However, we know by Corollary 2.8 that, without loss of generality, the quantifiers in GL can be restricted to a finite set.

We are now ready to give the translation of formulas of GL into MSO. More generally, we give for each formula ϕ of GL, a formula $[\![\phi]\!]^S$ of MSO which is to be read as "the translation of ϕ relativized to the set of edges S". In our translation, we use $S = S' \mid S''$ as an abbreviation for the MSO formula:

$$(\forall e.\ e \in S \ \Leftrightarrow\ (e \in S' \vee e \in S'')) \wedge \neg(\exists e.\ e \in S' \wedge e \in S'')$$

Definition 3.1 *We define a translation* $[\![_]\!]^S$ *from GL-formulas to MSO-formulas by induction on the structure of the GL-formulas:*

$$\begin{array}{llll}
[\![\mathbf{0}]\!]^S & =_{def} \neg\exists e.\ e \in S & & \\
[\![\alpha(\xi,\xi')]\!]^S & =_{def} \exists e \in S.\ [\mathsf{edge}(e,\alpha,\xi,\xi') \wedge \forall e' \in S.\ e = e'] & & \\
[\![\phi' \mid \phi'']\!]^S & =_{def} \exists S', S''.\ S = S' \mid S'' \wedge [\![\phi']\!]^{S'} \wedge [\![\phi'']\!]^{S''} & & \\
[\![\xi = \xi']\!]^S & =_{def} \xi = \xi' & [\![\alpha = \alpha']\!]^S & =_{def} \alpha = \alpha' \\
[\![\neg\phi]\!]^S & =_{def} \neg[\![\phi]\!]^S & [\![\phi \wedge \phi']\!]^S & =_{def} [\![\phi]\!]^S \wedge [\![\phi']\!]^S \\
[\![\exists x.\phi]\!]^S & =_{def} \exists x.\ [\![\phi]\!]^S & [\![\exists a.\phi]\!]^S & =_{def} \exists a.\ [\![\phi]\!]^S
\end{array}$$

A crucial property of this translation is the fact that every quantification over edges in $[\![\phi]\!]^S$ is actually bounded over S. This observation is explicit in cases $[\![\mathbf{0}]\!]^S$ and $[\![\alpha(\xi,\xi')]\!]^S$, and follows inductively in the other cases. It is formalized in the following lemma.

Lemma 3.2 *For any* $A \subseteq \mathcal{A}$, $X \subseteq \mathcal{X}$, *any two graphs* (E, edge) *and* (E', edge') *such that* $E \cap E' = \emptyset$, *any formula* $\phi \in GL$ *and any substitution* σ *over* $V_{\mathcal{X}}$ *and* $V_{\mathcal{A}}$,

$$(A, X, E, \mathsf{edge}) \models^{\sigma, S \to E}_{MSO} [\![\phi]\!]^S \ \Leftrightarrow\ (A, X, E \uplus E', \mathsf{edge} \uplus \mathsf{edge}') \models^{\sigma, S \to E}_{MSO} [\![\phi]\!]^S$$

We are now ready to formalize the relation between ϕ and $[\![\phi]\!]^S$.

Theorem 3.3 *For any graph* $G = (E, \mathsf{edge})$, *any formula* $\phi \in GL$, *any substitution* σ *over* $V_{\mathcal{X}}$ *and* $V_{\mathcal{A}}$, *any* $X \subseteq \mathcal{X}$ *and* $A \subseteq \mathcal{A}$ *such that* $X \sqsupseteq (G, \phi, \sigma)$, $A \sqsupseteq (G, \phi, \sigma)$:

$$G \models^{\sigma} \phi \ \Leftrightarrow\ (A, X, E, \mathsf{edge}) \models^{\sigma, S \to E}_{MSO} [\![\phi]\!]^S.$$

Proof By induction on the formula. We show the two interesting cases.

$G = (E, \mathsf{edge}) \models^{\sigma} \phi' \mid \phi''$:
$\Leftrightarrow$ $\exists E', E'', \mathsf{edge}', \mathsf{edge}''.\ E = E' \uplus E'', \mathsf{edge} = \mathsf{edge}' \uplus \mathsf{edge}''$,
$(E', \mathsf{edge}') \models^{\sigma} \phi', (E'', \mathsf{edge}'') \models^{\sigma} \phi''$
$\Leftrightarrow$ (by induction)
$\exists E', E'', \mathsf{edge}', \mathsf{edge}''.\ E = E' \uplus E'', \mathsf{edge} = \mathsf{edge}' \uplus \mathsf{edge}''$,
$(A, X, E', \mathsf{edge}') \models^{\sigma, S' \to E'}_{MSO} [\![\phi']\!]^{S'}, (A, X, E'', \mathsf{edge}'') \models^{\sigma, S'' \to E''}_{MSO} [\![\phi'']\!]^{S''}$
$\Leftrightarrow$ (by Lemma 3.2)
$\exists E', E'', \mathsf{edge}', \mathsf{edge}''.\ E = E' \uplus E'', \mathsf{edge} = \mathsf{edge}' \uplus \mathsf{edge}''$,
$(A, X, E, \mathsf{edge}) \models^{\sigma, S' \to E'}_{MSO} [\![\phi']\!]^{S'}, (A, X, E, \mathsf{edge}) \models^{\sigma, S'' \to E''}_{MSO} [\![\phi'']\!]^{S''}$
$\Leftrightarrow$ (we assume that S, S' and S'' are fresh variables)
$\exists E', E'' \subseteq E$.
$(A, X, E, \mathsf{edge}) \models^{\sigma, S \to E, S' \to E', S'' \to E''}_{MSO} S = S' \mid S''$
$(A, X, E, \mathsf{edge}) \models^{\sigma, S \to E, S' \to E', S'' \to E''}_{MSO} [\![\phi']\!]^{S'}$,
$(A, X, E, \mathsf{edge}) \models^{\sigma, S \to E, S' \to E', S'' \to E''}_{MSO} [\![\phi'']\!]^{S''}$,
$\Leftrightarrow$ $(A, X, E, \mathsf{edge}) \models^{\sigma, S \to E}_{MSO} \exists S', S''.\ S = S' \mid S'' \wedge [\![\phi']\!]^{S'} \wedge [\![\phi'']\!]^{S''}$

$G \vDash^{\sigma} \exists x.\, \psi$:
By definition, $\exists \mathbf{x} \in \mathcal{X}.\ G \vDash^{\sigma, x \to \mathbf{x}} \phi$. We reason as in the proof of Proposition 2.7 and show that this implies that: $\exists \mathbf{x} \in X.\ G \vDash^{\sigma, x \to \mathbf{x}} \phi$ (the ($\Leftarrow$) direction is trivial) and that $X \sqsupseteq (G, \psi, (\sigma, x \to \mathbf{x}))$. By induction, this is equivalent to

$$\exists \mathbf{x} \in X.\ (A, X, E, \mathsf{edge}) \vDash_{MSO}^{\sigma, x \to \mathbf{x}, s \to E} [\![\psi]\!]^E \ \Leftrightarrow$$
$$\exists \mathbf{x} \in X.\ (A, X, E, \mathsf{edge}) \vDash_{MSO}^{\sigma, s \to E, x \to \mathbf{x}} [\![\psi]\!]^E \ \Leftrightarrow$$
$$(A, X, E, \mathsf{edge}) \vDash_{MSO}^{\sigma, s \to E} \exists x.\ [\![\psi]\!]^E \ \Leftrightarrow\ (A, X, E, \mathsf{edge}) \vDash_{MSO}^{\sigma, s \to E} [\![\exists x.\ \psi]\!]^E.$$

□

The effect of Theorem 3.3 is that it immediately establishes upper bounds on the expressive power of GL and the complexity of the model-checking problem. It is known by [17, 26] that a property of relational structures is definable in second-order logic if, and only if, it is in the polynomial hierarchy PH. Thus, we have:

Corollary 3.4 *For any sentence $\phi \in GL$, the set of graphs $\mathcal{G}_\phi = \{G : G \vDash \phi\}$ belongs to* PH.

Proof In order to check whether $G \in \mathcal{G}_\phi$ one only needs to build A and X which are big enough for G, ϕ and check whether $(A, X, E, \mathsf{edge}) \vDash_{MSO}^{S \to E} [\![\psi]\!]^S$.

□

Moreover, for each level Σ_i^p of the polynomial hierarchy, there are properties definable in MSO that are Σ_i^p-complete. We show in Section 4 that this is also the case for GL. On the other hand, there are computationally simple properties that are not expressible in MSO which therefore could not be expressible in GL either. For instance, the following is a direct consequence of Theorem 3.3:

Corollary 3.5 *There is no formula ϕ of GL such that $G \vDash \phi$ if, and only if, G has an even number of edges.*

It is also known that the combined complexity of MSO is PSPACE-complete. That is, the decision problem $\text{Truths}_{\text{MSO}} = \{(G, \sigma, \phi) : \phi \in \text{MSO},\ G \vDash_{MSO}^{\sigma} \phi\}$ is PSPACE-complete. Since the translation given above is itself computable in polynomial time, it establishes that $\text{Truths}_{\text{GL}}$ is in PSPACE.

Corollary 3.6 *Truths_{GL} is in PSPACE.*

In Section 4 we show that it is, in fact, PSPACE-complete.

Remark: As in [6], the logic GL analyses an algebra for graph descriptions that does not involve name hiding. If name hiding is allowed, the operation of graph composition is subtly different. That is, if G_1 and G_2 are graphs, their composition $G_1 \mid G_2$ is defined by taking the disjoint union of the sets of edges and identifying nodes with the same name *provided that the name is not hidden*. Formulas of GL can still be translated to MSO under this interpretation, if we include in our graph structures a unary predicate Named for the set of nodes whose names are not hidden.

3.2 Expressing Properties with Graph Logic

We have seen that, in terms of expressive power, GL can be considered to be a fragment of monadic second-order logic. A translation in the other direction is not possible for trivial reasons, because of isolated nodes. For instance, in MSO, the sentence $\exists x \forall y \forall a \forall e (\neg \mathsf{edge}(e, a, x, y) \wedge \neg \mathsf{edge}(e, a, y, x))$ specifies that there is an isolated node in the graph, while in GL there is no distinction between isolated nodes and nodes that are not in the graph at all. Thus, for the purpose of a translation, it would make sense to restrict ourselves to sentences that are independent of isolated nodes. Still, we conjecture that

there is no translation possible. That is, we conjecture that there are MSO sentences that are independent of isolated nodes, but are still not equivalent to any sentence of GL. Establishing this conjecture is difficult. To illustrate this difficulty, we show how we can express some natural MSO properties in GL. Below we describe informally how these properties are expressed by GL sentences.

Connectivity. Graph connectivity is a problem well-known to be definable in MSO, but not in first-order logic (see [16]). We show that it is definable in GL. Consider the following formulas, which constrain the in- and out-degrees of nodes:

$$\begin{array}{llllll} \mathrm{In}_{\geq 1}(x) & =_{def} & \exists a, y.\ a(y,x) \mid \mathsf{T} & \mathrm{Out}_{\geq 1}(x) & =_{def} & \exists a, y.\ a(x,y) \mid \mathsf{T} \\ \mathrm{In}_{\geq 2}(x) & =_{def} & \mathrm{In}_{\geq 1}(x) \mid \mathrm{In}_{\geq 1}(x) & \mathrm{Out}_{\geq 2}(x) & =_{def} & \mathrm{Out}_{\geq 1}(x) \mid \mathrm{Out}_{\geq 1}(x) \\ \mathrm{In}_{0}(x) & =_{def} & \neg \mathrm{In}_{\geq 1}(x) & \mathrm{Out}_{0}(x) & =_{def} & \neg \mathrm{Out}_{\geq 1}(x) \\ \mathrm{In}_{1}(x) & =_{def} & \mathrm{In}_{\geq 1}(x) \wedge \neg \mathrm{In}_{\geq 2}(x) & \mathrm{Out}_{1}(x) & =_{def} & \mathrm{Out}_{\geq 1}(x) \wedge \neg \mathrm{Out}_{\geq 2}(x) \end{array} \quad (1)$$

We will also use the formula Here(x), defined by Here$(x) =_{def}$ $\mathrm{In}_{\geq 1}(x) \vee \mathrm{Out}_{\geq 1}(x)$. Using this formula we can identify those names that are actually in the graph. The formula $\forall x.$ Here$(x) \Rightarrow \phi$ specifies that ϕ holds for all names that are nodes in the graph. We henceforth use the abbreviation $\forall x \in G.\ \phi$ for $\forall x.$ Here$(x) \Rightarrow \phi$, and similarly $\forall x_1, \ldots, x_n \in G.\ \phi$.

Now, define the formula

$$\mathrm{Path}(x,y) =_{def} \mathrm{In}_0(x) \wedge \mathrm{Out}_0(y) \wedge \forall z \in G((z \neq x \Rightarrow \mathrm{In}_1(z)) \wedge (z \neq y \Rightarrow \mathrm{Out}_1(y))). \quad (2)$$

If, for a graph G, $G \vDash^{\sigma}$ Path(x,y), then G consists of a single path from σx to σy along with possibly other simple cycles disjoint from this path. Thus, $G \vDash^{\sigma}$ (Path$(x,y) \mid \mathsf{T}$) holds if, and only if, G contains a path from σx to σy. Indeed, if the formula is satisfied, then $G = G_1 \mid G_2$, where G_1 consists of a path and possibly some cycles, implying that G contains the required path. Conversely, if G contains a path from σx to σy, it can be expressed as the composition of this path and the rest of the graph, thereby satisfying the formula. Thus, we have that the formula $\forall x, y \in G.$(Path$(x,y) \mid \mathsf{T}$) expresses that a graph is strongly connected.

Disjoint Paths. Using the formula Path defined above, it is easy to construct a formula that expresses the property that there are two *edge-disjoint* paths between distinguished nodes $\mathbf{x}$ to $\mathbf{y}$ in a graph G. The formula is Path$(\mathbf{x},\mathbf{y}) \mid$ Path$(\mathbf{x},\mathbf{y}) \mid \mathsf{T}$. We show how to construct a formula that expresses that there are two *node-disjoint* paths from $\mathbf{x}$ to $\mathbf{y}$, which is a well-known NP-complete problem [18]. The formula TwoPaths(x,y)

$$\begin{array}{l} \mathrm{TwoPaths}(x,y) =_{def} \mathrm{Out}_{\geq 2}(x) \wedge \mathrm{In}_0(x) \wedge \mathrm{In}_{\geq 2}(y) \wedge \mathrm{Out}_0(y) \\ \qquad \wedge \forall z \in G.\ ((z \neq x \wedge z \neq y \Rightarrow \mathrm{In}_1(z)) \wedge (z \neq x \wedge z \neq y \Rightarrow \mathrm{Out}_1(z))) \end{array}$$

is true in a graph G if, and only if, G consists of two node-disjoint paths from x to y and possibly some additional simple cycles. Thus, TwoPaths$(x,y) \mid \mathsf{T}$ expresses the existence of two node-disjoint paths from x to y.

2-colourability. As a final example, we construct a formula that expresses that a graph is 2-colourable. In this discussion we assume that the graph satisfies $\neg(\exists a, x.\ a(x,x) \mid \mathsf{T})$, hence has no self loops. Similarly to the formulas $\mathrm{In}_{\geq 2}$ and $\mathrm{Out}_{\geq 2}$, we define the formula $\mathrm{Deg}_{\geq n}(x)$ to be the formula that asserts that x has at least n neighbours (regardless of the direction of the edges). Thus, for instance

$$\mathrm{Deg}_{\geq 2}(x) =_{def} (\exists a, y.\ (a(y,x) \vee a(x,y))) \mid (\exists a, y.\ (a(y,x) \vee a(x,y))) \mid \mathsf{T}.$$

Also, define $\mathrm{Deg}_{=n}(x)$ to be the formula $\mathrm{Deg}_{\geq n}(x) \wedge \neg \mathrm{Deg}_{\geq n+1}(x)$. We also define the formulas

$$\begin{array}{lll} \mathrm{Cycles} & =_{def} & \forall x \in G.\ \mathrm{Deg}_{=2}(x) \\ \mathrm{Edges} & =_{def} & \forall x \in G.\ \mathrm{Deg}_{=1}(x). \end{array}$$

Note that, if every node in a graph G has exactly two neighbours, then G is the disjoint union of simple cycles (ignoring directions on the edges). Similarly, if every node in G has exactly one neighbour, then G is the disjoint union of simple edges. Thus, the sentences Cycles and Edges express, respectively, that G is a disjoint collection of cycles and that G is a disjoint collection of edges. Also observe that a collection of simple cycles can be decomposed into two graphs each of which is a disjoint collection of edges if, and only if, all the cycles are of even length. Thus, the sentence Cycles $\wedge\ \neg$(Edges | Edges) is satisfied by G if, and only if, G is a collection of cycles at least one of which is of odd length. It is well-known that a graph is 2-colourable if, and only if, it contains no cycles of odd length. Thus, 2-colourability is defined by the sentence: $\neg(\exists a, x.\ a(x,x) \mid \mathsf{T}) \wedge \neg[(\text{Cycles} \wedge \neg(\text{Edges} \mid \text{Edges})) \mid \mathsf{T}]$.

These examples illustrate the varied expressive power of GL. Nonetheless, it seems unlikely that GL is as expressive as MSO. We conjecture that *Hamiltonicity*—the class of graphs that contain a Hamiltonian cycle—and *3-colourability* are both inexpressible in GL. These can both be seen to be expressible in MSO[3]. Intuitively, the MSO definition of these properties combines a monadic quantification with a universal quantification on the nodes. For example, to say that a graph contains a Hamiltonian cycle we would say that it contains a set of edges which form a simple cycle and *every node in the graph* is on this cycle. In GL it is possible to select a set of edges that form a cycle using the | operator but, having selected this set of edges, we are restricted to statements about this set or its complement. We can no longer talk of *all* nodes in the original graph. Of course, this intuition does not prove that Hamiltonicity is not expressible in GL. In order to separate the expressive power of MSO from that of GL, we need a method that can demonstrate that a given property such as Hamiltonicity is not definable in GL. A natural such methods is the use of Ehrenfeucht-Fraïssé style games. Games for spatial logics such as GL along these lines are introduced in [14], where we use them to show that adjunct operators can be eliminated from the logic. Using these games we can show that GL cannot express that a graph has an even number of edges, yielding a direct proof of Corollary 3.5. It remains a challenge to deploy this game method to prove the inexpressibility of more complex problems, such as Hamiltonicity or 3-colourability.

3.3 Strings

One class of particularly simple finite graphs on which the expressive power of MSO has been well characterized is the class of finite strings over a fixed alphabet A. It is well-known (by an old result of Büchi [3]) that a set of strings is definable in MSO if, and only if, it is a regular language. It is natural to consider a string over the alphabet A as a graph with labels from A (the precise encoding is given below). We show that every regular language can be defined by a formula of GL. This shows that on this natural class of graphs the expressive power of GL and MSO coincide.

In all this section we assume a fixed finite alphabet $A \subset \mathcal{A}$.

A *string graph* over the finite alphabet A is either the empty graph or a graph of the shape

$$\mathbf{a}_1(\mathbf{x}_1, \mathbf{x}_2) \mid \ldots \mid \mathbf{a}_{n-1}(\mathbf{x}_{n-1}, \mathbf{x}_n),$$

where each of the labels $\mathbf{a}_i$ is in A and the $\mathbf{x}_i$ are distinct. Identifying the A-word $\mathbf{a}_1 \ldots \mathbf{a}_{n-1}$ with the graph $\mathbf{a}_1(\mathbf{x}_1, \mathbf{x}_2) \mid \ldots \mid \mathbf{a}_{n-1}(\mathbf{x}_{n-1}, \mathbf{x}_n)$, we can see each sentence ϕ of GL as defining a language—namely the set of A-words that satisfy ϕ.

Definition 3.7 *For each word $w = \mathbf{a}_1 \ldots \mathbf{a}_{n-1} \in A^*$ and each sequence $\vec{\mathbf{x}} = \mathbf{x}_1, \ldots, \mathbf{x}_n$ of n distinct names, we write $G(w, \vec{\mathbf{x}})$ to denote the graph $\mathbf{a}_1(\mathbf{x}_1, \mathbf{x}_2) \mid \ldots \mid \mathbf{a}_{n-1}(\mathbf{x}_{n-1}, \mathbf{x}_n)$. A string graph is any graph G that is graph-equal to $G(w, \vec{\mathbf{x}})$ for some w, $\vec{\mathbf{x}}$.*

[3] That is, in the version of MSO we have defined, which allows quantification over sets of edges. If second-order quantification is restricted to sets of vertices, then Hamiltonicity is not definable [16, Cor. 6.3.5], though 3-colourability still is.

Definition 3.8 *For any sentence ϕ of GL, the corresponding A-language is defined as the set:*

$$L_A(\phi) =_{def} \{w \in A^* \ : \ \forall \vec{\mathbf{x}}.\ G(w, \vec{\mathbf{x}}) \vDash \phi\}.$$

We will only consider sentences with no name constants, hence the specific choice of the sequence $\vec{\mathbf{x}}$ is irrelevant, hence we have that $L_A(\phi) = \{w \in A^* \ : \ \exists \vec{\mathbf{x}}.\ G(w, \vec{\mathbf{x}}) \vDash \phi\}$.

A language is definable in MSO if, and only if, it is regular (Büchi [3]). This classical result is obtained in a setting where each label is represented by a different relation, which means that the MSO formula cannot quantify over labels, while GL allows label quantification. This makes a difference if one consider complexity issues, since formulas can be written in a more concise form. However, since A is fixed and finite, label quantification does not affect the ability to define languages over A. Consider a formula ϕ and a label set A^+_ϕ such that $A^+_\phi \sqsupseteq (G_A, \phi, \emptyset)$, where G_A is a graph that contains every label in A. By Corollary 2.8, for every string graph S_A over A, $S_A \vDash \phi$ if, and only if, $S_A \vDash_{\mathcal{X},A} \phi$, hence $L_A(\phi)$ does not change if we replace every label quantification in ϕ with a finite quantification over A^+_ϕ. It follows that any language definable in GL is regular.

We now show the converse, that every regular language is definable by a formula of GL. We do this by translating regular expressions into formulas of GL. As usual, we write $L_A(r)$ for the language denoted by a regular expression r. The crucial case in the translation is the Kleene star. The translation to GL is based on the observation that a string graph is in the language $L_A(r^*)$ if, and only if, it can be decomposed into two graphs, each of which is the disconnected sum of strings in $L_A(r)$.

In order to define the translation, we first introduce some auxiliary formulas to count the incoming and outgoing edges. In addition to the formulas introduced at (1) and (2), we need:

$$\begin{aligned} \text{In}_{\leq 1}(x) &=_{def} \text{In}_0(x) \vee \text{In}_1(x) \\ \text{Out}_{\leq 1}(x) &=_{def} \text{Out}_0(x) \vee \text{Out}_1(x) \end{aligned}$$

We can now introduce some additional predicates. NoCycles means that no subgraph of the current graph is a cycle. SimplePath(x, y) means that the graph is just a simple path from x to y. Finally, SetOfWords(ϕ) means that the graph is a set of disconnected simple paths, and each of these paths satisfies the formula ϕ. The operator $\Rightarrow$ binds more than $|\Rightarrow$, hence $\mathsf{T} \mid\Rightarrow \Phi \Rightarrow \Psi$ means "every subgraph that satisfies Φ satisfies Ψ".

$$\begin{aligned} \text{NotEmptySetOfCycles} &=_{def} (\neg \mathbf{0}) \wedge \forall x \in G.\ (\text{In}_1(x) \wedge \text{Out}_1(x)) \\ \text{NoCycles} &=_{def} \neg(\mathsf{T} \mid \text{NotEmptySetOfCycles}) \\ \text{SimplePath}(x, y) &=_{def} \text{NoCycles} \wedge \text{Path}(x, y) \\ \text{Word} &=_{def} \exists x, y.\ \text{SimplePath}(x, y) \\ \text{Final}(x) &=_{def} \text{Here}(x) \wedge \text{Out}_0(x) \\ \text{Initial}(x) &=_{def} \text{Here}(x) \wedge \text{In}_0(x) \\ \text{SetOfWords}(\phi) &=_{def} \text{NoCycles} \wedge (\forall x \in G.\ \text{In}_{\leq 1}(x) \wedge \text{Out}_{\leq 1}(x)) \\ & \quad \wedge (\forall x, y \in G.\ (\text{In}_0(x) \wedge \text{Out}_0(y) \\ & \qquad \Rightarrow (\mathsf{T} \mid\Rightarrow \text{SimplePath}(x, y) \Rightarrow \phi))) \end{aligned}$$

We make the following observations with regard to the above formulas.

1. $G \vDash$ Word if, and only if, G is a string graph.
2. $G \vDash^\sigma$ SimplePath(x, y) if, and only if, G is a string graph and $\sigma(x)$ is the initial node and $\sigma(y)$ the final node.

3. $G \vDash \text{SetOfWords}(\phi)$ if, and only if, each connected component of G is graph-equal to some string graph $G(w)$ such that $G(w) \vDash \phi$.

We can now translate every regular expression r into an equivalent formula $\mathsf{F}(r)$. The critical case is Kleene star. A word w belongs to $L_A(r^*)$ if it is the concatenation of a sequence of words $w_1, \ldots, w_n$, each belonging to $L_A(r)$. In this case, it is possible to split the corresponding graph $G(w)$ into the two subgraphs G_1 and G_2 which are not themselves words, but each is a set of disconnected words, each of them individually in $L_A(r)$. G_1 contains the subgraphs of G corresponding to w_i for odd i and G_2 contains those corresponding to w_i for even i. On the other hand, if it is possible to split $G(w)$ into two such graphs, then $w \in L_A(r^*)$. This is expressed by the formula $\mathsf{F}(r^*)$ below. Observe that each of the two graphs G_1 and G_2 may be empty.

$$
\begin{aligned}
\mathsf{F}(\epsilon) &=_{def} \mathbf{0} \\
\mathsf{F}(\mathbf{a}) &=_{def} \exists x, y.\ \mathbf{a}(x, y) \\
\mathsf{F}(r; r') &=_{def} \text{Word} \wedge \exists x.\ (\mathsf{F}(r) \wedge (\mathbf{0} \vee \text{Final}(x))) \mid ((\mathbf{0} \vee \text{Initial}(x)) \wedge \mathsf{F}(r')) \\
\mathsf{F}(r + r') &=_{def} \mathsf{F}(r) \vee \mathsf{F}(r') \\
\mathsf{F}(r^*) &=_{def} \mathbf{0} \vee [\text{Word} \wedge (\text{SetOfWords}(\mathsf{F}(r)) \mid \text{SetOfWords}(\mathsf{F}(r)))]
\end{aligned}
$$

The following lemma establishes the correctness of the translation.

Lemma 3.9 *For any r, w, $\vec{\mathbf{x}} = \mathbf{x}_1 \ldots \mathbf{x}_{n+1}$ with $n = |w|$: $G(w, \vec{\mathbf{x}}) \vDash \mathsf{F}(r)$ if, and only if, $w \in L_A(r)$.*

Proof The proof is by induction on the structure of the regular expression r. The case when r is ϵ is trivial, as is the case of $+$. $G \vDash \mathsf{F}(r; r')$ if, and only if, $G \vDash$ Word (i.e. it is a string graph) and can be split in G_1 and G_2 such that, for some $\mathbf{x}$, $G_1 \vDash \mathsf{F}(r) \wedge (\mathbf{0} \vee \text{Final}(\mathbf{x}))$ and $G_2 \vDash (\mathbf{0} \vee \text{Initial}(\mathbf{x})) \wedge \mathsf{F}(r')$. By induction, G_1 and G_2 correspond to words $w \in r$ and $w' \in r'$. If either is empty, then G corresponds immediately to $w; w'$. If both words are non empty, then G corresponds to the concatenation of w' after w. In both cases, G corresponds to a word in $r; r'$. In the other direction, the proof that, for any $w \in r$ and $w' \in r'$, $G(w; w', \vec{\mathbf{x}}) \vDash \mathsf{F}(r; r')$ follows the same path. Just observe that, if both w and w' are empty, the witness for $\exists x$ is a name $\mathbf{x}$ that is not in the graph.

The crucial case is r^*. If $w \in L_A(r^*)$ then either w is ϵ, in which case $G(w) \vDash \mathsf{F}(r^*)$ or $w = w_1 \cdots w_n$ where each $w_i \in (L_A(r) \setminus \{\epsilon\})$. We now define the graphs G_1 and G_2 by including in G_1 all edges corresponding to all positions in w which appear in w_i for some *odd i* while G_2 contains edges corresponding to positions in w which appear in w_i for some *even i*. Then, each of G_1 and G_2 consists of a collection of disjoint string graphs corresponding to the words $w_1, \ldots, w_n$. Thus, each of G_1 and G_2 satisifies SetOfWords($\mathsf{F}(r)$) and therefore $G(w) \vDash \mathsf{F}(r^*)$. For the converse, suppose the $G \vDash \mathsf{F}(r^*)$. Then either $G \vDash \mathbf{0}$, in which case $G \equiv G(\epsilon)$ or G is a string graph corresponding to some word w with $G \equiv G_1 \mid G_2$ and $G_1, G_2 \vDash \text{SetOfWords}(\mathsf{F}(r))$. Thus, each connected component in each of G_1 and G_2 is a string graph corresponding to a word in $L_A(r)$, each such word is a contiguous substring of w and concatenating them all gives us back the original word. Thus, $w \in L_A(r^*)$. □

We can now establish the main theorem of this section.

Theorem 3.10 *A language is definable in GL if, and only if, it is regular.*

Proof If a language is regular, it is given by a regular expression r and therefore defined in GL by $\mathsf{F}(r)$ by Lemma 3.9. In the other direction, if a language is defined by a sentence of GL, it is also defined by a sentence of MSO and therefore, by Büchi's theorem it is regular. □

Büchi's characterization of the expressive power of MSO on strings has been extended in several different directions. For instance, it is known that the properties of finite trees that are definable in MSO are exactly those that can be recognized by tree automata. It would be interesting to know whether this could be used to extend the equivalence of GL and MSO from strings to trees.

4 Complexity and Expressivity of Graph Logic

The translation of GL to MSO given in Section 3.1 established upper bounds on the expressive power and model-checking complexity of GL (Corollaries 3.4 and 3.6). In this section we establish lower bounds that match these upper bounds by showing that the model-checking problem is PSPACE-complete and that GL can express complete problems at every level of the polynomial hierarchy.

4.1 Combined complexity

We begin by showing that the set $\mathrm{Truths}_{\mathrm{GL}} = \{(G, \sigma, \phi) \;:\; \phi \in \mathrm{GL},\; G \vDash^{\sigma} \phi\}$ is PSPACE-hard, by a reduction from validity of quantified Boolean formulas (QBF). We use a fixed graph in the reduction. Translation from QBF is a canonical technique, used in [4, 9] to prove the same result for the separation logic and the ambient logic. Indeed, the proof is based on the PSPACE-completeness of Boolean quantification and does not depend on the presence of spatial operators such as $\mid$. We include it here for the sake of completeness.

A quantified Boolean formula is a term generated by the grammar.

$$\Phi \;::=\; \exists P.\, \Phi \;\mid\; \neg\Phi \;\mid\; \Phi \wedge \Phi' \;\mid\; P$$

where P ranges over propositional variables. The definitions of validity and satisfiability for QBF are standard (see [23]). QBF formulas Φ are translated into GL formulas as follows, where for each P, $x_P \in V_{\mathcal{X}}$ is a different name variable, and $\mathbf{t} \in \mathcal{X}$ is a name constant. The translation can be evaluated in Logspace.

$$\begin{array}{llllll} [\![\exists P.\, \Phi]\!] & =_{def} & \exists x_P.\, [\![\Phi]\!] & \qquad [\![P]\!] & =_{def} & x_P = \mathbf{t} \\ [\![\neg\Phi]\!] & =_{def} & \neg[\![\Phi]\!] & \qquad [\![\Phi \wedge \Phi']\!] & =_{def} & [\![\Phi]\!] \wedge [\![\Phi']\!] \end{array}$$

Lemma 4.1 *Satisfiability of closed QBF formulas can be reduced to the model-checking problem*

$$\mathit{Truths}_{GL} = \{(G, \sigma, \phi) \;:\; \phi \in GL,\; G \vDash^{\sigma} \phi\}$$

for a fixed graph G.

Proof We take G to be the empty graph. Given an assignment σ of truth values to the propositional variables in a QBF formula Φ, define σ' to be any variable assignment such that $\sigma(P) = \text{true} \;\Leftrightarrow\; \sigma'(x_P) = \mathbf{t}$. An easy induction then shows that Φ is true under σ if, and only if, $G \vDash^{\sigma'} [\![\Phi]\!]$, hence the closed formula Φ is satisfiable if, and only if, $(\mathbf{0}, \epsilon, [\![\Phi]\!]) \in \mathrm{Truths}_{\mathrm{GL}}$. □

Corollary 4.2 *The model-checking problem for GL is PSPACE-complete.*

4.2 Expressivity and Data Complexity

Combined complexity describes how the complexity of model-checking depends on the size of both the model and the formula. However, Graph Logic has been first proposed as the foundation for a query language for semistructured data. In this context, data are much bigger than formulas, hence the important question is how the cost of model-checking, hence of query-answering, increases with data size, for each fixed query.

We present here the following data-complexity result: for each positive integer k, there is a sentence ϕ_k such that the problem: 'given G, decide whether $G \vDash \phi_k$' is hard for the kth universal level of the polynomial hierarchy, i.e. $\mathcal{G}_{\phi_k}$ is Π^p_k-hard. Since GL is closed under negation, we also obtain hard problems for the existential levels of the hierarchy.

Combined with the fact that any $\mathcal{G}_\phi$ belongs to PH (Corollary 3.4), this gives us a tight characterization of the expressive power of GL.

The canonical complete problem for Π^p_k is the satisfiability of quantified Boolean formulas with k alternations of quantifiers, beginning with a universal quantifier. We present a logspace-computable reduction from such formulas to a class of graphs that is defined by a formula of GL.

To establish the ground work which we can then generalize to higher levels of the hierarchy, we begin by an encoding of Boolean formulas *without quantifiers* into graphs. This enables us to reduce the NP-complete problem of propositional satisfiability to a class of graphs defined by a formula ϕ of GL. To be precise, we assume that we are given a Boolean formula Φ in negation normal form i.e., negations only appear in front of propositional variables. We define a function G so that $G(\Phi)$ is a graph encoding the formula Φ as a circuit. The graph has edge labels from among *And*, *Or*, *Lit*, *Pos* and *Neg*. It also has a node named **result**. Informally, there is a node for each variable X occurring in Φ. This node has a number of outgoing edges labelled *Pos* and *Neg*. There is one *Pos* edge for each positive occurrence of X in Φ and one *Neg* edge for each negative occurrence of X (i.e. occurrence of $\neg X$) in Φ. In addition, there is a distinct node in $G(\Phi)$ for each subformula Ψ of Φ. If Ψ is $\Psi_1 \vee \Psi_2$ the node has one outgoing edge labelled *Or* and incoming edges from the nodes corresponding to Ψ_1 and Ψ_2, and similarly for $\Psi_1 \wedge \Psi_2$. If Ψ is a literal X or $\neg X$, the outgoing edge is labelled *Lit*, and the incoming edge is the *Pos*, or *Neg*, edge described in the previous sentence. The edge starting from the node that corresponds to the whole formula Φ ends into one further node, **result**. Thus, the graph $G((X \vee \neg Y) \wedge (Z \vee Y))$ is depicted in Figure 1. The tokens X, Y and Z in this figure are only used to identify nodes and do not form part of the graph.

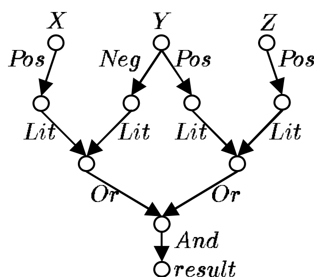

Figure 1: $G((X \vee \neg Y) \wedge (Z \vee Y))$

More formally, we define the translation G as follows. The translation is parametrized by a pair of functions $\gamma = (\gamma_v, \gamma_p)$. The function γ_v is an injective map taking each Boolean variable in Φ to a node. The function γ_p is an injective map taking occurrences in the formula Φ to nodes. The codomain of γ_p is disjoint from that of γ_v. As usual, an occurrence α is a string of 0's and 1's, possibly empty (ϵ), such

that, when α is associated to a formula rooted in a binary operator, $\alpha.0$ and $\alpha.1$ are associated to its two subformulas; we also consider one special occurrence, denoted ϵ^-. For an occurrence $\alpha \neq \epsilon^-$, $(\Phi)_\alpha$ denotes the subformula of Φ identified by α. The operator $(\alpha)^-$ is defined as $(\alpha.0)^- = (\alpha.1)^- = \alpha$, $(\epsilon)^- = \epsilon^-$. The functions γ_v and γ_p do not use up space, since they are not stored in a table, but perform some fixed bit-manipulation on the input to produce the output. As depicted in Figure 1, the function $[\![\Phi, \alpha]\!]^\gamma$ yields a DAG which is almost a tree, apart from the sharing of the nodes that correspond to variables; this tree is rooted in $\gamma_p(\alpha^-)$. We will often abbreviate $G(\Phi, \gamma)$ to $G(\Phi)$.

$$
\begin{array}{lll}
G(\Phi,\gamma) & =_{def} & [\![\Phi,\epsilon]\!]^\gamma\{\gamma_p(\epsilon^-) \leftarrow \textbf{result}\} \\
[\![\Phi \wedge \Phi', \alpha]\!]^\gamma & =_{def} & [\![\Phi,\alpha.0]\!]^\gamma \mid [\![\Phi',\alpha.1]\!]^\gamma \mid And(\gamma_p(\alpha),\gamma_p(\alpha^-)) \\
[\![\Phi \vee \Phi', \alpha]\!]^\gamma & =_{def} & [\![\Phi,\alpha.0]\!]^\gamma \mid [\![\Phi',\alpha.1]\!]^\gamma \mid Or(\gamma_p(\alpha),\gamma_p(\alpha^-)) \\
[\![\neg X, \alpha]\!]^\gamma & =_{def} & Neg(\gamma_v(X),\gamma_p(\alpha)) \mid Lit(\gamma_p(\alpha),\gamma_p(\alpha^-)) \\
[\![X, \alpha]\!]^\gamma & =_{def} & Pos(\gamma_v(X),\gamma_p(\alpha)) \mid Lit(\gamma_p(\alpha),\gamma_p(\alpha^-))
\end{array}
$$

Proposition 4.3

1. *For any occurrence $\alpha.\alpha'$ of Φ, the graph $[\![(\Phi)_{\alpha.\alpha'}, \alpha.\alpha']\!]^\gamma$ is a subgraph of $[\![(\Phi)_\alpha, \alpha]\!]^\gamma$, hence of $[\![\Phi, \epsilon]\!]^\gamma$.*
2. *For any occurrence α of Φ, $[\![(\Phi)_\alpha, \alpha]\!]^\gamma$ contains exactly one edge of the form $_(\gamma_p(\alpha), \gamma_p(\alpha^-))$.*
3. *For any occurrence α, if $\alpha.0$ and $\alpha.1$ are occurrences of Φ, then $[\![(\Phi)_\alpha, \alpha]\!]^\gamma$ contains exactly two edges that arrive at $\gamma_p(\alpha)$, one from $\gamma_p(\alpha.0)$ and the other from $\gamma_p(\alpha.1)$.*

Given a graph of the form $G(\Phi)$, we wish to identify a subgraph that can serve as a witness to the fact that Φ is satisfiable. To be precise, we say a subgraph S of $G(\Phi)$ is *satisfying* for the node $\mathbf{z}$ if the following conditions hold:

1. there is no node $\mathbf{x}$ with both outgoing edges labelled Pos and outgoing edges labelled Neg in S;
2. for every edge $Lit(\mathbf{x}, \mathbf{y})$ in S, $\mathbf{x}$ has an incoming edge in S;
3. for every edge $Or(\mathbf{x}, \mathbf{y})$ in S, $\mathbf{x}$ has an incoming edge in S;
4. for every edge $And(\mathbf{x}, \mathbf{y})$ in S, $\mathbf{x}$ has two incoming edges in S; and
5. $\mathbf{z}$ has an incoming edge in S.

This enables us to prove the key lemma linking satisfiability of Φ and $G(\Phi)$. We first define $[\![\Phi, \alpha, \sigma]\!]^\gamma$ as the subgraph of $[\![\Phi, \alpha]\!]^\gamma$ that only contains those edges that correspond to subformulas that are made true by the assignment σ. The notation "if C then G" denotes the graph G if C holds and the empty graph otherwise. $\sigma \vDash \Phi$ means that Φ holds under the assignment σ.

$$
\begin{array}{lll}
[\![\Phi \wedge \Phi', \alpha, \sigma]\!]^\gamma & =_{def} & [\![\Phi,\alpha.0,\sigma]\!]^\gamma \mid [\![\Phi',\alpha.1,\sigma]\!]^\gamma \mid \text{if } \sigma \vDash \Phi \wedge \Phi' \text{ then } And(\gamma_p(\alpha),\gamma_p(\alpha^-)) \\
[\![\Phi \vee \Phi', \alpha, \sigma]\!]^\gamma & =_{def} & [\![\Phi,\alpha.0,\sigma]\!]^\gamma \mid [\![\Phi',\alpha.1,\sigma]\!]^\gamma \mid \text{if } \sigma \vDash \Phi \vee \Phi' \text{ then } Or(\gamma_p(\alpha),\gamma_p(\alpha^-)) \\
[\![\neg X, \alpha, \sigma]\!]^\gamma & =_{def} & \text{if } \sigma \vDash \neg X \text{ then } (Neg(\gamma_v(X),\gamma_p(\alpha)) \mid Lit(\gamma_p(\alpha),\gamma_p(\alpha^-))) \\
[\![X, \alpha, \sigma]\!]^\gamma & =_{def} & \text{if } \sigma \vDash X \text{ then } (Pos(\gamma_v(X),\gamma_p(\alpha)) \mid Lit(\gamma_p(\alpha),\gamma_p(\alpha^-)))
\end{array}
$$

Proposition 4.4 *For any occurrence $\alpha.\alpha'$ of Φ, the graph $[\![(\Phi)_{\alpha.\alpha'}, \alpha.\alpha', \sigma]\!]^\gamma$ is a subgraph of $[\![(\Phi)_\alpha, \alpha, \sigma]\!]^\gamma$, hence of $[\![\Phi, \epsilon, \sigma]\!]^\gamma$.*

The next lemma implies that, if $\sigma \models \Phi$, then the graph $[\![\Phi, \epsilon, \sigma]\!]^\gamma$ is satisfying for $\gamma_p(\epsilon^-)$.

Lemma 4.5 *For any occurence α of any formula Φ, if $\sigma \models (\Phi)_\alpha$, then $[\![(\Phi)_\alpha, \alpha, \sigma]\!]^\gamma$ is a satisfying subgraph of $G(\Phi, \gamma)$ for the node $\gamma_p(\alpha^-)$.*

Proof By induction on the size of $(\Phi)_\alpha$. Conditions 1 and 2 hold by construction. Condition 4: by construction, if an edge $And(\mathbf{x}, \mathbf{y})$ is present in $[\![(\Phi)_\alpha, \alpha, \sigma]\!]^\gamma$, it must be equal to $And(\gamma_p(\beta), \gamma_p(\beta^-))$, for some $\beta = \alpha.\alpha'$ such that $(\Phi)_\beta = (\Phi)_{\beta.0} \wedge (\Phi)_{\beta.1}$ and $\sigma \models (\Phi)_\beta$. Hence, $\sigma \models (\Phi)_{\beta.0}$ and $\sigma \models (\Phi)_{\beta.1}$, hence, by induction, $[\![(\Phi)_{\beta.0}, \beta.0, \sigma]\!]^\gamma$ and $[\![(\Phi)_{\beta.1}, \beta.1, \sigma]\!]^\gamma$ are both satisfying for $\gamma_p((\beta.0)^-) = \gamma_p((\beta.1)^-) = \gamma_p(\beta)$, hence $[\![(\Phi)_\beta, \beta, \sigma]\!]^\gamma$ contains two distinct edges that enter $\gamma_p(\beta)$, hence $[\![(\Phi)_\alpha, \alpha, \sigma]\!]^\gamma$ satisfies condition 4. Condition 3 is similar. Condition 5: by construction, $\sigma \models (\Phi)_\alpha$ implies that $[\![(\Phi)_\alpha, \alpha, \sigma]\!]^\gamma$ contains an edge $l(\gamma_p(\alpha), \gamma_p(\alpha^-))$, where l is either And, Or, or Lit, depending on the outermost constructor of $(\Phi)_\alpha$.

□

Lemma 4.6 *Φ is satisfiable if, and only if, there exists a set S of edges in $G(\Phi)$ that is satisfying for **result**.*

Proof Suppose Φ is satisfiable and let σ be an assignment of truth values to the variables of Φ that makes Φ true. Then, $[\![\Phi, \epsilon, \sigma]\!]^\gamma\{\gamma_p(\epsilon^-) \leftarrow$ **result**$\}$ is a satisfying graph for **result** by Lemma 4.5.

For the other direction, suppose S is a subgraph of $G(\Phi)$ which is satisfying for **result**. Define σ on the variables in Φ by $\sigma(X) = true$ if S contains an edge $Pos(\gamma_v(X), \mathbf{y})$ and $\sigma(X) = false$ otherwise. We now claim the following about σ:

1. If S contains an edge $Pos(\gamma_v(X), \mathbf{y})$ then $\sigma(X) = true$. This is just by the definition of σ.

2. If S contains an edge $Neg(\gamma_v(X), \mathbf{y})$ then $\sigma(X) = false$. This follows immediately from the previous and the fact that S cannot contain both $Pos(\mathbf{x}, \mathbf{y})$ and $Neg(\mathbf{x}, \mathbf{y})$ for any $\mathbf{x}$.

3. If S contains an edge $Lit(\gamma_p(\alpha), \gamma_p(\alpha^-))$ then the literal occurring at α is made true by σ. This derives by the fact that S is satisfying, by the fact that a Lit edge only has Pos and Neg incoming edges, and by facts (1) and (2);

4. If S contains an edge $l(\gamma_p(\alpha), \gamma_p(\alpha^-))$, then $(\Phi)_\alpha$ is made true by σ. This is an easy induction on the size of $(\psi)_\alpha$, using the previous claim for the base case $l = Lit$. In the And case, where the edge is $And(\gamma_p(\alpha), \gamma_p(\alpha^-))$, by construction $(\Phi)_\alpha = (\Phi)_{\alpha.0} \wedge (\Phi)_{\alpha.1}$, and, by Condition 4, S contains two incoming edges into $\gamma_p(\alpha)$ which, by Proposition 4.3, arrive from $\gamma_p(\alpha.0)$ and $\gamma_p(\alpha.1)$. By induction, both $(\Phi)_{\alpha.0}$ and $(\Phi)_{\alpha.1}$ are made true by σ, hence the thesis. The Or case is similar.

5. σ makes Φ true. This follows from the previous claim and the fact that S contains the edge that leads from $\gamma_p(\epsilon)$ to **result**.

□

We now wish to construct a formula of GL that is true in the graph $G(\Phi)$ exactly when Φ is satisfiable. The conditions defining a satisfying subgraph are easily described by formulas of GL. They are given as formulas below. We use them to define a formula $GSat$ defining the graphs $G(\Phi)$ which

encode satisfiable Boolean formulas.

$$
\begin{array}{lll}
\mathit{Consistent} & =_{def} & \forall x, y, z. \neg(\mathit{Pos}(x, y) \mid \mathit{Neg}(x, z) \mid \mathsf{T}) \\
\mathit{LitTrue} & =_{def} & \forall x, y. \ (\mathit{Lit}(x, y) \mid \mathsf{T}) \ \Rightarrow \ \text{In}_{\geq 1}(x) \\
\mathit{OrTrue} & =_{def} & \forall x, y. \ (\mathit{Or}(x, y) \mid \mathsf{T}) \ \Rightarrow \ \text{In}_{\geq 1}(x) \\
\mathit{AndTrue} & =_{def} & \forall x, y. \ (\mathit{And}(x, y) \mid \mathsf{T}) \ \Rightarrow \ \text{In}_{\geq 2}(x) \\
\mathit{GSat} & =_{def} & (\mathit{Consistent} \wedge \mathit{LitTrue} \wedge \mathit{OrTrue} \wedge \mathit{AndTrue} \wedge \text{In}_{\geq 1}(\textbf{result})) \mid \mathsf{T}
\end{array}
$$

This gives us the NP-completeness result.

Theorem 4.7 *The set* $\{G \ : \ G \vDash \mathit{GSat}\}$ *is NP-complete.*

Proof The formula GSat translates into an *existential* MSO sentence and hence the set of graphs it defines is in NP. The existential MSO quantifier comes from the translation of the outermost |, while the internal occurrences of | can all be translated as first-order quantifications, as detailed in Section 6. The hardness follows from the fact that the function G is a reduction from satisfiability of Boolean formulas in negation normal form. □

Note that, dual to the notion of a satisfying subgraph S of $G(\Phi)$, we can define a *falsifying* subgraph S as one such that:

1. there is no node $\mathbf{x}$ with both outgoing edges labelled Pos and outgoing edges labelled Neg in S;
2. for every edge $\mathit{Lit}(\mathbf{x}, \mathbf{y})$ in S, $\mathbf{x}$ has an incoming edge in S;
3. for every edge $\mathit{Or}(\mathbf{x}, \mathbf{y})$ in S, $\mathbf{x}$ has two incoming edges in S;
4. for every edge $\mathit{And}(\mathbf{x}, \mathbf{y})$ in S, $\mathbf{x}$ has an incoming edge in S; and
5. **result** has an incoming edge in S.

Entirely analgously to Lemma 4.6 we can show that a graph $G(\Phi)$ admits a falsifying subgraph if, and only if, there is an assignment σ that makes the formula Φ false. The change to the proof of the Lemma is that S is obtained from σ by choosing edges corresponding to the subformulas of Φ that are made *false* by σ. Thus, by altering the formulas $\mathit{AndTrue}$ and OrTrue, we define the formulas GFal and GVal.

$$
\begin{array}{lll}
\mathit{LitFalse} & =_{def} & \forall x, y. \ (\mathit{Lit}(x, y) \mid \mathsf{T}) \ \Rightarrow \ \text{In}_{\geq 1}(x) \\
\mathit{OrFalse} & =_{def} & \forall x, y. \ (\mathit{Or}(x, y) \mid \mathsf{T}) \ \Rightarrow \ \text{In}_{\geq 2}(x) \\
\mathit{AndFalse} & =_{def} & \forall x, y. \ (\mathit{And}(x, y) \mid \mathsf{T}) \ \Rightarrow \ \text{In}_{\geq 1}(x) \\
\mathit{GFal} & =_{def} & (\mathit{Consistent} \wedge \mathit{LitFalse} \wedge \mathit{OrFalse} \wedge \mathit{AndFalse} \wedge \text{In}_{\geq 1}(\textbf{result})) \mid \mathsf{T} \\
\mathit{GVal} & =_{def} & \neg \mathit{GFal}.
\end{array}
$$

That is, for any Boolean formula Φ, $G(\Phi) \vDash \mathit{GVal}$ if, and only if, Φ is valid. This gives us the following result.

Theorem 4.8 *The set* $\{G \ : \ G \vDash \mathit{GVal}\}$ *is co-NP-complete.*

We now extend the construction above to encode quantified Boolean formulas as graphs and express their validity in GL. This enables us to prove that, for each k, there exists a formula ϕ_k such that the problem: 'given G, decide whether $G \vDash^{\sigma} \phi_k$' is Π^p_k-hard. The standard hard problem for a class Π^p_k of the polynomial hierarchy is the validity of a quantified Boolean formula with k alternations of

quantifiers, i.e. a formula like the following one (where no quantifier appears in Φ and the last quantifier is $\exists$ if k is even, as we assume below, and is $\forall$ if k is odd):

$$\forall X_1^k \ldots X_{i_k}^k.\exists Y_1^{k-1} \ldots Y_{i_{k-1}}^{k-1}.\ldots \forall X_1^2 \ldots X_{i_2}^2.\exists Y_1^1 \ldots Y_{i_1}^1.\Phi.$$

We encode such formulas into graphs which have edge labels *And*, *Or*, *Pos*, *Neg*, *Lit*, and a node named **result** as before. In addition, there is, for each $1 \leq i \leq k$, a label $Switch^i$. A node $\gamma_v(X)$ has an incoming edge labelled $Switch^i$ if, and only if, X is quantified in the ith quantifier block. With variables thus marked with an index, we are able to construct a formula $GVal_k$, using alternations of $|$ and negation, to express that the formula encoded by $G(\Phi)$ is valid. The quantifier blocks are numbered from the inside out as this makes it easier to define the formulas inductively.

In defining the formulas $GVal_k$, there is one important departure from our previous definitions. The formulas $GSat$ and $GVal$ were carefully crafted to make limited use of the operator $|$. To be precise, in each occurrence of the operator $\phi \mid \psi$, the formula ψ is either atomic (of the form $\alpha(\xi, \chi)$) or it is the formula T. The reason for doing this is elaborated in Section 6. This restriction was made possible by exploiting the monotonicity of the Boolean operators $\wedge$ and $\vee$ and for this reason we assumed that the Boolean formula Φ was in negation normal form. Thus, a satisfying subgraph S of $G(\Phi)$ may not contain *all* edges corresponding to subformulas made true by an assignment σ but the existence of such an S is sufficient. In restricting to S we may discard more edges than we need to as long as S still satisfies the necessary conditions. However, when we deal with alternations of quantifiers in QBF, we must ensure, while choosing a subset of the edges representing an assignment to the variables in quantifier block i that we do not discard any of the edges corresponding to inner blocks. For this, we use the following formulas:

$$\begin{array}{lll} Discard_k & =_{def} & \forall x, y, a.(a(x,y) \mid \mathsf{T}) \Rightarrow (a = Switch^k \vee a = Pos \vee a = Neg) \\ & & \wedge \forall x, y.[((Neg(x,y) \vee Pos(x,y)) \mid \mathsf{T}) \Rightarrow \exists z.(Switch^k(z,x) \mid \mathsf{T})] \\ & & \wedge\, Consistent \\ Consistent_k & =_{def} & \forall x, y.[\neg(Switch^k(x,y) \mid \mathsf{T})] \\ & & \wedge [(Neg(x,y) \wedge Pos(x,y)) \Rightarrow \bigvee_{i<k} \exists z.(Switch^i(z,x) \mid \mathsf{T})] \end{array}$$

These formulas are best explained by thinking of constructing in stages, for decreasing values of k, a truth assignment that satisfies the formula Φ. The truth assignment is represented in the graph by discarding the edges corresponding to the literals that are made false. The formula $Discard_k$ will be true of the set of edges that are discarded at stage k. It ensures that we only discard edges corresponding to variables that are in the kth block and that we do not discard both Pos and Neg edges from the same node (the formula $Consistent$ has been defined before Theorem 4.7). Note that it requires that the edges $Switch^k$ marking the nodes are also discarded. This is no loss as they are no longer needed. The formula $Consistent_k$ should be true of the graph formed by the edges that are not discarded at stage k. It requires that we keep only Pos or only Neg edges for variables in the kth block by saying that every node with both Pos and Neg outgoing edges is marked by an edge $Switch^i$ for some $i < k$.

We can now inductively define the formulas $GValE_k$ and $GValA_k$. These are intended to express the validity of a quantified Boolean formula with k quantifier blocks starting with an existential (respectively a universal) quantifier.

$$\begin{array}{lll} GValE_1 & =_{def} & GSat \\ GValA_1 & =_{def} & GVal \\ GValE_{k+1} & =_{def} & Discard_{k+1} \mid (Consistent_{k+1} \wedge GValA_k) \\ GValA_{k+1} & =_{def} & Discard_{k+1} \mid\Rightarrow (Consistent_{k+1} \Rightarrow GValE_k)]. \end{array}$$

Lemma 4.9 *If Φ is a quantified Boolean formula with k blocks of quantifiers beginning with an existential (resp. universal) quantifier, then $G(\Phi) \models GValE_k$ (resp. $GValA_k$) if, and only if, Φ is valid.*

Proof The proof is by induction on k. We aim to prove a stronger statement as inductively we need to consider the case where Φ has free Boolean variables. For such a Φ and an assignment σ of truth values to its free variables, let $G(\Phi, \sigma)$ be the graph obtained from $G(\Phi)$ by removing all Neg edges leaving a node $\gamma_v(X)$ for all free X such that $\sigma(X) = true$ and removing all Pos edges leaving a node $\gamma_v(X)$ for all free X such that $\sigma(X) = false$; the other edges are as in $G(\Phi)$. We show that $G(\Phi, \sigma) \models GValE_k$ (resp. $GValA_k$) if, and only if, σ makes Φ true.

For the case $k = 1$, if all quantifiers are existential then $\Phi = \exists \vec{X}.\Psi$ is valid if, and only if, the quantifier-free Ψ is satisifiable by an assignment (σ, σ') that extends σ, which we claim is expressed by $G(\exists \vec{X}.\Psi, \sigma) \models GSat$. Hence, we have to prove that (a) the existence of a satisfying subgraph S of $G(\exists \vec{X}.\Psi, \sigma)$ implies satisfiability of Ψ by some (σ, σ') and (b) if $(\sigma, \sigma') \models \Psi$ then the graph $[\![\Phi, \epsilon, (\sigma, \sigma')]\!]^\gamma\{\gamma_p(\epsilon^-) \leftarrow \textbf{result}\}$ is satisfying, and it is a subgraph of $G(\exists \vec{X}.\Psi, \sigma)$. Part (b) is identical to the proof of Lemma 4.5, and the construction of $[\![\Psi, \epsilon, (\sigma, \sigma')]\!]^\gamma$ ensures the absence of the Pos and Neg edges that are removed from $G(\exists \vec{X}.\Psi)$ to obtain $G(\exists \vec{X}.\Psi, \sigma)$. To prove part (a) we consider, as in the proof of Lemma 4.6, the substitution (σ, σ') that extends σ by assigning $true$ to X_i if S contains an edge $Pos(\gamma_v(X_i), \mathbf{y})$, and $false$ otherwise. This assignment enjoys the properties listed in the proof of Lemma 4.6. Specifically, items (1) and (2) hold, by construction, on the variables assigned by σ and on the variables assigned by σ'. The other properties are proved as in Lemma 4.6. Observe that the existence of a subgraph that is satisfying is independent of the presence of $Switch^i$ nodes.

Similarly if all quantifiers are universal, $\sigma \models \Phi$ is expressed by $G(\Phi, \sigma) \models GVal$.

For the inductive case, note that $G(\Phi, \sigma) \models GValE_{k+1}$ if, and only if, $G(\Phi, \sigma) \equiv G_1 \mid G_2$ where $G_1 \models Discard_{k+1}$ and $G_2 \models Consistent_{k+1} \wedge GValA_k$. The definitions of $Discard_{k+1}$ and $Consistent_{k+1}$ ensure that for each variable Y_i^{k+1} in quantifier block $k+1$ of Φ, either all Pos edges out of $\gamma_v(X)$ are in G_1 and all Neg edges in G_2 or vice versa. Moreover, all $Switch^{k+1}$ edges are in G_1. Thus, $G_2 \equiv G(\Phi', \sigma')$ for some σ' that extends σ by assigning a truth value to each variable Y_i^k, where Φ' is the formula obtained from Φ by removing the outermost block of quantifiers. Since, by induction hypothesis, $G(\Phi', \sigma') \models GValA_k$ if, and only if, σ' makes Φ' true, the result follows. The case of $GVal_{k+1}$ is dual. □

Theorem 4.10 *For each k there exist GL formulas ϕ_k and ψ_k that characterize sets of graphs that are complete for Π_k^p and Σ_k^p, respectively.*

Proof Take ϕ_k to be $GValA_k$ and ψ_k to be $GValE_k$. □

One consequence of this result is that the alternation of $|$ with negation forms an infinite hierarchy of expressive power in GL. Hence, it is not possible to obtain a normal form similar to the conjunctive or disjunctive normal forms of boolean operators, characterized by a fixed number of alternations between $|$ and negation. While implementing the TQL optimizer, we looked hard for such a normal form; it is useful to know that this is not worth pursuing any more.

Corollary 4.11 *Unless the polynomial hierarchy collapses, the alternation of $|$ and negation forms a strict hierarchy in GL.*

5 Graph Logic With Recursion

It is instructive to compare GL_μ with other logics of recursion, such as LFP, the extension of first-order logic with an operator for forming the least fixed points of relational expressions (see [16] for an

exposition). For a relational variable R and a formula ϕ in which R only appears positively, LFP allows the expression $\mu R.\phi$ which defines the least relation $\mathcal{R}$ such that, $\mathcal{R} = [\![\phi]\!]_{\sigma;\rho,R\mapsto\mathcal{R}}$. Just as the graph composition operator of GL can be simulated by a monadic second-order quantifier one might think that recursion can be simulated by the fixed-point operator of LFP. There is, however, a crucial difference. In order to model-check an LFP-recursive sentence, a fixed-point has to be computed in the lattice of the relations on the model domain. Model-checking $G \vDash \mu R.\phi$ in GL_μ requires the computation of the fix-point of a map $\lambda S.\ [\![\phi]\!]_{\sigma;\rho,R\mapsto S}$ defined on the lattice of *sets of subgraphs* of G.[4] The evaluation of a fixed-point in LFP in a graph G amounts to finding the least fixed-point of a montone function on the lattice of k-ary relations on G. The size of this lattice is exponential in the size of G and the length of a maximal chain is bounded by n^k where n is the size of G. This is what guarantees that the fixed-point can be evaluated in a polynomial number of steps. In contrast, evaluating a fixed-point in GL_μ is to find the least fixed-point of a monotone operator on the lattice of sets of subgraphs of G. This lattice is of size doubly exponential in the size of G and has chains of length exponential in the size of G. This suggests that evaluating a fixed-point may require an exponential number of steps. This is amply illustrated by the result in Section 5.2 that exhibits a PSPACE-complete problem that is definable in the logic. This is why the result in Section 5.1 showing that the model-checking complexity of the logic is still in PSPACE is quite interesting.

5.1 Combined Complexity

The set $\{(G, \sigma, \phi)\ :\ \phi \in \mathrm{GL}_\mu,\ G \vDash^\sigma \phi\}$ is PSPACE-hard, by results in Section 4. We now exhibit a PSPACE algorithm to decide the problem, establishing a tight upper bound on its complexity. The way the algorithm deals with recursion is analogous to Winskel's algorithm for model-checking the μ-calculus [27]. The algorithm is given in Table 1. There, $get(stack, "\xi")$ returns ξ if it is a constant, and finds its associated value in $stack$ if ξ is a variable. In the line for $\alpha(\xi_1, \xi_2)$, $G[i]$ is the i-th edge of G, encoded as a triple (i, j, k), representing the indexes of the label, the first, and the second node.

We assume that no variable in the formula is bound in two distinct places. We associate a counter to each variable and a bitmask to each $|$ operator in the formula. We have a variable bitmap *mask* that specifies which edges in the graph are included in the current subgraph. To check whether $\phi \mid \psi$ holds, we let the corresponding bitmask *sm* iterate over all the submasks of the current mask *m*. For each value of *sm* we check ϕ against the subgraph identified by *sm* and ψ against its complement wrt m, $minus(m, sm)$. $\phi \mid \psi$ holds if and only if we find a value for *sm* such that both checks succeed. Actually, we cannot have a different variable for each $|$, since the algorithm is written to work for formulas of any size. Hence, *sm* is just a local variable of the procedure that checks the $|$ case, which will be automatically saved on the call stack when a subroutine is called, and restored when the subroutine exits.

To check whether $\exists x.\phi$ holds, we let the corresponding counter *x* enumerate all the names that appear either in the graph or in the formula, plus one fresh name for each variable in the formula. By Proposition 2.7, $\exists x.\phi$ holds if and only if we find a value for *x* such that ϕ model-checks. As above, we do not have a different counter for each variable *x*, but we use a stack. In this case we push a pair *"x"=x* on an explicit stack every time a quantification $\exists x$ is met. We do not use the call stack for *x* because later, to check whether $\alpha(\xi_1, \xi_2)$ holds, we will have to substitute all the variables among α, ξ_1, ξ_2 with their value, and we can retrieve those values by exploring the explicit stack.

[4]It is well-known that recursive formulas can be model-checked either by repeatedly substituting the recursion variable with its definition, Prolog-style (top-down evaluation, typical of programming languages) or by actually computing the fix-point that corresponds to the formula, Datalog-style (bottom-up evaluation, typical of deductive databases). Our informal discussion assumes the bottom-up implementation. It would be difficult to discuss top-down implementations in such general terms, since they can be based on quite different terminating conditions.

```
inputs: G, ψ; other global variables: stack, and the implicit call stack;
Evaluate(ψ,G) =
    let mask= 1_1 1_2 ... 1_n where n = sizeof(G);
    return(eval(ψ, mask));
eval(¬φ, mask) =
    return(not eval(φ,mask));
eval(0, mask) =
    return(mask== 0_1 ... 0_n);
eval(T, mask) =
    return(true);
eval(ξ_1 = ξ_2, mask) =
    return(get(stack,"ξ_1") == get(stack,"ξ_2"));
eval(α_1 = α_2, mask) = similar to previous case
eval(φ ∧ ψ, mask) =
    if (eval(φ, mask) andif eval(ψ, mask)) {return(true);}
    else {return (false);}
eval(∃x.φ, mask) =
    for i in 1..n do
        push(⟨"x" = i⟩, stack);
        if eval(φ, mask) {pop(stack); return(true);}
        pop(stack)
    return(false);
eval(∃a.φ, mask) = similar to previous case
eval(φ | ψ, mask) =
    for submask in submasks(mask) do
        if (eval(φ, submask) and eval(ψ, minus(mask,submask))) {return(true);}
    return (false);
eval(α(ξ_1, ξ_2), mask) =
    if ((mask== 0 ... 01_i 0 ... 0) and
        (G[i] == (get(stack,"α"),get(stack,"ξ_1"),get(stack,"ξ_2") ) {return(true);}
    else {return(false);}
eval(μR.ψ, mask) =
    push(⟨"R" = mask⟩, stack);
    res = eval(ψ, mask);
    pop(stack); return(res);
eval(R, mask)
    if mask== get(stack,"R") {return(false)}
    else { find μR.φ in the input formula ψ
             and return(eval(μR.φ, mask)); }
```

Table 1: The model-checking algorithm for GL_μ

To model-check $\mu R.\phi$, we first push on the stack a pair "R"-*current mask*. Then, when R is met, we first check whether the current mask is still equal to that associated to R on the stack. If the current mask is a strict subset of that stored in R, we substitute R with $\mu R.\phi$ and continue. If it is equal to that stored in R, then this branch can only loop forever,[5] hence *false* is returned.

We prove that this algorithm runs in polynomial space and that it is correct.

Theorem 5.1 Evaluate(ψ, G) *always terminates and can be executed with polynomial space.*

Proof Let n be the maximum of the number of edges and the number of nodes in G. The algorithm uses the call stack and the variable *stack*. Each recursive call pushes its local variables, the return address, and the call parameters on the call stack. The worst case is that of $|$, where we have one local variable that is n bits long ($submask$) plus the *mask* parameter that is n bits long as well. Hence, each stack frame on the call stack is linearly bounded by the input size. The same is true for the *stack* variable, where each stack frame has either n size (if it is a mask) or $\log(n)$ size, in the $\exists x/a.\phi$ cases. Moreover, each procedure call performs at most one push, and always pops what it pushed, hence the *stack* variable never contains more frames than the call stack. Hence we have only to show that the call stack growth is bounded by a polynomial. This bound implies termination as well, since all the *for* loops in the code are bounded.

Every frame in the call stack contains a bit mask. This mask is always equal to, or included in, the one of the preceding frame. Let a stack-chunk be a sequence of stack frames which all contain the same mask. The stack will always be composed by at most $n+1$ stack-chunks, where n is the size of the input graph, since $n+1$ is the length of the longest chain of n-bit masks ordered by strict inclusion. A single stack-chunk may contain two frames that correspond to the evaluation of the same R variable only if the second one is the last frame on the stack, since the first evaluation of R with mask m is followed by an evaluation of $\mu R.\psi$ which pushes $\langle \mathrm{R} = m\rangle$ on *stack*, so that the next evaluation of R with mask m returns immediately. Hence, any stack-chunk contains at most $k+1$ recursion-variable frames, if k is the number of recursive variables in ϕ. Finally, the sub-chunk included between two consecutive recursion-variable frames cannot contain more than l frames, where l is the longest path in the syntax tree of ϕ, since any other case but R walks one step down along ϕ. This gives an $O(nkl)$ bound on the number of frames of the call stack. □

Theorem 5.2 Evaluate(ψ, G) = *true if, and only if,* $G \vDash \psi$.

Proof We prove the correctness of the algorithm by presenting a proof system and showing that it is sound with respect to the semantics. We then show that the algorithm faithfully implements the proof system.

Consider the proof system of Table 2. The formula $\underline{G}$ in rule ($\nvDash \mu$) is the sentence that is only satisfied by G (up to graph-equality), defined in Lemma 2.4. $X(G, \phi, \sigma)$ is a subset of $\mathcal{X}$ such that $X \sqsupseteq (G, \phi, \sigma)$, so that, by Proposition 2.7, quantification need be checked only on $X(G, \phi, \sigma)$. σ is a substitution, i.e. an ordered sequence of pairs $x \mapsto \mathbf{x}$, $a \mapsto \mathbf{a}$, $R \mapsto \phi$, where no variable is bound twice; $\{x \mapsto c\} \cdot \sigma$ is the substitution obtained by removing any $x \mapsto c'$ pair from σ, and adding the new pair $x \mapsto c$ at the beginning of σ. $x\sigma$ is just the element that σ associates with x. $\phi \cdot \sigma$ applies all pairs in σ to ϕ *one after the other*, hence $\phi \cdot (\{x \mapsto c\} \cdot \sigma)$ is equal to $(\phi\{x \leftarrow c\}) \cdot \sigma$.

To study the correspondence between the algorithm and the proof system, we first define a mapping $\Sigma_\psi(s)$ that maps a stack s into a substitution, using ψ in the translation of bindings $(R = m)$. Essentially, a pair $(R = m)$ in the stack is translated into a mapping $R \mapsto ((\mu R.\phi) \wedge \neg(\underline{G \cap m}))$ that

[5] The fact that R is still associated with the same mask implies that the rest of the stack retains its old content as well. We do not discuss this fact here, because it is the kernel of the proof of Theorem 5.2.

expresses the termination condition that characterizes our algorithm. $G\cap m$ is the subgraph of G identified by the mask m. The empty stack becomes the empty substitution. Stacks are reversed because the last element of the stack is the first substitution to be applied.

$$
\begin{array}{ll}
\Sigma_\psi(s, R = m) = \{R \mapsto (\mu R.\phi \wedge \neg\underline{G\cap m})\} \cdot (\Sigma_\psi(s)) & \text{if } \mu R.\phi \text{ is a subterm of } \psi \\
\Sigma_\psi(s, x = \mathbf{x}) \ \ = \{x \mapsto \mathbf{x}\} \cdot (\Sigma_\psi(s)) & \\
\Sigma_\psi(s, a = \mathbf{a}) \ \ = \{a \mapsto \mathbf{a}\} \cdot (\Sigma_\psi(s)) &
\end{array}
$$

We will prove soundness of the algorithm w.r.t. the proof system, and soundness of the proof system w.r.t. satisfaction, as specified below, where $eval_{\psi,G,s}(\phi,\ mask)$ is the result of calling $eval(\phi,\ mask)$ when input formula is ψ, input graph is G, and current stack is s.

$$
\begin{array}{lll}
eval_{\psi,G,s}(\phi, m) = true & \Rightarrow\ G\cap m; \Sigma_\psi(s) \vdash \phi & (1a)\text{: soundness of } eval\text{, case } true \\
eval_{\psi,G,s}(\phi, m) = false & \Rightarrow\ G\cap m; \Sigma_\psi(s) \not\vdash \phi & (1b)\text{: soundness of } eval\text{, case } false \\
\phi \cdot \sigma \text{ is closed, } G; \sigma \vdash \phi & \Rightarrow\ G \vDash \phi \cdot \sigma & (2a)\text{: soundness of } \vdash \\
\phi \cdot \sigma \text{ is closed, } G; \sigma \not\vdash \phi & \Rightarrow\ G \vDash \neg(\phi \cdot \sigma) & (2b)\text{: soundness of } \not\vdash
\end{array}
$$

Properties $(1a)$ and $(1b)$ are easy to prove by induction on the depth of the call stack of the algorithm, and by cases. We show the cases for recursion.

$eval_{\psi,G,s}(R, m) = false$, where $R = m'$ is in s $\Rightarrow$

either $m' = m$, or $eval_{\psi,G,s}(\mu R.\phi, m) = false$	
in the second case, by induction	$G\cap m; \Sigma_\psi(s) \not\vdash \mu R.\phi$
in the first case, by $m = m'$	$G\cap m; \Sigma_\psi(s) \vdash \underline{G\cap m'}$
hence	$G\cap m; \Sigma_\psi(s) \not\vdash \neg\underline{G\cap m'}$
in both cases, we can apply rule ($\not\vdash \wedge$):	$G\cap m; \Sigma_\psi(s) \not\vdash (\mu R.\phi) \wedge \neg\underline{G\cap m'}$
$R(\Sigma_\psi(s)) = (\mu R.\phi) \wedge \neg\underline{G\cap m'}$, hence:	$G\cap m; \Sigma_\psi(s) \not\vdash R(\Sigma_\psi(s))$
by ($\not\vdash R$):	$G\cap m; \Sigma_\psi(s) \not\vdash R$

$eval_{\psi,G,s}(R, m) = true$, where $R = m'$ is in s $\Rightarrow$

$m' \neq m$ and $eval_{\psi,G,s}(\mu R.\phi, m) = true$	
by induction	$G\cap m; \Sigma_\psi(s) \vdash \mu R.\phi$
by $m \neq m'$	$G\cap m; \Sigma_\psi(s) \vdash \neg\underline{G\cap m'}$
hence	$G\cap m; \Sigma_\psi(s) \vdash (\mu R.\phi) \wedge \neg\underline{G\cap m'}$
$R(\Sigma_\psi(s)) = (\mu R.\phi) \wedge \neg\underline{G\cap m'}$, hence:	$G\cap m; \Sigma_\psi(s) \vdash R(\Sigma_\psi(s))$
by ($\vdash R$):	$G\cap m; \Sigma_\psi(s) \vdash R$

$eval_{\psi,G,s}(\mu R.\phi, m) = false \Rightarrow$

$eval_{\psi,G,(s,R=m)}(\phi, m) = false$,	
by induction	$G\cap m; \Sigma_\psi(s, R = m) \not\vdash \phi$
by definition, $\Sigma_\psi(s, R = m) = (\{R \mapsto ((\mu R.\phi) \wedge \neg\underline{G\cap m})\} \cdot \Sigma_\psi(s))$	
hence, by rule ($\not\vdash \mu$)	$G\cap m; \Sigma_\psi(s) \not\vdash \mu R.\phi$

$eval_{\psi,G,s}(\mu R.\phi, m) = true$: the proof is identical

Soundness of the proof system (property (2a) and (2b)) is proved by induction on the size of a proof, and by cases on the last rule applied. All cases are trivial but ($\not\vdash \mu$), ($\vdash \mu$), ($\not\vdash R$) and ($\vdash R$).

Cases ($\not\vdash \mu$) and ($\vdash \mu$) follow by induction once we prove that, for any pair ϕ, σ, such that $\mathrm{fv}(\phi \cdot \sigma) \subseteq \{R\}$,

$$
\begin{array}{l}
G \vDash \neg(\phi \cdot (\{R \mapsto ((\mu R.\phi) \wedge \neg\underline{G})\} \cdot \sigma)) \ \Rightarrow\ G \vDash \neg((\mu R.\phi) \cdot \sigma) \\
G \vDash (\phi \cdot (\{R \mapsto ((\mu R.\phi) \wedge \neg\underline{G})\} \cdot \sigma)) \ \Rightarrow\ G \vDash ((\mu R.\phi) \cdot \sigma)
\end{array}
$$

which are equivalent to the following, where we abbreviate $[\![\phi]\!]_{();()}$ by $[\![\phi]\!]$:

$$G \notin [\![\phi \cdot (\{R \mapsto ((\mu R.\phi) \ \wedge \ \neg \underline{G})\} \cdot \sigma)]\!] \ \Rightarrow \ G \notin [\![(\mu R.\phi) \cdot \sigma]\!] \quad (3a)$$
$$G \in [\![\phi \cdot (\{R \mapsto ((\mu R.\phi) \ \wedge \ \neg \underline{G})\} \cdot \sigma)]\!] \ \Rightarrow \ G \in [\![(\mu R.\phi) \cdot \sigma]\!] \quad (3b)$$

We will exploit the following properties, proved in [6]:

$$[\![\phi\{R \leftarrow \psi\}]\!]_{\sigma;\rho} = [\![\phi]\!]_{\sigma;\rho\{R \mapsto [\![\psi]\!]_{\sigma;\rho}\}} \quad (4a)$$
$$[\![\mu R.\phi]\!]_{\sigma;\rho} = \mathit{fixpoint}(\lambda S.\ [\![\phi]\!]_{\sigma;\rho\{R \mapsto S\}}) \quad (4b)$$
$$\text{the function } \lambda S.\ [\![\phi]\!]_{\sigma;\rho\{R \mapsto S\}} \quad (4c) \text{ is monotone in } S$$

We can rewrite $[\![\phi \cdot (\{R \mapsto ((\mu R.\phi) \ \wedge \ \neg \underline{G})\} \cdot \sigma)]\!]$ as follows, where $M = [\![(\mu R.\phi) \cdot \sigma]\!]$, and $F(\phi, \sigma)$ is defined as $\lambda S.\ [\![(\phi \cdot \sigma)]\!]_{();\{R \mapsto S\}}$

$$\begin{array}{lcl}
[\![\phi \cdot (\{R \mapsto ((\mu R.\phi) \ \wedge \ \neg \underline{G})\} \cdot \sigma)]\!] & = & \\
[\![(\phi\{R \leftarrow ((\mu R.\phi) \ \wedge \ \neg \underline{G})\}) \cdot \sigma]\!] & = & \\
[\![(\phi \cdot \sigma)\{R \leftarrow (((\mu R.\phi) \cdot \sigma) \ \wedge \ \neg \underline{G})\}]\!] & = & \text{by } (4a) \\
[\![(\phi \cdot \sigma)]\!]_{();\{R \mapsto [\![(((\mu R.\phi) \cdot \sigma) \ \wedge \ \neg \underline{G})]\!]\}} & = & \\
[\![(\phi \cdot \sigma)]\!]_{();\{R \mapsto M \setminus G\}} & = & \\
F(\phi, \sigma)(M \setminus G) & &
\end{array}$$

We have now to prove that $G \notin F(\phi, \sigma)(M \setminus G) \ \Rightarrow \ G \notin M$ $(5a)$ and $G \in F(\phi, \sigma)(M \setminus G) \ \Rightarrow \ G \in M$ $(5b)$. M is a fixed point of the monotone function $F(\phi, \sigma)$ (by (4b) and (4c)), hence $F(\phi, \sigma)(M \setminus G) \subseteq F(\phi, \sigma)(M) \subseteq M$, which immediately gives us (5b). From $G \notin F(\phi, \sigma)(M \setminus G)$ we conclude $F(\phi, \sigma)(M \setminus G) \subseteq (M \setminus G)$; M is defined as the intersection of all pre-fixpoints, of $F(\phi, \sigma)$ hence $M \subseteq F(\phi, \sigma)(M \setminus G) \subseteq (M \setminus G)$, hence $G \notin M$, c.v.d.

Cases $(\nvdash R)$ and $(\vdash R)$ follow immediately once we prove that $R \cdot \sigma = (R\sigma) \cdot \sigma$ (recall that $R \cdot \sigma$ is not just $R\sigma$). To this aim, observe that $\sigma = (\sigma' \cdot \{R \mapsto \Psi\} \cdot \sigma'')$, (with dom$(\sigma')$, $\{R\}$, and dom(σ'') mutually disjoint) and hence $R \cdot \sigma = \Psi \cdot (\sigma'')$ and, from the assumption that $R \cdot \sigma$ is closed, we deduce that fv$(\Psi) \subseteq$ dom(σ''), hence fv$(\Psi) \cap$ dom$(\sigma' \cdot \{R \mapsto \Psi\}) = \emptyset$, hence $(R\sigma) \cdot \sigma = \Psi \cdot (\sigma' \cdot \{R \mapsto \Psi\} \cdot \sigma'')$ is equal to $\Psi \cdot (\sigma'')$, c.v.d. □

5.2 Expressivity

While recursion does not take the combined complexity out of PSPACE, it adds expressive power to the logic. As a simple example, here is a formula that characterizes the graphs with an even number of edges, which is not expressible in either MSO or LFP.

$$\mu R.\ \mathbf{0} \ \vee \ \ ((\exists a, x, y.\ a(x, y)) \mid (\exists a, x, y.\ a(x, y)) \mid R\ ) \quad (3)$$

In this section we show that we can express a PSPACE-complete problem in this language. This is achieved by an encoding of quantified Boolean formulas (QBF) as graphs. The encoding is similar to the one in Section 4.2 except, in order to work with a finite set of labels, we do not have different edge labels for the different number of quantifier alternations. Instead, we have two labels, *Forall* and *Exist*, in addition to *Switch*, and the alternation of edges with these labels leading up to a node $\gamma_v(X)$ indicates the quantifier type and index of the Boolean variable X. Similarly to Section 4.2, quantified variables are guarded by an incoming Switch label, while free variables have no incoming Switch edge. The translation of a formula ψ is parametrized over a boolean assignment σ defined over fv(ψ). Quantified

$$(\vdash \wedge)\quad \frac{G;\sigma \vdash \phi \;\wedge\; G;\sigma \vdash \psi}{G;\sigma \vdash \phi \wedge \psi} \qquad (\not\vdash \wedge)\quad \frac{G;\sigma \not\vdash \phi \;\vee\; G;\sigma \not\vdash \psi}{G;\sigma \not\vdash \phi \wedge \psi} \qquad (\vdash \neg)\quad \frac{G;\sigma \not\vdash \phi}{G;\sigma \vdash \neg\phi} \qquad (\not\vdash \neg)\quad \frac{G;\sigma \vdash \phi}{G;\sigma \not\vdash \neg\phi}$$

$$(\vdash \exists)\quad \frac{\exists \mathbf{x} \in X(G,\sigma,\psi).\; G;\{x \mapsto \mathbf{x}\}\cdot\sigma \vdash \phi}{G;\sigma \vdash \exists x.\phi} \qquad (\not\vdash \exists)\quad \frac{\forall \mathbf{x} \in X(G,\sigma,\psi).\; G;\{x \mapsto \mathbf{x}\}\cdot\sigma \not\vdash \phi}{G;\sigma \not\vdash \exists x.\phi}$$

$$(\vdash \alpha(\xi,\xi'))\quad \frac{G = \alpha\sigma(\xi\sigma,\xi'\sigma)}{G;\sigma \vdash \alpha(\xi,\xi')} \qquad (\not\vdash \alpha(\xi,\xi'))\quad \frac{G \neq \sigma(\xi\sigma,\xi'\sigma)}{G;\sigma \not\vdash \alpha(\xi,\xi')} \qquad (\vdash \xi = \xi')\quad \frac{\xi\sigma = \xi'\sigma}{G;\sigma \vdash \xi = \xi'} \qquad (\not\vdash \xi = \xi')\quad \frac{\xi\sigma \neq \xi'\sigma}{G;\sigma \not\vdash \xi = \xi'}$$

$$(\vdash \mid)\quad \frac{\exists G', G''.\; G = G'|G'' \;\wedge\; G';\sigma \vdash \phi \;\wedge\; G'';\sigma \vdash \psi}{G;\sigma \vdash \phi \mid \psi}$$

$$(\not\vdash \mid)\quad \frac{\forall G', G''.\; G = G'|G'' \;\Rightarrow\; (G';\sigma \not\vdash \phi \;\vee\; G'';\sigma \not\vdash \psi)}{G;\sigma \not\vdash \phi \mid \psi}$$

$$(\vdash R)\quad \frac{G;\sigma \vdash R\sigma}{G;\sigma \vdash R} \qquad (\not\vdash R)\quad \frac{G;\sigma \not\vdash R\sigma}{G;\sigma \not\vdash R} \qquad (\vdash \mu)\quad \frac{G;(\{R \mapsto ((\mu R.\phi) \;\wedge\; \neg\underline{G})\}\cdot\sigma) \vdash \phi}{G;\sigma \vdash \mu R.\phi}$$

$$(\not\vdash \mu)\quad \frac{G;(\{R \mapsto ((\mu R.\phi) \;\wedge\; \neg\underline{G})\}\cdot\sigma) \not\vdash \phi}{G;\sigma \not\vdash \mu R.\phi}$$

Table 2: Proof system for $G \vdash \phi$ and $G \not\vdash \phi$

Figure 2: $G_\mu(\psi = (\forall X, Y.\exists Z.((Z \vee \neg W) \wedge (W \vee \neg X)) \wedge ((X \vee \neg Y) \wedge (Y \vee \neg Z))), \sigma = (W \mapsto \mathit{true}))$

variables have both Pos and Neg outgoing edges. A free variable Z has its Pos edges iff $\sigma(Z) = true$, and has its Neg edges otherwise (see Figure 2 for an example).

Formally, we define a translation $G_\mu(\Psi, \sigma, \gamma)$ which maps a QBF formula Ψ and an assignment σ of truth values to the free variables of Ψ to a graph. As in the translation $G(\Phi, \gamma)$ of Section 4.2, the translation is parametrized by a pair of functions $\gamma = (\gamma_v, \gamma_p)$ giving nodes corresponding to the variables and the occurrences in Ψ respectively. Suppose Ψ is

$$\Psi = \forall X_1^1 \ldots X_{i_1}^1 . \exists Y_1^2 \ldots Y_{i_2}^2 . \ldots \exists Y_1^n \ldots Y_{i_n}^n . \Phi \qquad \mathrm{dom}(\sigma) = \mathrm{fv}(\Psi) = \{Z_1, \ldots, Z_m\}$$

The graph $G_\mu(\Psi, \sigma, \gamma)$ is then described by the following term. The notation $G \setminus S$ denotes the graph G after the edges in S are removed.

$$\begin{aligned} G_\mu(\Psi, \sigma, \gamma) =_{def} & \ \mathit{Forall}(\mathbf{x}_0, \mathbf{x}_1) \\ & \mid \mathit{Switch}(\mathbf{x}_1, \gamma_v(X_1^1)) \mid \ldots \mid \mathit{Switch}(\mathbf{x}_1, \gamma_v(X_{i_1}^1)) \\ & \mid \mathit{Exist}(\mathbf{x}_1, \mathbf{x}_2) \\ & \quad \mid \mathit{Switch}(\mathbf{x}_2, \gamma_v(Y_1^2)) \mid \ldots \mid \mathit{Switch}(\mathbf{x}_2, \gamma_v(Y_{i_2}^2)) \\ & \quad \mid \mathit{Forall}(\mathbf{x}_2, \mathbf{x}_3) \\ & \quad\quad \mid \ldots \\ & \quad\quad\quad \mid \mathit{Exist}(\mathbf{x}_{n-1}, \mathbf{x}_n) \\ & \quad\quad\quad\quad \mid \mathit{Switch}(\mathbf{x}_n, \gamma_v(Y_1^n)) \mid \ldots \mid \mathit{Switch}(\mathbf{x}_n, \gamma_v(Y_{i_n}^n)) \\ & \quad\quad\quad\quad \mid G(\Phi, \gamma) \setminus \{\mathit{Neg}(\gamma_v(Z), \mathbf{x}) \ : \ \sigma \vDash Z\} \setminus \{\mathit{Pos}(\gamma_v(Z), \mathbf{x}) \ : \ \sigma \nvDash Z\} \end{aligned}$$

Theorem 5.3 *There exists a GL_μ formula that characterizes a set of graphs that is PSPACE-complete.*

Proof To be precise we define a formula $GVal_\mu$ that is true in a graph $G_\mu(\Psi, \sigma, \gamma)$ if, and only if, Ψ is made true by the assignment σ. The formula is a variation of the formulas constructed in Section 4.2. For this purpose, we need to introduce variants of the formulas $Discard_k$ and $Consistent_k$ that do not use k different forms of the $Switch$ label. These are defined as follows.

$$\begin{aligned} \mathit{Discard}'(z) \quad &=_{def} \quad \forall x, y, a.(a(x,y) \mid \mathsf{T}) \ \Rightarrow \ ((a = \mathit{Switch} \ \wedge \ x = z) \ \vee \ a = \mathit{Pos} \ \vee \ a = \mathit{Neg}) \\ & \qquad\quad \wedge \forall x, y.[(\mathit{Neg}(x,y) \ \vee \ \mathit{Pos}(x,y) \mid \mathsf{T}) \ \Rightarrow \ (\mathit{Switch}(z,x) \mid \mathsf{T})] \\ & \qquad\quad \wedge \mathit{Consistent} \\ \mathit{Consistent}'(z) \quad &=_{def} \quad \forall x, y.[\neg(\mathit{Switch}(z,y) \mid \mathsf{T})] \\ & \qquad\quad \wedge [(\mathit{Neg}(x,y) \ \wedge \ \mathit{Pos}(x,y)) \ \Rightarrow \ \exists z' \neq z.(\mathit{Switch}(z',x) \mid \mathsf{T})] \end{aligned}$$

Note that the formula $Discard'(z)$ is parametrized by the free variable z which is intended to ensure that at each stage, all discarded $Switch$ edges are for the same quantifier block.

The formula $GVal_\mu$ is now defined as follows.

$$\begin{aligned} \mu R. \quad & \neg(\exists x, y. \ (\mathit{Forall}(x,y) \ \vee \ \mathit{Exist}(x,y)) \mid \mathsf{T}) \ \wedge \ \mathit{GVal} \\ & \vee (\exists x, z. \ \mathit{QRoot}(x) \ \wedge \ (\mathit{Exist}(x,z) \mid (\mathit{Discard}'(z) \mid (\mathit{Consistent}'(z) \ \wedge \ R)))) \\ & \vee (\exists x, z. \ \mathit{QRoot}(x) \ \wedge \ (\mathit{Forall}(x,z) \mid (\mathit{Discard}'(z) \mid\Rightarrow (\mathit{Consistent}'(z) \ \Rightarrow \ R)))) \end{aligned}$$

where $QRoot(x)$ is defined as:

$$\mathit{QRoot}(x) =_{def} \neg \exists x'. \ (\mathit{Forall}(x', x) \ \vee \ \mathit{Exist}(x', x)) \mid \mathsf{T}$$

The formula $QRoot$ identifies the node $\mathbf{x}_i$ which corresponds to the quantifier block currently being evaluated. Thus, a graph $G_\mu(\Psi, \sigma, \gamma)$ satisfies $GVal_\mu$ if, and only if, either (1) it satisfies $GVal$ in which case all truth assignments are satisfying or (2) there is an $Exist$ edge leaving $\mathbf{x}_i$ and removing it there is

some way of consistently discarding Pos and Neg edges from the variables pointed to by $\mathbf{x}_{i+1}$ so that the resulting graph satisfies $GVal_\mu$ or (3) there is a $Forall$ edge leaving $\mathbf{x}_i$ and every consistent way of discarding Pos and Neg edges from the variables pointed to by $\mathbf{x}_{i+1}$ leaves a graph satisfying $GVal_\mu$. □

5.3 Strings

We now briefly examine the expressive power of GL$_\mu$ in the same way as we showed in Section 3.3 that GL defines exactly the regular languages. Since GL$_\mu$ can express mutually recursive equations, it can easily be shown to express context-free grammars (CFGs) and context-free languages (CFLs). However, CFLs are not closed under intersection, while GL$_\mu$ features conjunction, hence we should look for a wider class, which should at least include the closure of CFLs under finite intersection. We prove here that GL$_\mu$ can express conjunctive context-free languages, which strictly generalize the finite intersection of CFLs.

Conjunctive context-free grammars (C-CFGs) have been introduced in [22]; they generalize context-free grammars (CFGs) by adding an operation of language intersection. A context-free grammar can be defined as a finite set of mutually recursive equations (one equation for each variable) with the form:

$$X = w_1 + \ldots + w_n$$

where each w_i is generated by the grammar $w ::= \mathbf{a} \mid X \mid w; w$. A variable X denotes a component of the minimal solution of the system, $w_1 + w_2$ is language union, $\mathbf{a}$ is the singleton language $\{\mathbf{a}\}$, and $w; w'$ is language concatenation. Conjunctive CFGs are obtained by adding an operation of language intersection $\&$ inside the equations, which have now the following shape:

$$X = (w_1^1 \& \ldots \& w_{n_1}^1) + \ldots + (w_1^m \& \ldots \& w_{n_m}^m).$$

C-CFGs are strictly more expressive than CFGs. For example, the language $\{\mathbf{a}^i\mathbf{b}^i\mathbf{c}^i\}$ is not a context-free language, but can be expressed by the following grammar, where $A; X$ generates $\{\mathbf{a}^i; \mathbf{b}^j; \mathbf{c}^j\}$ and $Y; C$ generates $\{\mathbf{a}^i; \mathbf{b}^i; \mathbf{c}^j\}$ [22]:

$$\begin{array}{rcl}
S & = & (A; X) \,\&\, (Y; C) \\
A & = & \mathbf{a}; A + \epsilon \\
X & = & \mathbf{b}; X; \mathbf{c} + \epsilon \\
Y & = & \mathbf{a}; Y; \mathbf{b} + \epsilon \\
C & = & \mathbf{c}; C + \epsilon
\end{array}$$

This language is just the intersection of two CFLs; in Section 6.4 we exhibit a C-CFL which is not expressible as the intersection of any finite set of CFLs.

A C-CFG equation can be translated into a GL$_\mu$ equation, as follows; $\mathbf{a}$ and $w_1; w_2$ are translated as in Section 3.3, while X is translated as X.

$$X = (\mathsf{F}(w_1^1) \wedge \ldots \wedge \mathsf{F}(w_{n_1}^1)) \vee \ldots \vee (\mathsf{F}(w_1^m) \wedge \ldots \wedge \mathsf{F}(w_{n_m}^m))$$

It is well-known that μ-recursion can express systems of mutually recursive equations; for example, a system $X = \phi(X, Y),\ \ Y = \psi(X, Y)$ can be expressed as $\mu X.\ \phi(X, \mu Y.\ \psi(X, Y))$. Hence, every C-CFG can be translated into an equivalent GL$_\mu$ sentence, hence any C-CFL can be expressed in GL$_\mu$.

6 Linear Graph Logics

6.1 Linear Graph Logic

The high data complexity of GL_μ derives from the composition operator $\mid$ which quantifies over all subgraphs of the current graph, hence has an essentially second-order nature. In practice, however, most uses of the composition operator are limited to splitting a graph one edge at a time. That is, formulas which use the operator often fit the following pattern: $\exists_(\alpha(\xi,\xi') \wedge \cdots) \mid \phi$. We formalize this by defining a restricted version of the composition operator, denoted $\rfloor$, which we call *linear composition*. Its semantics is defined by the following rule:

$$G \vDash^\sigma \phi \rfloor \phi' \;\; \Leftrightarrow \;\; G \equiv (\mathbf{a}(\mathbf{x},\mathbf{y}) \mid G') \text{ and } \mathbf{a}(\mathbf{x},\mathbf{y}) \vDash^\sigma \phi \text{ and } G' \vDash^\sigma \phi'.$$

The operator can also be seen as a derived operator in GL by the following definition:

$$\phi \rfloor \phi' \;=_{def}\; (\phi \wedge \exists a, x, y.\ a(x,y)) \mid \phi'.$$

Essentially, this could be seen as a modal operator, parametrized by ϕ. That is, a graph G satisfies $\phi \rfloor \phi'$ if, and only if, there is an edge in G which satisfies ϕ and such that the graph obtained by removing that edge satisfies ϕ'.

We write LGL for *linear graph logic*, the language obtained from GL by replacing the $\mid$ operator by $\rfloor$ and LGL_μ for its extension with recursion. We begin by examining the expressive power and complexity of LGL.

The translation of GL into MSO given in Section 3.1 is easily adapted to show that LGL translates into *first-order* logic. In order to do this, we need to eliminate all uses of the second-order variables from that translation.

Towards this end we define three translations, each of them transforming a GL formula about a graph $G = (E, \mathsf{edge})$ into a FO formula about a structure $G^+ = (A, X, E, \mathsf{edge})$ (we are here identifying the function edge in G with its relational encoding in G^+). $[\![\phi]\!]^{\mathbf{0}}$ holds in G^+ iff ϕ holds in the empty graph; $[\![\phi]\!]^*_e$, where e is one edge variable, holds in G^+ iff ϕ holds in the the graph G restricted to the only edge e; $[\![\phi]\!]^F$, where F is a set of edge variables, holds in G^+ iff ϕ holds in G after the edges denoted by F have been removed from G (see the statement of Lemma 6.1; F denotes the *forbidden* edges). Hence, for example, $[\![\phi \rfloor \phi']\!]^F$ holds if there exists one edge e that can be removed from $G \setminus F$ such that both $[\![\phi]\!]^*_e$ and $[\![\phi']\!]^{F\cup\{e\}}$ hold.

The translation is defined by induction. The operators $\neg$, $\wedge$, $\exists$ and $=$ are mapped to themselves by all of the three translations, hence we only report the $[\![_]\!]^F$ case for them. The notation $\exists e \notin F.\phi$ abbreviates the first-order formula $\exists e.(\bigwedge_{f\in F} e \neq f) \wedge \phi$. The notation $\forall e \notin F.\phi$ abbreviates $\forall e.(\bigwedge_{f\in F} e \neq f) \Rightarrow \phi$.

$$
\begin{array}{lll}
[\![\neg\phi]\!]^F =_{def} \neg[\![\phi]\!]^F & [\![\phi \wedge \phi']\!]^F =_{def} [\![\phi]\!]^F \wedge [\![\phi']\!]^F & \\
[\![\exists x.\phi]\!]^F =_{def} \exists x.\ [\![\phi]\!]^F & [\![\exists a.\phi]\!]^F =_{def} \exists a.\ [\![\phi]\!]^F & \\
[\![\xi = \xi']\!]^F =_{def} \xi = \xi' & [\![\alpha = \alpha']\!]^F =_{def} \alpha = \alpha' & \\
\multicolumn{3}{l}{[\![\alpha(\xi,\xi')]\!]^F =_{def} \exists e \notin F.\ [\mathsf{edge}(e,\alpha,\xi,\xi') \wedge \forall e' \notin F.\ e = e']} \\
\multicolumn{3}{l}{[\![\mathbf{0}]\!]^F =_{def} \neg\exists e.\ e \notin F} \\
\multicolumn{3}{l}{[\![\phi \rfloor \phi']\!]^F =_{def} \exists e \notin F.\ [\![\phi]\!]^*_e \wedge [\![\phi']\!]^{F\cup\{e\}}} \\
[\![\alpha(\xi,\xi')]\!]^*_e =_{def} \mathsf{edge}(e,\alpha,\xi,\xi') & [\![\mathbf{0}]\!]^*_e =_{def} \mathsf{F} & [\![\phi \rfloor \phi']\!]^*_e =_{def} [\![\phi]\!]^*_e \wedge [\![\phi']\!]^{\mathbf{0}} \\
[\![\alpha(\xi,\xi')]\!]^{\mathbf{0}} =_{def} \mathsf{F} & [\![\mathbf{0}]\!]^{\mathbf{0}} =_{def} \mathsf{T} & [\![\phi \rfloor \phi']\!]^{\mathbf{0}} =_{def} \mathsf{F}
\end{array}
$$

The next lemma implies that, for any sentence ϕ, $(A, X, E, \mathsf{edge}) \vDash_{FO} \phi \;\Leftrightarrow\; (E, \mathsf{edge}) \vDash_{GL} [\![\phi]\!]^{\emptyset}$.

Lemma 6.1 *For any formula ϕ, substitution σ defined on all free variables of ϕ, for any $G = (E, \textsf{edge})$ and $G^+ = (A, X, E, \textsf{edge})$ such that $X \sqsupseteq (G, \phi, \sigma)$, $A \sqsupseteq (G, \phi, \sigma)$, for any set of edge variables $F \cup \{e\}$, for any $\mathbf{F} \subseteq E$, $\mathbf{e} \in E$, the following equivalences hold, where $\textsf{edge} \setminus \mathbf{F}$ removes the edges in $\mathbf{F}$ from the domain of $\textsf{edge}$, $(E, \textsf{edge}) \setminus \mathbf{F} = (E \setminus \mathbf{F}, \textsf{edge} \setminus \mathbf{F})$, and, similarly, $(E, \textsf{edge}) \cap \mathbf{e}$ intersects both E and $\textsf{edge}$ with $\{\mathbf{e}\}$.*

$$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi]\!]^F \quad \Leftrightarrow \quad G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \phi$$
$$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\phi]\!]^*_e \quad \Leftrightarrow \quad G \cap \mathbf{e} \vDash_{GL}^{\sigma} \phi$$
$$G^+ \vDash_{FO}^{\sigma} [\![\phi]\!]^{\mathbf{0}} \quad \Leftrightarrow \quad \mathbf{0} \vDash_{GL}^{\sigma} \phi$$

Proof By induction and by cases. We omit cases that are trivial or very similar to those we present.

$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi \rfloor \phi']\!]^F \Leftrightarrow G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} \exists e \notin F.\ [\![\phi]\!]^*_e \wedge [\![\phi']\!]^{F \cup \{e\}}$
$\Leftrightarrow$ There exists $\mathbf{e} \in E \setminus \mathbf{F}$ such that $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}, e \mapsto \mathbf{e}} [\![\phi]\!]^*_e$ and $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}, e \mapsto \mathbf{e}} [\![\phi']\!]^{F \cup \{e\}}$
$\Leftrightarrow$ There exists $\mathbf{e} \in E \setminus \mathbf{F}$ such that $G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\phi]\!]^*_e$ and $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}, e \mapsto \mathbf{e}} [\![\phi']\!]^{F \cup \{e\}}$
(Since no variable in F appears in $[\![\phi]\!]^*_e$.)
$\Leftrightarrow$ There exists $\mathbf{e} \in E \setminus \mathbf{F}$ such that $G \cap \mathbf{e} \vDash_{GL}^{\sigma} \phi$ and $G \setminus \mathbf{F} \setminus \{\mathbf{e}\} \vDash_{GL}^{\sigma} \phi'$
$\Leftrightarrow$ There exists $\mathbf{e} \in E \setminus \mathbf{F}$ such that $(G \setminus \mathbf{F}) \cap \mathbf{e} \vDash_{GL}^{\sigma} \phi$ and $(G \setminus \mathbf{F}) \setminus \{\mathbf{e}\} \vDash_{GL}^{\sigma} \phi'$
$\Leftrightarrow$ $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \phi \rfloor \phi'$

$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\alpha(\xi, \xi')]\!]^F \Leftrightarrow G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} \exists e \notin F.\ [\textsf{edge}(e, \alpha, \xi, \xi') \wedge \forall e' \notin F.\ e = e']$
$\Leftrightarrow$ There exists $\mathbf{e} \in (E \setminus \mathbf{F})$ such that $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}, e \mapsto \mathbf{e}} \textsf{edge}(e, \alpha, \xi, \xi')$ and forall $\mathbf{e}' \in (E \setminus \mathbf{F})$ $\mathbf{e} = \mathbf{e}'$
$\Leftrightarrow$ $(\mathbf{e}, \sigma\alpha, \sigma\xi, \sigma\xi') \in \textsf{edge}$ and $E \setminus \mathbf{F} = \{\mathbf{e}\}$
$\Leftrightarrow$ $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \alpha(\xi, \xi')$

$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\mathbf{0}]\!]^F \Leftrightarrow G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} \neg \exists e.\ e \notin F \Leftrightarrow$ There does not exist $\mathbf{e} \in (E \setminus \mathbf{F}) \Leftrightarrow G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \mathbf{0}$

$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\neg \phi]\!]^F \Leftrightarrow G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} \neg [\![\phi]\!]^F$
$\Leftrightarrow$ Not $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi]\!]^F \Leftrightarrow$ Not $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \phi \Leftrightarrow G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \neg \phi$

$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi \wedge \phi']\!]^F \Leftrightarrow G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi]\!]^F \wedge [\![\phi']\!]^F$
$\Leftrightarrow$ $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi]\!]^F$ and $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\phi']\!]^F$
$\Leftrightarrow$ $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \phi$ and $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \phi'$
$\Leftrightarrow$ $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \phi \wedge \phi'$

$G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} [\![\exists x. \phi]\!]^F \Leftrightarrow G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}} \exists x. [\![\phi]\!]^F$
$\Leftrightarrow$ There exists $\mathbf{x}$ in X s.t. $G^+ \vDash_{FO}^{\sigma, F \mapsto \mathbf{F}, x \mapsto \mathbf{x}} [\![\phi]\!]^F$
$\Leftrightarrow$ There exists $\mathbf{x}$ in X s.t. $G \setminus \mathbf{F} \vDash_{GL}^{\sigma, x \mapsto \mathbf{x}} \phi$
$\Leftrightarrow$ There exists $\mathbf{x}$ in $\mathcal{X}$ s.t. $G \setminus \mathbf{F} \vDash_{GL}^{\sigma, x \mapsto \mathbf{x}} \phi$. (Reasoning as in the proof of Proposition 2.7)
$\Leftrightarrow$ $G \setminus \mathbf{F} \vDash_{GL}^{\sigma} \exists x. \phi$

$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\phi \rfloor \phi']\!]^*_e \Leftrightarrow G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\phi]\!]^*_e \wedge [\![\phi']\!]^{\mathbf{0}}$
$\Leftrightarrow$ $G \cap \mathbf{e} \vDash_{GL}^{\sigma} \phi$ and $\mathbf{0} \vDash_{GL}^{\sigma} \phi' \Leftrightarrow G \cap \mathbf{e} \vDash_{GL}^{\sigma} \phi \rfloor \phi'$

$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\alpha(\xi, \xi')]\!]^*_e \Leftrightarrow G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} \textsf{edge}(e, \alpha, \xi, \xi')$
$\Leftrightarrow$ $(\mathbf{e}, \sigma\alpha, \sigma\xi, \sigma\xi') \in \textsf{edge} \Leftrightarrow G \cap \mathbf{e} \vDash_{GL}^{\sigma} \alpha(\xi, \xi')$

$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\mathbf{0}]\!]^*_e \Leftrightarrow G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} \mathsf{F} \Leftrightarrow G \cap \mathbf{e} \vDash_{GL}^{\sigma} \mathbf{0}$, with $G = (E, \textsf{edge})$ and $\mathbf{e} \in E$,

$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\phi \rfloor \phi']\!]^{\mathbf{0}} \Leftrightarrow G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} \mathsf{F} \Leftrightarrow \mathbf{0} \vDash_{GL}^{\sigma} \phi \rfloor \phi'$

$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\alpha(\xi, \xi')]\!]^{\mathbf{0}} \Leftrightarrow G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} \mathsf{F} \Leftrightarrow \mathbf{0} \vDash_{GL}^{\sigma} \alpha(\xi, \xi')$

$G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} [\![\mathbf{0}]\!]^{\mathbf{0}} \Leftrightarrow G^+ \vDash_{FO}^{\sigma, e \mapsto \mathbf{e}} \mathsf{T} \Leftrightarrow \mathbf{0} \vDash_{GL} \mathbf{0}$

$\square$

The translation establishes an upper bound on the expressive power of LGL. Any property that is expressible in first-order logic is decidable in LogSpace and therefore this is still true for LGL.

Proposition 6.2 *For every sentence ϕ of LGL, $\mathcal{G}_\phi = \{G \,:\, G \vDash \phi\}$ is in LogSpace.*

On the other hand, the model-checking complexity of LGL is still PSPACE-complete. This follows from the observation that the proof of PSPACE-hardness of GL model checking in Section 4.1 does not use the composition operator $|$ at all.

Proposition 6.3 *The model-checking problem for LGL is PSPACE-complete.*

6.2 Linear Graph Logic and Strings

Just as GL is as expressive as MSO on string graphs, we show that on these graphs LGL has the same expressive power as FO, being able to express any star-free language.

Star-free languages are those regular languages that are denoted by expressions in the following grammar, where $\mathbf{a}$ belongs to a fixed alphabet A, and $\overline{r}$ denotes the complement of r with respect to A^*:

$$r \;::=\; \epsilon \;|\; \mathbf{a} \;|\; r; r \;|\; r + r \;|\; \overline{r}$$

Star-free languages are exactly the languages that can be described by FO over ordered structures [20]. The encoding we provided in Section 3.3 cannot be reused here, since concatenation was translated using the full power of the $|$ operator. However, it is easy to adapt the construction in [24] to prove that every star-free language can also be described by a term of the following grammar:

$$r \;::=\; \epsilon \;|\; \mathbf{a}; r \;|\; r + r \;|\; \overline{r}$$

These expressions can be translated to LGL as follows. We parametrize here the translation with respect to the first node "x" in the graph, which simplifies case $\mathbf{a}; r$; this could not be done in Section 3.3 because of the $*$ case.

$$\begin{array}{rcl}
\mathsf{F}(\epsilon, x) & =_{def} & \mathbf{0} \\
\mathsf{F}(\mathbf{a}; r', x) & =_{def} & \exists y.\, \mathbf{a}(x, y) \rfloor \mathsf{F}(r', y) \\
\mathsf{F}(r + r', x) & =_{def} & \mathsf{F}(r, x) \vee \mathsf{F}(r', x) \\
\mathsf{F}(\overline{r}, x) & =_{def} & \neg \mathsf{F}(r, x)
\end{array}$$

The following lemma establishes the correctness of the translation.

Lemma 6.4 *For any expression r over the alphabet A, $w \in A^*$, $\vec{\mathbf{x}} = \mathbf{x}_1 \ldots \mathbf{x}_{n+1}$ with $n = |w|$, x :*

$G(w, \vec{\mathbf{x}}) \vDash^{x \mapsto \mathbf{x}_1} \mathsf{F}(r, x)$ *if, and only if,* $w \in L(r)$.

Proof The proof is by induction on the structure of the regular expression r and by cases. All cases are trivial. $\square$

6.3 Recursive Linear Graph Logic

Given that LGL is no more expressive than first-order logic, one might be tempted to think that its extension with recursion LGL_μ could be translated into LFP, the extension of first-order logic with a least fixed-point operator. However, a simple example shows that LGL_μ can express properties that are not definable in LFP. Consider the following formula.

$$\mu R.\,(\mathbf{0} \vee (\mathsf{T} \rfloor \mathsf{T} \rfloor R))$$

This formula expresses that a graph has an even number of edges. We know this property is not definable in LFP.

Though the length of inductions in LGL_μ may be bounded by the size of the graph G on which they are evaluated, there are two significant ways in which these inductions are different from the fixed-points definable in LFP. For one, the recursion variables in LGL_μ range over sets of subgraphs of G, so the number of possible evaluations is still exponentially larger than for the recursion variables in LFP which range over subsets of G. Secondly, the operator $\rfloor$ affords a kind of nondeterministic choice. The particular edge that is chosen is not determined and may violate symmetries of the graph. This property is crucially used in the definition of evenness given above.

Indeed, it turns out that the local nondeterminism afforded by the $\rfloor$ operator, combined with inductions of polynomial length is sufficient to express properties that are even PSPACE-complete. We show this by establishing that a limited form of the $|$ operator is definable in LGL_μ, namely when the $|$ is only applied to a pair of formulas where one of the formulas is T.

Lemma 6.5 *If ϕ is a formula of GL that is equivalent to a formula of LGL_μ, then the formula $\phi \mid \mathsf{T}$ also has an equivalent formula in LGL_μ.*

Proof If ϕ' is the formula of LGL_μ that is equivalent to ϕ, then $\phi \mid \mathsf{T}$ is equivalent to

$$\mu R.(\phi' \vee (\mathsf{T}\rfloor R)).$$

□

We now observe that, in the definition of the formula $GSat$ in Section 4.2, the only uses of the $|$ operator are linear (i.e. they can be equivalently replaced by $\rfloor$) or they are of the form $\phi \mid \mathsf{T}$. This gives us the following lemma.

Lemma 6.6 *There is a formula of LGL_μ equivalent to $GSat$.*

The following theorem is now immediate from the lemma.

Theorem 6.7 *There is a formula ϕ of LGL_μ such that the set $\mathcal{G}_\phi = \{G \,:\, G \vDash \phi\}$ is NP-complete.*

As the logic LGL_μ is closed under negation, we also have the following corollary.

Corollary 6.8 *There is a formula ϕ of LGL_μ such that the set $\mathcal{G}_\phi = \{G \,:\, G \vDash \phi\}$ is co-NP-complete.*

We can do better, however. The formula $GVal_\mu$ in Section 5.2 cannot be expressed directly in LGL_μ because the coherence of the set of edges that is discarded at each stage is expressed as:

$$\ldots \vee (\exists x, z.\ \mathit{QRoot}(x) \wedge (\mathit{Exist}(x,z) \mid (\mathit{Discard}'(z) \mid (\mathit{Consistent}'(z) \wedge R))))$$
$$\ldots \vee (\exists x, z.\ \mathit{QRoot}(x) \wedge (\mathit{Forall}(x,z) \mid (\mathit{Discard}'(z) \mid\Rightarrow (\mathit{Consistent}'(z) \Rightarrow R))))$$

where the $|$ between $\mathit{Discard}'(z)$ and $(\mathit{Consistent}'(z) \wedge R)$ is crucially non-linear, and cannot be substituted by $\mathsf{T} \mid (\mathit{Consistent}'(z) \wedge R)$ or $\mathsf{T} \mid\Rightarrow (\mathit{Consistent}'(z) \Rightarrow R)$, since the subgraph discarded as T may otherwise contain a non consistent set of edges, or edges which should not be discarded at the current stage. We can solve this problem by adopting a different encoding, where the coherence of the discarded part can be tested by looking, essentially, at the non-discarded part alone.

We have first to ensure that, for each variable, a single Pos is discarded iff all Pos edges are discarded, and similarly for Neg. Toward this aim, instead of having all Pos edges of each X starting from the same node, we link all of them along a linear list, labelled by occ and delimited by *start* and *end* edges, labelled occ^s and occ^e, as follows (Figure 3) (this encoding is not minimal, but is designed to be easy to reason about):

$$\begin{array}{l} occ^s(\mathbf{x}, \mathbf{x}') \mid occ(\mathbf{x}', \mathbf{x}_1) \mid Pos(\mathbf{x}_1, \gamma_p(\alpha_1)) \mid \ldots \\ \mid occ(\mathbf{x}_{n-1}, \mathbf{x}_n) \mid Pos(\mathbf{x}_n, \gamma_p(\alpha_n)) \mid occ^e(\mathbf{x}_n, \mathbf{x}'') \end{array}$$

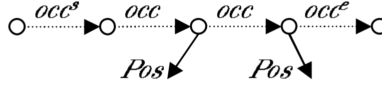

Figure 3: An occurrence list

Now, it is easy to require the integrity of such structures, hence ensuring that either all Pos edges for a variable are removed, or none of them is.

We use hereafter the operator $\phi \rfloor\Rightarrow \psi$, defined as $\neg(\phi \rfloor \neg\psi)$, i.e., if any edge satisfying ϕ is removed, then the rest of the graph satisfies ψ. For readability, we abbreviate $(\exists z.\ \alpha(\xi, z))$, where z is a fresh variable, with $\alpha(\xi, _)$, and similarly for $\alpha(_, \xi)$. $\alpha/\beta(\xi, \chi)$ abbreviates $\alpha(\xi, \chi) \vee \beta(\xi, \chi)$, so that the first line below abbreviates $((\exists z.\ occ^s(z, x)) \vee (\exists z.\ occ(z, x))) \rfloor\Rightarrow (((\exists z.\ occ(x, z)) \vee (\exists z.\ occ^e(x, z))) \mid \mathsf{T})$. The condition below accounts for both Pos and Neg lists.

The four lines can be read as: (1) any occ^s or occ edge is followed by a occ or an occ^e edge; (2) any occ or occ^e edge is preceded by a occ or an occ^s edge; (3) any occ edge has it associated Pos/Neg edge and (4) vice versa. If a graph contains a set of occurrence lists, for any of these lists, any subgraph that satisfies this predicate either contains the whole list with all of its Pos/Neg edges or it contains no piece of the list, hence none of the associated Pos/Neg edges.

$$\begin{array}{rl} OccOK = \forall x. & occ^s/occ(_, x) \rfloor\Rightarrow ((occ/occ^e)(x, _) \mid \mathsf{T}) \\ & \wedge (occ/occ^e(x, _) \rfloor\Rightarrow (occ^s/occ(_, x) \mid \mathsf{T})) \\ & \wedge (occ(_, x) \rfloor\Rightarrow (Pos/Neg(x, _) \mid \mathsf{T})) \\ & \wedge (Pos/Neg(x, _) \rfloor\Rightarrow (occ(_, x) \mid \mathsf{T})) \end{array}$$

For any quantifier block, we build a similar list of all the variables in the corresponding block; there is an intermediate node $\mathbf{y}_i$ for each variable of that block, and the Neg and Pos occurrence lists for that variable both start from that node. The top-down evaluation of the sentence proceeds by peeling the quantifier blocks off, starting from the outermost. At the beginning every variable has both its Neg and Pos lists. When a quantifier block is evaluated we discard half of these lists. To ensure an orderly execution of this procedure, every quantifier block is provided with two linear lists, labelled by var and $dvar$, which go through the same nodes $\mathbf{y}$, $\mathbf{y}'$, $\mathbf{y}_i$, and $\mathbf{y}''$, and each intermediate node $\mathbf{y}_i$ starts both the Pos and the Neg occurrence lists of the corresponding variable. Hence, for every quantifier block we build the following structure (where the first edge may be either a Exist or a Forall; this is depicted in the upper part of Figure 4, where the var and the $dvar$ lists are represented by vertical dotted edges,

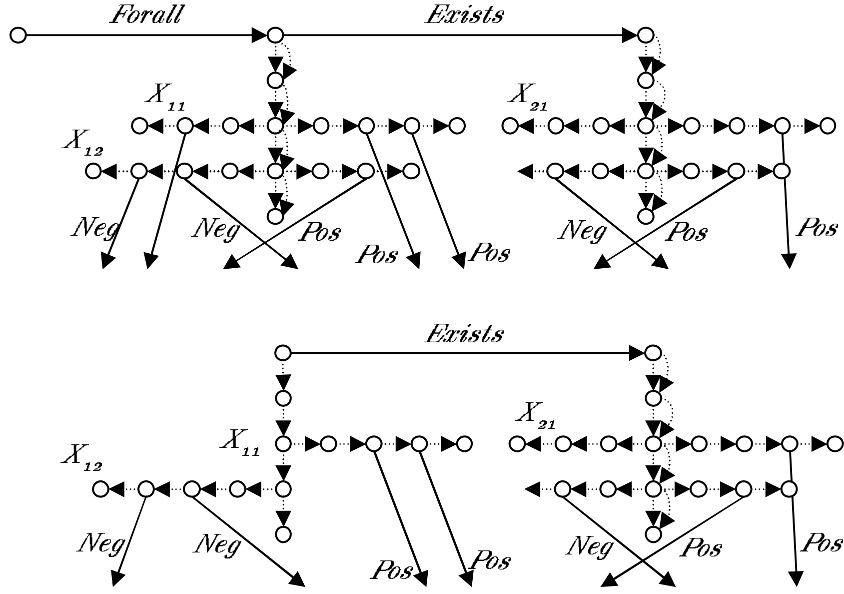

Figure 4: Coherence before and after the removal of a quantifier edge

and the occ lists are represented by horizontal dotted edges).

$$\begin{array}{l} \mathit{Exist}(\mathbf{z},\mathbf{y}) \mid \mathit{var}^s(\mathbf{y},\mathbf{y}') \mid \mathit{dvar}^s(\mathbf{y},\mathbf{y}') \\ \mid \mathit{var}(\mathbf{y}',\mathbf{y}_1) \mid \mathit{dvar}(\mathbf{y}',\mathbf{y}_1) \mid \mathit{occ}^s(\mathbf{y}_1,\ldots) \mid \mathit{occ}^s(\mathbf{y}_1,\ldots) \mid \ldots \\ \mid \ldots \\ \mid \mathit{var}(\mathbf{y}_{n-1},\mathbf{y}_n) \mid \mathit{dvar}(\mathbf{y}_{n-1},\mathbf{y}_n) \mid \mathit{occ}^s(\mathbf{y}_n,\ldots) \mid \mathit{occ}^s(\mathbf{y}_n,\ldots) \mid \ldots \\ \mid \mathit{var}^e(\mathbf{y}_n,\mathbf{y}'') \mid \mathit{dvar}^e(\mathbf{y}_n,\mathbf{y}'') \end{array}$$

Before evaluating a quantifier block, its $dvar$ list has to be present and intact, and every intermediate node $\mathbf{y}_i$ has to start two occurrence lists. After a quantifier block has been discarded, the corresponding $dvar$ list must completely disappear, and wherever a Pos list starts no Neg list must start. However, the var edges will not be discarded, and they will ensure that at least one among the Pos and Neg lists will stay. We first describe the integrity condition for the $dvar$ variable list. The first two conditions ensure the integrity of the list. The third and fourth force the $dvar$ list to stay or to go when the corresponding quantifier block stays or goes. The last two conditions specify that one variable has both its occurrence lists if, and only if, the $dvar$ list that corresponds to its quantifier block is still in place (Figure 4).

$$\begin{array}{rl} \mathit{DVarOK} = \forall x. & \mathit{dvar}^s/\mathit{dvar}(_,x) \rfloor\Rightarrow ((\mathit{dvar}/\mathit{dvar}^e)(x,_) \mid \mathsf{T}) \\ & \wedge\,(\mathit{dvar}/\mathit{dvar}^e(x,_) \rfloor\Rightarrow (\mathit{dvar}^s/\mathit{dvar}(_,x) \mid \mathsf{T})) \\ & \wedge\,(\mathit{dvar}^s(x,_) \rfloor\Rightarrow \mathit{Forall}/\mathit{Exist}(_,x) \mid \mathsf{T}) \\ & \wedge\,(\mathit{Forall}/\mathit{Exist}(_,x) \rfloor\Rightarrow \mathit{dvar}^s(x,_) \mid \mathsf{T}) \\ & \wedge\,(\mathit{dvar}(_,x) \rfloor\Rightarrow (\mathit{occ}^s(x,_) \rfloor (\mathit{occ}^s(x,_) \mid \mathsf{T})) \\ & \wedge\,(\mathit{occ}^s(x,_) \rfloor\Rightarrow \mathit{occ}^s(x,_) \rfloor\Rightarrow (\mathit{dvar}(_,x) \mid \mathsf{T})) \end{array}$$

We now describe the integrity condition of the var variable list, which ensures the presence of at

least one of the two occurrence lists for any traversed node.

$$\begin{array}{rl} \mathit{VarOK} = \forall x. & \mathit{var}^s/\mathit{var}(_,x) \rfloor\!\Rightarrow ((\mathit{var}/\mathit{var}^e)(x,_) \mid \mathsf{T}) \\ & \wedge (\mathit{var}/\mathit{var}^e(x,_) \rfloor\!\Rightarrow (\mathit{var}^s/\mathit{var}(_,x) \mid \mathsf{T})) \\ & \wedge (\mathit{var}(_,x) \rfloor\!\Rightarrow (\mathit{occ}^s(x,_) \mid \mathsf{T})) \\ & \wedge (\mathit{occ}^s(x,_) \rfloor\!\Rightarrow (\mathit{var}(_,x) \mid \mathsf{T})) \end{array}$$

We introduce now the abbreviation $\phi \rfloor^* \psi =_{def} \mu R.\, \psi \vee (\phi \rfloor R)$ which means: there exists a set of edges G_1, each satisfying ϕ, such that $G \setminus G_1$ satisfies ψ:

$$G \vDash \phi \rfloor^* \psi \;\Leftrightarrow\; \exists G_1, G_2.\, G_1 \mid G_2 \equiv G \,\wedge\, (\forall \mathbf{e} \in G_1.\, G_1 \cap \mathbf{e} \vDash \phi) \,\wedge\, G_2 \vDash \psi$$

For example, as already observed, $\mathsf{T} \rfloor^* \psi$ is equivalent to $\psi \mid \mathsf{T}$. We also use the dual abbreviation $\phi \rfloor^*\!\Rightarrow \psi =_{def} \neg(\phi \rfloor^* \neg\psi)$ characterized by the dual property:

$$G \vDash \phi \rfloor^*\!\Rightarrow \psi \;\Leftrightarrow\; \forall G_1, G_2.\, G_1 \mid G_2 \equiv G \;\Rightarrow\; (\forall \mathbf{e} \in G_1.\, G_1 \cap \mathbf{e} \vDash \phi) \;\Rightarrow\; G_2 \vDash \psi.$$

The operator $\phi \rfloor^* \psi$ is still less expressive than $\phi \mid \psi$, since G_1 can only be examined edge by edge, which is a very strong limitation. Nevertheless, its use will simplify the expression of our sentence.

We are now ready to express validity of the encoding of any QBF formula. Informally, at each iteration step, one of the following conditions holds:

1. no quantification is left and the QBF formula is valid
2. an existential quantification is outermost and, after removing its edge and the corresponding $dvar$ list, one can find a way of choosing half of the corresponding occ occurence lists, so that, after their removal, what remains is still valid
3. a universal quantification is outermost and, after removing its edge, the corresponding $dvar$ list, and half of the corresponding occ occurence lists, what remains is valid, independently of the removed occ lists, provided that the remaining graph is consistent.

$$\begin{array}{lll} \mathit{Disposable} & =_{def} & \exists x, y.\ \mathit{occ}^s/\mathit{occ}/\mathit{occ}^e/\mathit{dvar}^s/\mathit{dvar}/\mathit{dvar}^e/\mathit{Pos}/\mathit{Neg}(x,y) \\ \mathit{Consistent}'' & =_{def} & \mathit{OccOK} \wedge \mathit{VarOK} \wedge \mathit{DVarOK} \\ \mathit{GVal}_{L\mu} & =_{def} & \mu R.\, (\neg(\exists x, y.\, (\mathit{Forall}/\mathit{Exist}(x,y)) \mid \mathsf{T}) \wedge \mathit{GVal}) \\ & & \vee (\exists x.\, \mathit{QRoot}(x) \wedge (\mathit{Exist}(x,_) \rfloor (\mathit{Disposable} \rfloor^* (\mathit{Consistent}'' \wedge R)))) \\ & & \vee (\exists x.\, \mathit{QRoot}(x) \wedge (\mathit{Forall}(x,_) \rfloor (\mathit{Disposable} \rfloor^*\!\Rightarrow (\mathit{Consistent}'' \Rightarrow R)))) \end{array}$$

Hence, we have the following theorem.

Theorem 6.9 *There exists an LGL$_\mu$ formula that characterizes a set of graphs that is PSPACE-complete.*

6.4 Recursive Linear Graph Logic and Strings

We prove here that LGL$_\mu$ can express every conjunctive linear context-free language (CL-CFLs), which is a class that includes regular language, some non-regular languages, and even some languages that cannot be expressed by any CFG.

A conjunctive linear CFG is a conjunctive CFG, as defined in Section 5.3, where no word w^i_j contains more than one recursion variable [22]. Such grammars admit a normal form where every word w^i_j

has either the shape $\mathbf{a}; X; \mathbf{b}$ or the shape $\mathbf{a}$. The logic LGL$_\mu$ can express mutual recursion, union, and intersection as seen in Section 5.3, and linear language concatenation $\mathbf{a}; X; \mathbf{b}$ can be expressed as

$$\exists x, y, x', y'.\ \mathbf{a}(x, y) \rfloor \mathbf{b}(y', x') \rfloor ((\mathbf{0} \vee (\text{Initial}(y) \wedge \text{Final}(y'))) \wedge X)$$

CL-CFGs include right-linear context-free grammars as a special case, hence LGL$_\mu$ can express every regular language, but their expressive power is greater. For example, the grammar $S ::= \epsilon \mid \mathbf{a}S\mathbf{b}$ is linear, and denotes the language $\mathbf{a}^n\mathbf{b}^n$ which is not regular. The grammar

$$\begin{array}{lcl} S & = & C \,\&\, D \\ C & = & \mathbf{a}; C; \mathbf{a} + \mathbf{a}; C; \mathbf{b} + \mathbf{b}; C; \mathbf{a} + \mathbf{b}; C; \mathbf{b} + \mathbf{c} \\ D & = & (\mathbf{a}; A) \,\&\, (\mathbf{a}; D) + (\mathbf{b}; B) \,\&\, (\mathbf{b}; D) + \mathbf{c}; E \\ A & = & \mathbf{a}; A; \mathbf{a} + \mathbf{a}; A; \mathbf{b} + \mathbf{b}; A; \mathbf{a} + \mathbf{b}; A; \mathbf{b} + \mathbf{c}; E; \mathbf{a} \\ B & = & \mathbf{a}; B; \mathbf{a} + \mathbf{a}; B; \mathbf{b} + \mathbf{b}; B; \mathbf{a} + \mathbf{b}; B; \mathbf{b} + \mathbf{c}; E; \mathbf{b} \\ E & = & \mathbf{a}; E + \mathbf{b}; E + \epsilon \end{array}$$

denotes the language $\{w; \mathbf{c}; w \ :\ w \in \{\mathbf{a}; \mathbf{b}\}^*\}$ which is not context-free, and cannot be expressed as the intersection of any finite set of CFLs [22].

We do not know whether LGL$_\mu$ can express any language beyond this family.

6.5 GL$_\mu$ vs. LGL$_\mu$

The proof of Theorem 6.9, and the possibility to define the operators $\phi \mid \mathsf{T}$ and $\phi \rfloor^* \psi$ in LGL$_\mu$, seem to indicate that the expressive power of GL$_\mu$ and LGL$_\mu$ are very similar. Even in the case of strings, we proved that LGL$_\mu$ can encode CL-CFGs, but we have no upper bound that may be used to show that LGL$_\mu$ *cannot* encode any C-CFGs. Hence, we leave the actual separation between LGL$_\mu$ and GL$_\mu$ as an open problem.

7 Related Work

We describe the related work on the complexity and expressivity of spatial logics. There is also a vast literature on the expressive power of first-order logic and monadic second-order logic. We refer to standard techniques arising from that work throughout the paper.

7.1 Previous Literature

Until recently, people working on spatial logics have restricted their attention to the combined complexity problem, since their motivation has been to develop either local Hoare reasoning about heap update or modal reasoning about process algebras. In contrast, our motivation arose from TQL project, and hence we study both complexity and expressivity.

Calcagno *et al.* studied the complexity of model-checking and validity for the separation logic, which is used to describe properties about mutable data structures stored in a heap [4]. They prove the undecidability of validity for the logic with quantifiers, and then focus on the quantifier-free fragment. Their most interesting result is decidability of validity for the quantifier-free fragment in the presence of the composition adjunct operator 'magic wand', which is surprising since for the adjoint case the satisfaction relation requires a universal quantification over heaps. The analogous decidability result is also show to hold for the quantifier-free static Ambient Logic [5], by adapting the technique used for the separation logic. More recently, Dal Zilio et al. have given a translation from the quantifier-free static Ambient Logic extended with Kleene star to a logic based on Presburger arithmetic, which provides a

different proof of the decidability result but more importantly yields a decision procedure [13] which is much more efficient.

These decidability results concern logics with adjuncts but no quantification. In contrast, we study versions of the spatial logic for graphs that include quantifiers but do not include the adjoint operator 'magic wand'. The combination of quantifiers and adjoints makes model-checking, and query-answering, undecidable. Quantification seems far more useful than 'magic wand' for querying trees, and this is the reason why TQL logic includes quantifiers but not the adjoint. When different uses of the logic are considered, the relative importance of adjoints and quantifiers is less easy to assess. Magic wand is used in an essential way to describe the weakest preconditions for O'Hearn and Reynold's Hoare logic for reasoning about heap update, although again it is not clear whether it is useful to specify properties of heaps. Lozes [14, 19] has proved that, in static Ambient Logic where quantification is only allowed on private names, every property expressible with 'magic wand' is also expressible without. In contrast, in [14], we prove that expressive power does increase when 'magic wand' is added to static Ambient Logic with quantification on *public* names. In both cases, the addition of 'magic wand' makes model-checking undecidable [10] in the presence of quantifiers.

Boneva, Talbot, and Tison study a logic without adjuncts and without quantification. They call it *STL*, and it is an extension of quantifier-free static Ambient Logic with μ-recursion [2]. Hence, *STL* is a quantifier-free version of the logic GL_μ that we study here. Absence of quantifiers means that they cannot use trees to encode graphs, as we did in Section 2.4, which makes *STL* much less expressive than GL_μ. They show that μ-recursion makes validity undecidable, contrasting with the results of [5, 13] which prove that validity in the logic with adjuncts and Kleene star, instead of recursion, is decidable, as described above. They also give complexity and expressivity result: they define an automata translation for this logic, and show how the logic can be syntactically restricted in order to match the expressive power of MSO and Presburger-MSO, which is an extension of MSO defined in [25]. Of course, they compare *STL* with MSO without quantification over names. Finally, they state that their automata approach can be used to prove that the data complexity of this quantifier-free logic is PTime, in sharp contrast with our PSPACE-completeness result for GL_μ.

Charatonik *et al.* studied the complexity of model-checking for the full ambient logic [9], which includes the reasoning about processes using spatial and temporal modalities. They prove that model-checking is PSPACE-complete for the logic with and without modalities, and with no adjunct. We do not consider modalities in this paper, since this work focusses on querying static graphs.

7.2 A Parallel Paper

A recent paper by Boneva and Talbot [1], independently presents similar results to those presented here and in [15]. The authors of [1] start from the same motivation as us—the characterization of the expressive power of TQL—which explains the similarity of the methods employed. Whilst we study the problem using GL, they target the full TQL logic and data model. They study three different fragments of TQL logic: TL, $TL^{\exists}$, and $TL^{\exists}_{\nu}$. The first is quantifier-free, hence is weaker than GL and non-comparable to LGL. The second includes both first-order and tree-quantification, hence is more expressive than MSO and GL. The third is similar to GL_μ, but stronger in that it includes both μ-recursion and tree-quantification. Despite these differences, the results about $TL^{\exists}$ are quite similar to our results about GL (model-checking is in PSPACE, and $TL^{\exists}$ can express complete problems at every level of the polynomial hierarchy), and their results about $TL^{\exists}_{\nu}$ are very similar to our results about GL_μ (model-checking is in PSPACE, and PSPACE-complete problems can be expressed). Their proof technique is based on a translation of quantified Boolean formulas, but the translation is different. Their proof of the PSPACE complexity of $TL^{\exists}_{\nu}$ model-checking is based on a version of Winskel's algorithm, as is ours. They do not study the linear version of $|$, nor the expressive power of their fragments of TQL

over strings. Finally, their conference paper only contains sketches of proofs.

8 Conclusions

We have investigated the complexity and expressive power of the spatial logic for graphs introduced by Cardelli, Gardner and Ghelli. The graph composition operator $|$ in this logic has a natural translation into second-order logic using an existential quantifier over sets of edges. This allows us to translate the logic GL to monadic second-order logic MSO, establishing upper bounds on the complexity of GL. We show that these bounds are optimal by showing, in particular, that complete problems at all levels of the polynomial hierarchy are expressible in GL. It seems unlikely that GL is as expressive as MSO in general. We are, however, able to show the surprising result that GL is as expressive as MSO when we restrict ourselves to strings, as it is able to define all regular languages. It remains to be seen whether this result can be extended to graphs that are trees. We have also shown that GL_μ, the logic GL extended with μ-recursion, has interesting expressive power. It allows us to define recursions of exponential length and to express PSPACE-complete problems. Nonetheless, the model-checking complexity of the full logic, with $|$ and recursion, remains within PSPACE.

Finally, we studied LGL with its restricted form of linear composition. We show that it relates to FO much in the same way as GL relates to MSO. There is a natural translation into FO, hence qualifying LGL as a fragment of FO, but the combined complexity of the two logics is the same, and their expressive power on strings is the same since both are able to express star-free regular languages. However, the recursion operator adds much more to LGL than LFP adds to FO: we can express PSPACE-complete problems, and a family of languages which are non-regular.

References

[1] I. Boneva and J.-M. Talbot. On complexity of model-checking for the TQL logic. In *3rd IFIP International Conference on Theoretical Computer Science*, August 2004.

[2] I. Boneva, J.-M. Talbot, and S. Tison. Expressiveness of a spatial logic for trees. In *LICS 2005*, pages 280–289, 2005.

[3] J. R. Büchi. Weak second order arithmetic and finite automata. *Zeitschrift f. Mathematische Logik und Grundlagen d. Mathematik*, 6:66–92, 1960.

[4] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS: 21st Conference*, volume 2245 of *Springer LNCS*, pages 108–119, 2001.

[5] Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. *ACM SIGPLAN Notices*, 38(3):62–73, 2003.

[6] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *ICALP: Automata, Languages, and Programming, 29th International Colloquium*, volume 2380 of *Springer LNCS*, pages 597–610, 2002.

[7] L. Cardelli and G. Ghelli. TQL: A query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 2003. To appear.

[8] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *POPL*, pages 365–377, 2000.

[9] W. Charatonik, S. Dal Zilio, A. D. Gordon, S. Mukhopadhyay, and J-M. Talbot. Model checking mobile ambients. *Theoretical Computer Science*, 308:277–331, 2003.

[10] G. Conforti and G. Ghelli. Decidability of freshness, undecidability of revelation (extended abstract). In *FoSSaCS: Foundations of Software Science and Computation Structures, 7th Conference*, pages 105–120, March-April 2004.

[11] G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The query language TQL. In *International Workshop on the Web and Databases (WebDB), Madison, Wisconsin, USA*, pages 19–24, 2002.

[12] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Graph Transformations*, chapter 5, pages 313–400. World Scientific, 1997.

[13] S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *POPL: 31st ACM Symposium on Principles of Programming Languages*, pages 135–146, 2004.

[14] A. Dawar, P. Gardner, and G. Ghelli. Games for the ambient logic. forthcoming.

[15] A. Dawar, P. Gardner, and G. Ghelli. Expressiveness and complexity of graph logic. Technical Report 2004/3, Imperial College, London, April 2004.

[16] H-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 2 edition, 1999.

[17] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. M. Karp, editor, *Complexity of Computation, SIAM-AMS Proceedings, Vol 7*, pages 43–73, 1974.

[18] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[19] E. Lozes. Adjuncts elimination in the static ambient logic. In *Proc. of Express'03, Marseille*, 2003.

[20] R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, 1971.

[21] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19. Springer-Verlag, 2001.

[22] A. Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[23] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[24] Kai Salomaa and Sheng Yu. Alternating finite automata and star-free languages. *Theoretical Computer Science*, 234(1-2):167–176, 2000.

[25] H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *22nd Symp. on Principles of Database Systems*, pages 155–166, 2003.

[26] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[27] G. Winskel. A note on model checking the modal nu-calculus. *Theoretical Computer Science*, 83(1):157–167, 1991.