Canonical Graph Shapes

Arend Rensink*

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, The Netherlands
rensink@cs.utwente.nl

Abstract. Graphs are an intuitive model for states of a (software) system that include pointer structures — for instance, object-oriented programs. However, a naive encoding results in large individual states and large, or even unbounded, state spaces. As usual, some form of abstraction is necessary in order to arrive at a tractable model.
In this paper we propose a decidable fragment of first-order graph logic that we call *local shape logic* (LSL) as a possible abstraction mechanism, inspired by previous work of Sagiv, Reps and Wilhelm. An LSL formula constrains the multiplicities of nodes and edges in state graphs; abstraction is achieved by reasoning not about individual, concrete state graphs but about their characteristic shape properties. We go on to define the concept of the *canonical shape* of a state graph, which is expressed in a *monomorphic* sub-fragment of LSL, for which we define a graphical representation. We show that the canonical shapes give rise to an automatic finite abstraction of the state space of a software system, and we give an upper bound to the size of this abstract state space.

1 Introduction

This paper is part of an investigation into the use of graphs as models of system states, for the (eventual) purpose of system verification, especially of software. In this approach, an individual system state (state snapshot) is modeled as an *edge-labeled graph*, in which the nodes roughly stand for the entities (records, objects) present in the state and the edges for properties or fields (attributes, variables) of those resources. The dynamic behavior of a system is modeled as a *transition system* in which the states are graphs in the above sense.

As usual in the context of verification, the main problem is *state space explosion*; i.e., the effect that, even for small systems, the number of states to be analyzed exceeds all reasonable bounds. The most promising solution technique to cope with this is *abstraction*, meaning that information is discarded from the model, after which it can be represented more compactly — usually at the cost of either soundness or completeness of the verification. In the context of graphs, we have previously studied abstraction techniques in [6,7]. The idea there is to have one or more nodes whose cardinality is not *a priori* fixed but may grow

* The work in this paper took place in the GROOVE project (Graphs for Object-Oriented Verification), funded under the Dutch NWO grant 612.000.314

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 401–415, 2004.

unboundedly in the course of system execution. This idea can be found also in *shape graphs*, as defined by Sagiv, Reps and Wilhelm in [18]. There, too, some graph nodes (called *summary nodes*) stand for multiple instances; furthermore, shape graphs contain additional information about whether an edge is necessarily there for every instance of a given node and whether outgoing edges may be pointing to (i.e., sharing) the same node instance.

The current paper is a consequence of our earlier efforts, inspired by the work on shape graphs. We define a theory of graph shapes by formulating additional information about the node and edge multiplicity as a constraint in what we call *local shape logic* (LSL). LSL is essentially a fragment of typed first-order logic, where the typing is controlled by a *type graph*. The definition of LSL is parameterized by a *multiplicity algebra*, which partially controls the expressiveness of the logic. We show LSL to be decidable.

The combination of a type graph and a shape constraint gives rise to a (generalized) shape graph. Thus, each shape graph defines a set of state graphs, viz. those instances of the type graph that satisfy the shape constraint. Unfortunately, LSL formulae in general lack the appealing pictorial representation of graphs. To alleviate this, we define a graphical representation of a so-called *monomorphic* fragment of LSL, and we show that any shape graph is equivalent to a set of monomorphic shape graphs. Finally, we define a restricted class of *canonical* monomorphic shapes, with the property that there is an automatic abstraction from state graphs to canonical shapes. We show that the set of distinct canonical shape graphs is finite, and we give an upper bound for its size.

The paper is structured as follows: Sect. 2 contains basic definitions, in Sect. 3 we introduce local shape logic and show its decidability; and in Sect. 4 we discuss (monomorphic) shape graphs. Sect. 5 concludes and discusses related work. A more extensive exposition of the material presented here, including proofs of the main theorems, can be found in the full report version: see [16].

2 Graphs and Type Graphs

We represent states as graphs. Nodes can be thought of as *locations* or *objects*, and edges are used to represent *variables*, in particular *references*. For the formal definition, we assume the existence of a global set $\mathbf{N}$ of *nodes*, ranged over by u, v, w; subsets of $\mathbf{N}$ are denoted N, V, W. We use the symbol $\bot$ ($\notin \mathbf{N}$) to denote an *undefined* node; for an arbitrary set $N \subseteq \mathbf{N}$ we write $N_\bot$ to denote $N \cup \{\bot\}$. We also assume a global set $\mathbf{L}$ of *labels*, ranged over by a, b, c.

Definition 1 (graph). *A* graph G *is a tuple* (N, E)*, where* $N \subseteq \mathbf{N}$ *is a finite set of* nodes*, and* $E \subseteq N \times \mathbf{L} \times N_\bot$ *is a finite set of* edges*.*

It follows that a graph consists of binary edges (v, a, w) with $w \in \mathbf{N}$ but also *unary* edges $(v, a, \bot)$. We also refer to unary edges as *node properties* or *node predicates*. For most purposes unary and binary edges will be treated uniformly. We use $\mathbf{G}$ to denote the set of all graphs, ranged over by G, H. Given a graph G, we denote the node and edge sets of G by N_G, E_G, respectively. We drop the subscripts $_G$ when clear from the context.

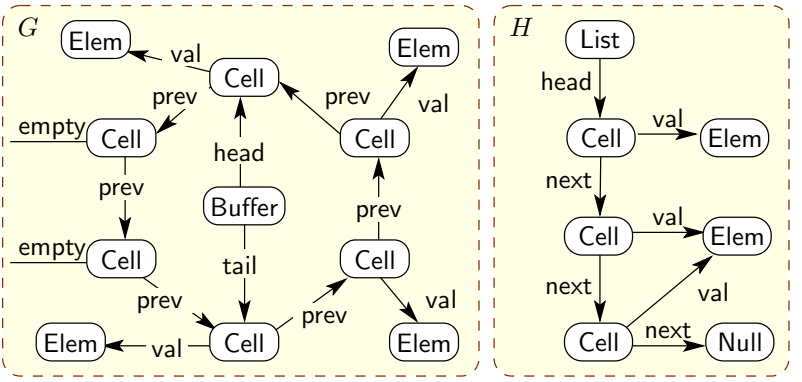


Fig. 1. State snapshots of a circular buffer and a linked list

As usual, we draw graphs by showing nodes as boxes and binary edges as arrows between them. Unary edges are represented by lines without arrow heads or target nodes, or by writing their labels inside the source nodes.

Example 1. Fig. 1 shows two state graphs. The graph G on the left hand side models a circular buffer, as a backwards-linked list of cells of which some have values (modeled by val-labeled edges to the nodes representing the respective values) and the others are empty (modeled by empty-labeled unary edges). One of the cells is designated head to indicate that this is the head of the buffer, whose value is to be retrieved next; in contrast, the tail cell contains the newest value. Buffer, Cell and Elem are node predicates used to reflect the types of the nodes. The right hand side graph H depicts a linked list, using a similar encoding.

The effects of execution steps are modeled by modifications to the state graphs, resulting in new graphs which differ (locally) from the original ones. This gives rise to a transition system in which the states are graphs and the transitions graph transformations. For instance, the primary operations upon a circular buffer are the insertion and retrieval of values: these result in changes in the neighborhood of the tail- and head-edges, which remove empty-predicates and add val-edges and Elem-nodes. See [16] for further details.

The particular choice of node identities in a given graph is regarded as incidental and not as part of the structure: they only serve to distinguish the nodes from one another. In figures we usually omit the node identities altogether; the nodes are then already distinguished by their position. The same principle can be found also in the notion of graph *morphism*: these preserve only the structurally important information.

Definition 2 (graph morphism). *Given two graphs G, H, a* graph morphism *f from G to H is a function $f\colon N_G \to N_H$, strictly extended to $\bot$, such that $(f(p), a, f(q)) \in E_H$ for all $(p, a, q) \in E_G$.*

$f\colon G \to H$ denotes that f is a morphism from G to H. A morphism f is called *injective* [*surjective, bijective*] if f is an injective [surjective, bijective] function both on nodes and edges.

In the following we also need the following (standard) notions of graph *partitioning*: for a given graph $G = (N, E)$, every node equivalence relation

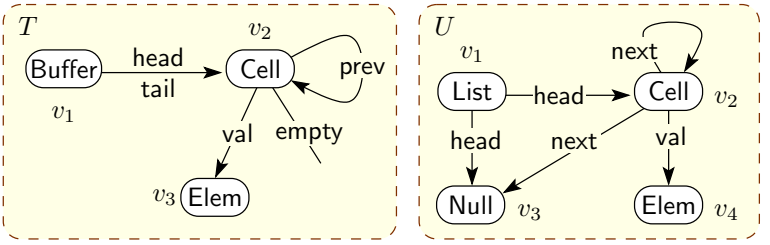


Fig. 2. Type graphs T for circular buffers and U for linked lists

$\sim \subseteq N \times N$ gives rise to a partitioned graph $G/\!\sim = (N/\!\sim, E/\!\sim)$ where

$$N/\!\sim = \{[v]_\sim \mid v \in N\}$$
$$E/\!\sim = \{[v]_\sim, a, [w]_\sim \mid (v, a, w) \in E\}$$

(in which, as usual, $[v]_\sim = \{w \in N \mid v \sim w\}$ for all $v \in N$; so $[\bot]_\sim = \bot$). Furthermore, we use $\pi_\sim : G \to G/\!\sim$ to denote the (surjective) morphism defined by $\pi_\sim(v) = [v]_\sim$ for all $v \in N$. Also, for an arbitrary partitioning Π of the set of nodes N and an arbitrary node $v \in N_\bot$, we use $[v]_\Pi$ to denote the unique $V \in \Pi$ such that $v \in V$; so $[\bot]_\Pi = \bot$.

The graphs used to model the states of a given software system are, of course, not arbitrary. For one thing, not all edges or combinations of edges are allowed. To take the circular buffer G of Fig. 1, there are only eight labels; Buffer, Cell and Elem only occur as node predicates of mutually exclusive nodes; prev-edges only occur between Cell-nodes; et cetera. Such information can be captured partially using *graph typings*.

Definition 3 (typing). *Let $G \in \mathbf{G}$ be arbitrary. A* typing *of G is a morphism $\tau : G \to T$, where $T \in \mathbf{G}$ is called a* type graph.

We call T a *type of* G and G an *instance of* T. We use $\mathbf{G}^T$ to denote the set of instances of a graph T. For instance, Fig. 2 shows types for the graphs in Fig. 1.

3 Shape Logic

The notion of graph typing is rather weak: the existence of a morphism from a would-be instance graph to a would-be type graph can only forbid but never enforce the presence of certain edges in the instance. To take the type graph U in Fig. 2, the *intention* is that any instance obeys the following structural properties (among others). Although H in Fig. 1 indeed satisfies these intended properties, U has many instances that do not.

1. Every List-labeled node has precisely one outgoing head-edge, to the first element of the corresponding list or to a Null-node;
2. There is precisely one Null-node;
3. Every Cell-node has precisely one outgoing next-edge, either to another Cell-node or to a Null-node;
4. Every Cell-node has either one incoming next-edge or one incoming head-edge (so Cell-nodes are not shared).

To strengthen the notion of typing we need to formulate additional constraints on instances. For this purpose we define a *shape logic*, SL^T, which is a first-order graph logic, defined relative to a type graph T. Later on (Sect. 3.2) we restrict this to a fragment of which we show decidability. We assume a countable set of variables $\mathbf{V}$. Formulae of SL^T are generated by the following grammar:

$$\phi \;::=\; \mathbf{tt} \mid x = x \mid x \xrightarrow{a} \mid x \xrightarrow{a} x \mid \neg\phi \mid \phi \vee \phi \mid \forall x{:}\, v.\, \phi \;.$$

In this grammar, $a \in \mathbf{L}$ denotes an arbitrary label, $x \in \mathbf{V}$ an arbitrary (node) variable and $v \in N_T$ a node of the type graph T. We employ the usual abbreviations $\phi \Rightarrow \psi$, $\phi \wedge \psi$, $\exists x{:}\, v.\, \phi$, $\exists! x{:}\, v.\, \phi$ and $\mathbf{tt}$.

The predicates $x \xrightarrow{a}$ and $x \xrightarrow{a} y$ express that there exists a (unary resp. binary) a-labeled edge in a would-be instance of T, from the node denoted by x and leading (in the case of a binary edge) to the node denoted by y. It is sometimes convenient to write $x \xrightarrow{a} \bot$ rather than $x \xrightarrow{a}$.

We consider only formulae that are *well-typed* according to T, in the sense that each free variable x in a formula has an associated type node $v_x \in N_T$ such that $(v_x, a, \bot) \in E_T$ for all propositions $x \xrightarrow{a}$ and $(v_x, a, v_y) \in E_T$ for all propositions $x \xrightarrow{a} y$. We do not work this out formally here.

Example 2. Using SL we can give a much tighter characterization of lists than through type graphs. Given U in Fig. 2, consider the following SL^U-constraints:

$$\forall x{:}\, v_1.\; x \xrightarrow{\mathsf{List}} \wedge\, ((\exists! y{:}\, v_2.\; x \xrightarrow{\mathsf{head}} y \wedge \nexists y{:}\, v_3.\; x \xrightarrow{\mathsf{head}} y) \vee (\exists! y{:}\, v_3.\; x \xrightarrow{\mathsf{head}} y \wedge \nexists y{:}\, v_2.\; x \xrightarrow{\mathsf{head}} y)) \tag{1}$$

$$(\exists! y{:}\, v_3.\; \mathbf{tt}) \wedge (\forall x{:}\, v_3.\; x \xrightarrow{\mathsf{Null}}) \tag{2}$$

$$\forall x{:}\, v_2.\; (\exists! y{:}\, v_2.\; x \xrightarrow{\mathsf{next}} y \wedge \nexists y{:}\, v_3.\; x \xrightarrow{\mathsf{next}} y) \vee (\exists! y{:}\, v_3.\; x \xrightarrow{\mathsf{next}} y \wedge \nexists y{:}\, v_2.\; x \xrightarrow{\mathsf{next}} y) \tag{3}$$

$$\forall x{:}\, v_2.\; (\exists y{:}\, v_1.\; y \xrightarrow{\mathsf{head}} x \wedge \nexists y{:}\, v_2.\; y \xrightarrow{\mathsf{next}} x) \vee (\nexists y{:}\, v_1.\; y \xrightarrow{\mathsf{head}} x \wedge \exists! y{:}\, v_2.\; y \xrightarrow{\mathsf{next}} x) \tag{4}$$

$$\exists x{:}\, v_4.\; \nexists y{:}\, v_2.\; y \xrightarrow{\mathsf{val}} x \tag{5}$$

Constraints (1)–(4) precisely capture the list properties enumerated above. On the other hand, (5) expresses that there exists an Elem-node *not* reachable from a Cell-node; this is *not* satisfied by the U-typing of H.

Note that we can *not* express in SL that every v_2-instance should be connected by a sequence of next-edges to a v_3-instance. This is a consequence of the inability of first-order graph logic to express connectedness (see, e.g., Courcelle [5]).

Example 3. Assume an additional edge (v_4, leq, v_4) in U of Fig. 2, modeling a pre-order $<$ over Elem-nodes; that is, $(v, \mathsf{leq}, w) \in E$ implies $v < w$. The following constraints express that $<$ is transitive and Cell-values are in ascending order:

$$\forall x{:}\, v_4.\; \forall y{:}\, v_4.\; \forall z{:}\, v_4.\; x \xrightarrow{\mathsf{leq}} y \wedge y \xrightarrow{\mathsf{leq}} z \Rightarrow x \xrightarrow{\mathsf{leq}} z \tag{6}$$

$$\forall c_1{:}\, v_2.\; \forall c_2{:}\, v_2.\; \forall x_1{:}\, v_4.\; \forall x_1{:}\, v_4.\; c_1 \xrightarrow{\mathsf{next}} c_2 \wedge c_1 \xrightarrow{\mathsf{val}} x_1 \wedge c_2 \xrightarrow{\mathsf{val}} x_2 \Rightarrow x_1 \xrightarrow{\mathsf{leq}} x_2 \tag{7}$$

The meaning of a shape constraint is defined as the set of typings that satisfy it. Formally, satisfaction of a formula $\phi \in \mathsf{SL}^T$ is defined by a ternary predicate $\tau, \theta \vDash \phi$, where $\tau{:}\, G \to T$ is a typing and $\theta{:}\, fv(\phi) \to N_G$ a valuation of the free

variables of ϕ (extended strictly to $\bot$) such that $\tau(\theta(x)) = v_x$ for all $x \in fv(\phi)$. The following rules, plus the obvious ones for $\mathbf{tt}$, $\vee$ and $\neg$, define $\vDash$ ($\theta\{v/x\}$ denotes the valuation that maps x to v and equals θ elsewhere):

$$\begin{array}{l}\tau, \theta \vDash x = y \quad \text{if } \theta(x) = \theta(y)\\ \tau, \theta \vDash x \xrightarrow{a} y \quad \text{if } (\theta(x), a, \theta(y)) \in E_G\\ \tau, \theta \vDash \forall x{:}\, v.\ \phi \text{ if } \tau, \theta\{w/x\} \vDash \phi \text{ for all } w \in \tau^{-1}(v).\end{array}$$

Unfortunately, although shape logic clearly strengthens the notion of typing as intended, it does too much of that. Being a (non-monadic) first order logic, SL is not decidable, which is necessary if it is to be used for verification. Instead, we define a reduced fragment which only allows to express *local* shape properties: *multiplicities* of single node instances, and of their incoming and outgoing edges.

3.1 Multiplicities

A multiplicity is an abstract indication the cardinality of a given set; for instance, the set of instances of a given (type) node or the set of edges leaving a node.

Definition 4. *A* multiplicity algebra *is a tuple* $\langle \mathbf{M}, \_ : \_, \bigsqcap, \mathbf{0}, \mathbf{1} \rangle$ *where*

- $\mathbf{M}$ *is a set of* multiplicities*;*
- $\_ : \_ \subseteq \mathbb{N} \times \mathbf{M}$ *is* membership*. For* $\mu \in \mathbf{M}$ *we denote* $\mathbb{N}^\mu = \{m \in \mathbb{N} \mid m : \mu\}$*;*
- $\bigsqcap : \mathbf{2}^{\mathbf{M}} \to \mathbf{M}$ *is* intersection*, such that* $\mathbb{N}^{\bigsqcap M} = \bigcap_{\mu \in M} \mathbb{N}^\mu$*.*
- $\mathbf{0} \in \mathbf{M}$ *is the* zero multiplicity*, such that* $\mathbb{N}^{\mathbf{0}} = \{0\}$*;*
- $\mathbf{1} \in \mathbf{M}$ *is the* singular multiplicity*, such that* $\mathbb{N}^{\mathbf{1}} = \{1\}$*.*

It can be deduced that $\mathbf{M}$ also contains the *inconsistent multiplicity* $\bot = \bigsqcap \mathbf{M}$ and a *universal multiplicity* $\top = \bigsqcap \emptyset$ (thus $\mathbb{N}^\bot = \emptyset$ and $\mathbb{N}^\top = \mathbb{N}$). Some derived concepts (where $\mu, \mu_1, \mu_2 \in \mathbf{M}$ are arbitrary and V is an arbitrary set):

- Lower bounds $\lfloor \mu \rfloor = \min \mathbb{N}^\mu$ (where $\min \emptyset = \omega$);
- Upper bounds $\lceil \mu \rceil = \max \mathbb{N}^\mu$ (where $\max \mathbb{N} = \omega$ and $\max \emptyset = 0$);
- Multiplicity addition $\mu_1 \oplus \mu_2 = \bigsqcap \{\mu \in \mathbf{M} \mid m_1{:}\mu_1 \wedge m_2{:}\mu_2 \Rightarrow m_1{+}m_2 : \mu\}$;
- A partial multiplicity ordering $\mu_1 \sqsubseteq \mu_2 \Leftrightarrow \mu_1 \sqcap \mu_2 = \mu_1$;
- Set multiplicity $\#_{\mathbf{M}} V = \bigsqcap \{\mu \in \mathbf{M} \mid \#V : \mu\}$ (where $\#V$ is V's cardinality).

It follows that $\mathbb{N}^{\mu_1 \oplus \mu_2} \supseteq \mathbb{N}^{\mu_1} + \mathbb{N}^{\mu_2}$ (where addition is extended pointwise to sets) and $\mu_1 \sqsubseteq \mu_2$ if and only if $\mathbb{N}^{\mu_1} \subseteq \mathbb{N}^{\mu_2}$.

- We call $\mathbf{M}$ *interval based* if $\mathbb{N}^\mu = \{i \in \mathbb{N} \mid \lfloor \mu \rfloor \leq i \leq \lceil \mu \rceil\}$ for all $\mu \in \mathbf{M}$.
- We say that $\mathbf{M}$ has *collective complements* if for all $\mu \in \mathbf{M}$, there is a set $\bar{\mu} \subseteq \mathbf{M}$ such that $\mu \sqcap \nu = \bot$ for all $\nu \in \bar{\mu}$, and $\mathbb{N}^\mu \cup \bigcup_{\nu \in \bar{\mu}} \mathbb{N}^\nu = \mathbb{N}$.

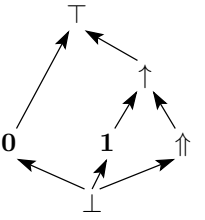
Henceforth, we only consider interval-based multiplicity algebras. In examples we will use, apart from the multiplicities enforced by the definition, $\uparrow$ for *at least one* and $\Uparrow$ for *more than one* (hence $\mathbb{N}^\uparrow = \{1, \ldots\}$ and $\mathbb{N}^\Uparrow = \{2, \ldots\}$). Note that $\{\bot, \mathbf{0}, \mathbf{1}, \uparrow, \Uparrow, \top\}$ has collective complements; in particular, $\bar{\mathbf{0}} = \uparrow$ and $\bar{\mathbf{1}} = \{\mathbf{0}, \Uparrow\}$. Membership, addition and intersection for this set are shown in Table 1.

Example 4. The multiplicities in the UML are ranges $i..j$ where $i \in \mathbb{N}$ and $j \in \mathbb{N} \cup \{*\}$ with $i \leq j$ — where $*$ denotes "arbitrarily many" and satisfies $k < *$ and $*{+}k = k{+}* = *$ for all $k \in \mathbb{N}$. After adding $\bot$, this gives rise to the *largest* interval based multiplicity algebra.

Table 1. Multiplicity membership, addition and ordering

$m : \_$	condition
$\top$	$\mathbf{tt}$
$\Uparrow$	$m > 1$
$\uparrow$	$m > 0$
$\mathbf{1}$	$m = 1$
$\mathbf{0}$	$m = 0$
$\bot$	$\mathbf{ff}$

$\oplus$	$\top$	$\Uparrow$	$\uparrow$	$\mathbf{1}$	$\mathbf{0}$	$\bot$
$\top$	$\top$	$\Uparrow$	$\uparrow$	$\uparrow$	$\top$	$\bot$
$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\bot$
$\uparrow$	$\uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\uparrow$	$\bot$
$\mathbf{1}$	$\uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\mathbf{1}$	$\bot$
$\mathbf{0}$	$\top$	$\Uparrow$	$\uparrow$	$\mathbf{1}$	$\mathbf{0}$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$



3.2 Local Shape Logic

On the basis of a multiplicity algebra we now define *local shape logic*, LSL^T, as the set of formulae ϕ generated by the following grammar:

$$\xi ::= v \mid \xrightarrow{a} v \mid \xleftarrow{a} v \mid \underline{\overset{a}{}} \ .$$
$$\phi ::= \mathbf{tt} \mid {}^{\mu}[\xi] \mid \neg\phi \mid \phi \vee \phi \mid \forall_v \phi \ .$$

ξ ranges over (node or edge set) *expressions*: v stands for the set of instances of the type node v ($\in N_T$); $\xrightarrow{a} v$ stands for the set of a-labeled edges from the current node to some instance of v; $\xleftarrow{a} v$ is the dual, expressing the set of a-labeled edges from some instance of v to the current node; and $\underline{\overset{a}{}}$ stands for the (empty or singleton) set of a-labeled unary edges from the current node.

The formula ${}^{\mu}[\xi]$ expresses that the set denoted by ξ has multiplicity μ ($\in \mathbf{M}$), and $\forall_v \phi$ expresses that ϕ holds for all *instances* of v. We write $\exists_v \phi$ for the dual of $\forall_v \phi$. The following example gives an impression of the scope of LSL^T.

Example 5. The constraints in Ex. 2 have equivalent counterparts in LSL^T:

(1′) $\forall_{v_1}({}^{\uparrow}[\underline{\overset{\mathsf{List}}{}}] \wedge (({}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_3]) \vee ({}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_3])))$
(2′) ${}^{\mathbf{1}}[v_3] \wedge \forall_{v_3} {}^{\uparrow}[\underline{\overset{\mathsf{Null}}{}}]$
(3′) $\forall_{v_2}(({}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} v_2] \wedge {}^{\mathbf{0}}[\xrightarrow{\mathsf{next}} v_3]) \vee ({}^{\mathbf{0}}[\xrightarrow{\mathsf{next}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} v_3]))$
(4′) $\forall_{v_2}(({}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{0}}[\xleftarrow{\mathsf{next}} v_2]) \vee ({}^{\mathbf{0}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{1}}[\xleftarrow{\mathsf{next}} v_2]))$
(5′) $\exists_{v_4} {}^{\mathbf{0}}[\xleftarrow{\mathsf{val}} v_2]$

On the other hand, the constraints in Ex. 3 do *not* have an equivalent counterpart in LSL^T. Informally, the reason is that these constraints express structural properties involving more than two nodes at a time.

Clearly, LSL^T formulae can be regarded as sugared SL^T-formulae. We give a direct semantics for LSL^T, shown in [16] to coincide with the SL^T-semantics of the de-sugared formulae. The meaning of expressions ξ is given by the following function, where $\tau\colon G \to T$ is a typing and $u \in N_G$:

$$[\![\xrightarrow{a} v]\!]_{\tau,u} = \{(u,a,w) \in E_G \mid v = \tau(w)\}$$
$$[\![\xleftarrow{a} v]\!]_{\tau,u} = \{(w,a,u) \in E_G \mid v = \tau(w)\}$$
$$[\![\underline{\overset{a}{}}]\!]_{\tau,u} = \{(u,a,\bot) \in E_G\}$$
$$[\![v]\!]_{\tau,u} = \{w \in N_G \mid v = \tau(w)\} \ .$$

We now define a satisfaction relation $\tau, u \vDash_{\mathsf{LSL}} \phi$ for $\phi \in \mathsf{LSL}^T$. The rules for negation and disjunction are as always; we just show the special constructors.

$$\begin{array}{l} \tau, u \vDash_{\mathsf{LSL}} {}^{\mu}[\xi] \;\text{ if } \#[\![\xi]\!]_{\tau,u} : \mu \\ \tau, u \vDash_{\mathsf{LSL}} \forall_v \phi \;\text{ if } \tau, w \vDash \phi \text{ for all } w \in \tau^{-1}(v) \;. \end{array}$$

Henceforth, we will drop the suffix LSL. Some notational shorthand for LSL:

- ${}^{\mu}[\Xi]$ for a finite set of expressions Ξ will express that the *sum* of the multiplicities of the expressions in Ξ equals μ, i.e., $\mu = \bigoplus_{\xi \in \Xi} \mu_\xi$ where for all $\xi \in \Xi$, μ_ξ is the smallest multiplicity w.r.t. $\sqsubseteq$ such that ${}^{\mu_\xi}[\xi]$. More precisely:

$$ {}^{\mu}[\{\xi_i\}_i] \;\equiv\; \bigvee_{\mu = \bigoplus_i \mu_i} \bigwedge_i {}^{\mu_i}[\xi_i] \;. $$

- ${}^{\mu}[\xrightarrow{a} V]$ for a finite set of nodes V is equivalent to ${}^{\mu}[\{\xrightarrow{a} v \mid v \in V\}]$ and ${}^{\mu}[\xleftarrow{a} V]$ is equivalent to ${}^{\mu}[\{\xleftarrow{a} v \mid v \in V\}]$.

For instance, ${}^{\uparrow}[\{\xi_i\}_i]$ is shorthand for $\bigvee_i ({}^{\uparrow}[\xi_i] \wedge \bigwedge_{j \neq i} {}^{\top}[\xi_j])$ and ${}^{\mathbf{1}}[\{\xi_i\}_i]$ for $\bigvee_i ({}^{\mathbf{1}}[\xi_i] \wedge \bigwedge_{j \neq i} {}^{\mathbf{0}}[\xi_j])$. Concretely, we may abbreviate constraint (4′) of Ex. 5 to $\forall_{v_2} {}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1 \mid \xleftarrow{\mathsf{next}} v_2]$ and (3′) to $\forall_{v_2} {}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} \{v_2, v_3\}]$. As another example, in the circular buffer type graph T in Fig. 2 we should have $\forall_{v_2} {}^{\mathbf{1}}[\xrightarrow{\mathsf{empty}} \mid \xrightarrow{\mathsf{val}} v_3]$.

Satisfiability, implication and equivalence of local shape logic are decidable. We prove this by defining a translation from LSL to sets of *integer programs* (IPs),[1] such that a given formula is satisfiable if and only if (at least) one integer program in the corresponding set has a solution.

Theorem 1. *Let $T \in \mathbf{G}$ be arbitrary and let $\mathbf{M}$ be an arbitrary multiplicity algebra with collective complements. For every formula $\phi \in \mathsf{LSL}^T$ there is a set $\mathbf{S}$ of IPs, such that ϕ is satisfiable if and only if some IP in $\mathbf{S}$ admits a solution.*

The proof (which can be found in [16]) proceeds by first defining a *disjunctive normal form* for LSL formulae; for each of the conjunctive sub-terms of the normal form there is a (more or less) direct translation to a characteristic IP. The inverse of the theorem also holds: for any IP there is an LSL-formula whose instances essentially correspond to solutions of the IP. See [16] for details. From the decidability of integer programming (see, e.g., [15]) follows:

Corollary 1 (decidability). *Satisfiability of LSL^T is decidable for any $T \in \mathbf{G}$.*

4 Shape Graphs

We now combine type graphs with shape constraints; the resulting models will take over the role of type graphs in capturing intended graph structure.

Definition 5. *A* shape graph *is a tuple $\Gamma = (T, \phi)$ where T is a type graph and $\phi \in \mathsf{LSL}^T$ a closed shape constraint. A Γ-*shaping *(or just* shaping*) is a typing $\tau\colon G \to T$ such that $\tau \vDash \phi$. We will call Γ a* shape of G *and G an* instance of Γ.

[1] An integer program is a set of linear equations for which an integer solution is sought. Such a program can be represented by a matrix $A \in \mathbb{Z}^{m \times n}$ together with a vector $b \in \mathbb{Z}^n$; a solution is then a vector $x \in \mathbb{Z}^m$ for which $A \cdot x = b$.

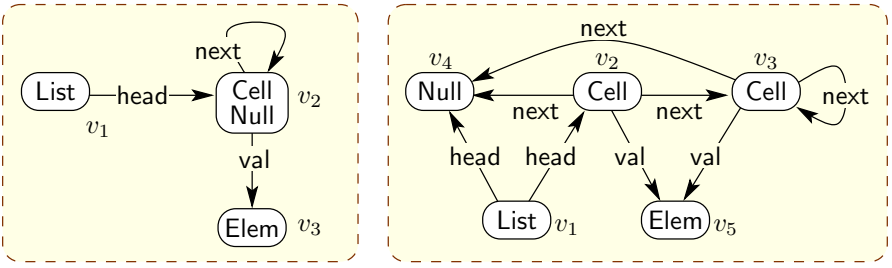


Fig. 3. Alternative list shapes (see Ex. 6)

We use N_Γ etc. to denote the components of Γ, and $\tau\colon G \to \Gamma$ to denote that τ is a shaping of G in Γ. Every state graph has many different shapes. We consider one shape to be more *abstract* than another if it allows more instances.

Example 6. An alternative shaping for lists is induced by the left hand graph in Fig. 3, with a shape constraint containing the following sub-formula for v_2:

$$\forall_{v_2} {}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} v_2] \wedge (({}^{\mathbf{1}}[\xrightarrow{\mathsf{Cell}}] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{val}} v_3]) \vee {}^{\mathbf{1}}[\xrightarrow{\mathsf{Null}}]) \wedge {}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1 \mid \xleftarrow{\mathsf{next}} v_2] \ .$$

(The intended shape constraints for the other nodes are omitted here.) This is strictly more abstract than U in Fig. 2 with the constraints in Ex. 5, because it allows instances with arbitrarily many Null-nodes, whereas (2′) specifies that there should be precisely one such. In contrast, the right hand side of Fig. 3 shows a more *concrete* list shape: here the head Cell-node is explicitly distinguished.

To formalize this we have to relate shape constraints over different type graphs. The following converts a morphism $f\colon T \to U$ to a reverse mapping $f^{-1}\colon \mathsf{LSL}^U \to \mathsf{LSL}^T$. $\xi\langle N\rangle$ denotes the set obtained by replacing v in ξ by of the nodes in N.

$$\begin{aligned} f^{-1}(\mu[\xi]) &= \mu[\xi\langle f^{-1}(v)\rangle] \\ f^{-1}(\neg\phi) &= \neg f^{-1}(\phi) \\ f^{-1}(\phi \vee \psi) &= f^{-1}(\phi) \vee f^{-1}(\psi) \\ f^{-1}(\forall_v \phi) &= \textstyle\bigwedge_{v=f(w)} \forall_w f^{-1}(\phi) \ . \end{aligned}$$

Definition 6. *Given two shape graphs Γ, Δ, an* abstraction *from Γ to Δ is a morphism $\alpha\colon T_\Gamma \to T_\Delta$ such that $\phi_\Gamma \Rightarrow \alpha^{-1}(\phi_\Delta)$.*

Example 7. Taking into account the shape constraints discussed in Examples 5 and 6, the following mapping α_1 defines an abstraction from the right hand side of Fig. 2 to the left hand side of Fig. 3, whereas α_2 defines an abstraction from the right hand side of Fig. 3 to the right hand side of Fig. 2.

$$\begin{aligned} \alpha_1 &= \{(v_1, v_1), (v_2, v_2), (v_3, v_2), (v_4, v_3)\} \\ \alpha_2 &= \{(v_1, v_1), (v_2, v_2), (v_3, v_2), (v_4, v_3), (v_5, v_4)\} \ . \end{aligned}$$

We write $\alpha\colon \Gamma \to \Delta$ to denote that α is an abstraction from Γ to Δ. Note that abstractions can be composed. The following is immediate.

Proposition 1. *If $\tau\colon G \to \Gamma$ is a shaping and $\alpha\colon \Gamma \to \Delta$ an abstraction, then $\alpha \circ \tau\colon G \to \Delta$ is a shaping.*

A shape graph can be seen as a *specification* of the set of state graphs of which it is a shape. Alternatively we may consider *sets* of shape graphs. We call two sets of shape graphs, $\mathbf{\Gamma}$ and $\mathbf{\Delta}$, *equivalent* if they specify the same sets of instances:

$$\mathbf{\Gamma} \equiv \mathbf{\Delta} \quad :\Leftrightarrow \quad \forall G \in \mathbf{G} : (\exists \Gamma \in \mathbf{\Gamma}, \tau : \tau : G \to \Gamma) \iff (\exists \Delta \in \mathbf{\Delta}, \tau : \tau : G \to \Delta) \ .$$

For instance, if $\phi_\Gamma = \bigvee_c \phi_c$ then $\{\Gamma\} \equiv \{T_\Gamma, \phi_c\}_c$.

4.1 Monomorphic Shapes

Due to the combination of conjunction and disjunction in shape constraints, shape graphs are not easy to visualize, or to reason about on an intuitive level. This may not be important from a formal point of view but makes shape graphs less useful when it comes to conveying ideas and principles. To alleviate this, we define a *monomorphic* fragment of local shape logic for which a concise visual representation can be given, and we show that every shape graph is equivalent to a set of monomorphic shape graphs.

Definition 7. *A shape graph Γ is* monomorphic *if there is a partitioning Π of N_Γ and consistent multiplicities μ^v for $v \in N_\Gamma$ and $\mu^{v,a,[w]_\Pi}, \mu^{[v]_\Pi,a,w}$ for $(v,a,w) \in E_\Gamma$, such that ϕ_Γ is equivalent to*

$$\textstyle\bigwedge_{v \in N} {}^{\mu^v}[v] \wedge \bigwedge_{(v,a,w)\in E} \big(\forall_v {}^{\mu^{v,a,[w]_\Pi}} [\xrightarrow{a}[w]_\Pi] \wedge \forall_w {}^{\mu^{[v]_\Pi,a,w}} [\xleftarrow{a}[v]_\Pi] \big) \ .$$

Thus, the multiplicity of each edge $e = (v,a,w)$ in a monomorphic shape graph is constrained by precisely one (combined) incoming edge predicate $\xleftarrow{a}[v]_\Pi$ for w with multiplicity $\mu^{[v]_\Pi,a,w}$, and precisely one (combined) outgoing edge predicate $\xrightarrow{a}[w]_\Pi$ for v with multiplicity $\mu^{v,a,[w]_\Pi}$. Note that a monomorphic shape graph is completely characterized by the type graph T, the partitioning Π and the multiplicities μ^v for $v \in N$ and $\mu^{v,a,[w]_\Pi}, \mu^{[v]_\Pi,a,w}$ for $(v,a,w) \in E$.

We use the term *monomorphic* because, in a sense, these constraints specify a singular shape — as seen from their conjunctive form. To some degree, the non-determinism in a monomorphic shape constraint is "pushed" into the syntactic sugar, i.e., the collective predicates of the form ${}^{\mu}[\Xi]$. Another degree of variability is obtained by taking *sets* of monomorphic shape graphs to represent states. Indeed, it can be proved that sets of monomorphic shape graphs are equally expressive to arbitrary shape graphs. Formally:

Theorem 2. *For any shape graph Γ there exists a set of monomorphic shape graphs $\mathbf{\Delta}$ such that $\{\Gamma\} \equiv \mathbf{\Delta}$.*

Example 8. The shape graph Γ consisting of U in Fig. 2 as a type graph and the conjunction of (1′)–(4′) in Ex. 5 as a shape constraint is *not* monomorphic: Constraint (4′) contains a disjunction $({}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{0}}[\xleftarrow{\mathsf{next}} v_2]) \vee ({}^{\mathbf{0}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{1}}[\xleftarrow{\mathsf{next}} v_2])$ that cannot be rewritten to the appropriate form. Indeed, this is a typical example where the local shape is not singular: the constraint expresses that a Cell-node *either* has an incoming head-edge *or* an incoming next-edge. For Γ to be monomorphic, these two cases should be embodied in distinct nodes of the type graph. The more concrete type graph on the right hand side of Fig. 3 does

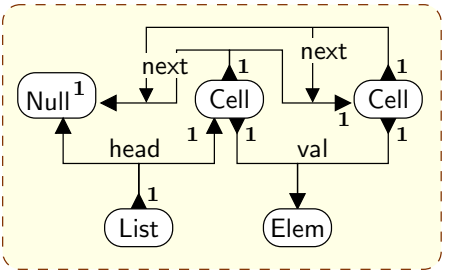


Fig. 4. A monomorphic list shape for the right hand side of Fig. 3

make this distinction and is, in fact, monomorphic. On the other hand, the similar formula $({}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_3]) \vee ({}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_3])$ in Constraint $(1')$ does *not* violate monomorphism: it is equivalent to ${}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} \{v_2, v_3\}]$.

As an example of the equivalence in Th. 2, note that the non-monomorphic Γ is equivalent to (the singleton set consisting of) the monomorphic shape graph in Fig. 4. The left hand shape in Fig. 3 gives rise to almost the same monomorphic shape graph, but without the **1**-multiplicity at the Null-node.

In the meanwhile, the single-shape property gives rise to the desired graphical representation. We can depict monomorphic shape graphs as follows:

- Nodes and edges with **0** multiplicity are omitted.
- Each node $v \in N$ receives an "outgoing edge port" for each a such that $\exists(v, a, w) \in E$; all outgoing a-edges are drawn as starting from this port, and the multiplicity $\mu^{v,a,[w]_\Pi}$, if unequal to $\top$, is written at the port.
- Likewise, v receives an "incoming edge port" for each a such that $\exists(w, a, v) \in E$; $\mu^{[w]_\Pi,a,v}$ is written there unless it equals $\top$.
- Node multiplicities unequal to $\top$ are written inside the nodes.

For instance, the monomorphic right hand shape graph of Fig. 3 is represented in this form in Fig. 4.

4.2 Canonical Shapes

Among the many different (monomorphic) shapes of a given state graph, it is useful to single out one that can be generated from the graph automatically, and whose size is bounded, not by the size of the state graph but by some global constant. For this purpose, we introduce the notion of a *canonical shape*. The idea of a canonical shape is that nodes are distinguished only if their local structure is sufficiently different in the first place.

To work this out, we just have to define "sufficiently different." In this paper we take a very straightforward notion: the multiplicities of the sets of incoming and outgoing edges should be the different for some edge label. For an arbitrary graph $G \in \mathbf{G}$, define *characteristic* functions $\gamma^G_{\rightarrow}, \gamma^G_{\leftarrow} \colon N \times \mathbf{L} \to \mathbf{M}$ as follows:

$$\gamma^G_{\rightarrow} \colon (v, a) \mapsto \#_{\mathbf{M}}\{u \mid (v, a, u) \in E\} \qquad \gamma^G_{\leftarrow} \colon (v, a) \mapsto \#_{\mathbf{M}}\{u \mid (u, a, v) \in E\} \quad .$$

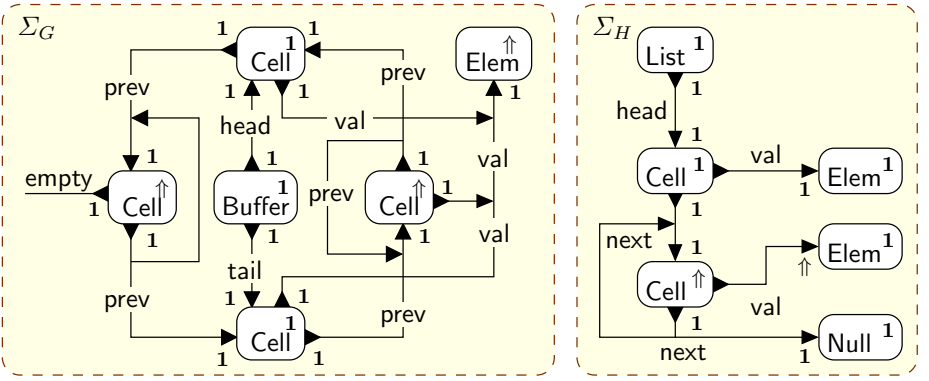


Fig. 5. Canonical shapings of the buffer and list graphs in Fig. 1

We consider $v, w \in N$ to be shape-indistinguishable if the characteristic functions yield the same result for all edge labels:

$$v \sim w \quad :\Leftrightarrow \quad \forall a \in \mathbf{L} : (\gamma^G_{\leftarrow}(v,a), \gamma^G_{\rightarrow}(v,a)) = (\gamma^G_{\leftarrow}(w,a), \gamma^G_{\rightarrow}(w,a)) \ .$$

The canonical shaping of a graph G will equate all $\sim$-equivalent nodes of G; the edge multiplicities are determined by the characteristic function.

Definition 8. *Let G be a graph and $i \in \mathbb{N}$. The* canonical shaping *of G is given by the projection morphism $\pi_\sim : G \to \Sigma$, where Σ is a monomorphic shape graph with $T_\Sigma = G/\sim$, $\Pi_\Sigma = \{N_G\}$ and for all $V \in N_\Sigma$ and $(V, a, W) \in E_\Sigma$*

$$\begin{aligned} \mu^V &= \#_{\mathbf{M}} V \\ \mu^{(V,a,N_G)} &= \gamma^G_{\rightarrow}(v,a) \ \textit{ for any } v \in V \\ \mu^{(N_G,a,W)} &= \gamma^G_{\leftarrow}(w,a) \ \textit{ for any } w \in W. \end{aligned}$$

By the definition of $\gamma^G_{\leftarrow}$ and $\gamma^G_{\rightarrow}$, the choice of $v \in V$ resp. $w \in W$ in the side conditions is irrelevant.

Example 9. Fig. 5 shows the canonical shapes Σ_G and Σ_H of the circular buffer instance G and the list instance H in Fig. 1. Note that the (only) Elem-node in Σ_G has multiplicity $\Uparrow$ but incoming edge multiplicity **1**. This indicates that there are multiple instances of this node, but they are not shared. In contrast, Σ_H has two distinct Elem-nodes, one of which is shared whereas the other is not.

By construction, canonical shapes are monomorphic; but we can actually characterize them more closely than that. For an arbitrary monomorphic shape graph Γ we again define characteristic functions $\gamma^\Gamma_{\rightarrow}, \gamma^\Gamma_{\leftarrow} : N \times \mathbf{L} \to \mathbf{M}$, as follows:

$$\begin{aligned} \gamma^\Gamma_{\rightarrow} &: (v,a) \mapsto \bigoplus \{\mu^{v,a,[w]_\Pi} \mid (v,a,w) \in E\} \\ \gamma^\Gamma_{\leftarrow} &: (v,a) \mapsto \bigoplus \{\mu^{[w]_\Pi,a,v} \mid (w,a,v) \in E\} \ . \end{aligned}$$

Definition 9. *A monomorphic shape graph Γ is called* canonical *if $\Pi = \{N_\Gamma\}$ and $\forall a \in \mathbf{L} : (\gamma^\Gamma_{\leftarrow}(v,a), \gamma^\Gamma_{\rightarrow}(v,a)) = (\gamma^\Gamma_{\leftarrow}(w,a), \gamma^\Gamma_{\rightarrow}(w,a))$ implies $v = w$.*

The following proposition states that this is a valid characterization of the canonical shapings obtained from Def. 8.

Proposition 2. *Σ_G is a canonical shape graph for any $G \in \mathbf{G}$.*

We propose canonical shape graphs as a workable state space abstraction. The following theorem states that the set of canonical shape graphs is finite, albeit still very large. As a corollary, it follows that, in contrast to monomorphic shapes, canonical shapes are not as expressive as general shape graphs: the number of inequivalent shape graphs is infinite.

Theorem 3. *Assume* $\mathbf{L}$ *to be a finite set.*

1. *The number of canonical shape graphs is $O\big(2^{n^2 \cdot \#\mathbf{L}} \cdot \#\mathbf{M}^n\big)$ with $n = \#\mathbf{M}^{2\#\mathbf{L}}$.*
2. *The number of canonical shape graphs over a type graph T is $O\big(\#\mathbf{M}^{n+2m}\big)$, where $n = \#N_T$ and $m = \min(\#E_T, \#N_T \cdot \#\{a \mid (p, a, q) \in E_G\}$.*

The upper bound can be improved somewhat (see [16]): $\#\mathbf{M}$ can be replaced by the number of *minimal consistent multiplicities*; for instance, in our chosen multiplicity algebra this is the set $\{\mathbf{0}, \mathbf{1}, \Uparrow\}$, hence $\#\mathbf{M} = 6$ can be improved to 3. Even so, however, the number of canonical shapes, even over a fixed type graph, is exponential. Still, canonical shape graphs do provide an automatic abstraction from possibly unbounded state graphs to a finite operational model. It will remain to be seen how well the abstraction performs in practice.

5 Conclusion

We discuss below some related work, which is in some cases quite close to that presented in the current paper. In this comparison we stress similarities not differences. Before doing so, however, we point out some aspects of our approach that, combined, distinguish our work from the papers cited below:

- The explicit use of a multiplicity algebra.
- The use of a decidable, but still quite expressive, logic over type graphs.
- The graphical representation of monomorphic shapes.
- The automatic abstraction to (bounded) canonical shapes.

Related work. The logic presented here is closely related to logics developed for *static analysis*, especially based on shape graphs, with Benedikt, Reps and Sagiv [3] as a prime example — but see also [9,12]. The main difference is that their approach is more language-oriented and models states as *stores*, in which pointer variables are distinguished from field selectors. For instance, the logic L_r defined in [3] can express the existence of paths between pointer variables, and node sharing along (forward) paths from pointer variables — where the pointer variables are fixed, i.e., no quantification is possible. We conjecture that LSL and L_r are independent.

In a setting more similar to ours, Baldan, König and König [2] have recently defined a graph abstraction logic for essentially the same purpose as LSL. The main difference is that their logic does not have nodes but edges as basic entities.

Another logic for graphs is studied by Cardelli, Gardner and Ghelli in [4]. This so-called *spatial* logic is intended for the purpose of *querying* graphs but looks to be suited also for *typing* them. It allows quantification over both nodes and edges and appears to be properly encompassing SL.

Also fairly recently, O'Hearn, Reynolds and Yang have proposed, in a series of papers [14,20,17], a branch of logic called *separation logic* for reasoning about dynamic storage allocation. A prime feature of the logic is the ability to express the partitioning of a graph into disjoint subgraphs — a feature also present in spatial logic. Apart from this commonality, however, separation logic is more similar in spirit to that by Sagiv et al., discussed above.

There is a close connection of LSL to *description logic*, a long-standing approach to knowledge representation; see [1] for an overview. One important feature of description logic that is missing altogether from LSL is the *intersection of concepts* — which in our setting would be represented best by an *inheritance* between nodes. In the theory of graph types, we have only seen this in a recent paper by Ferreira and Ribeiro [8]. Taking inspiration from description logic, node inheritance is an issue we intend to investigate in the future.

Expressiveness and decidability. One of our main results is the decidability of LSL (Cor. 1). This is actually a consequence of prior decidability results in first-order logic, though we became aware of the relevant facts only after finishing the work reported here. Briefly: any LSL formula containing just multiplicities $\uparrow$ is equivalent to a formula of $\mathcal{L}^2$, first order logic on two variables. This fragment was shown a long time ago to be decidable by Scott [19] and later to be NEXPTIME-complete [13,10]. Moreover, an *arbitrary* formula of LSL is equivalent to a formula in $\mathcal{C}^2$, two-variable first-order logic *with counting quantifiers*, which was shown decidable in [11]. In fact, LSL is strictly less expressive than $\mathcal{C}^2$. See [16] for a more extensive overview.

References

1. F. Baader, ed. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
2. P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In R. Cousot, ed., *Static Analysis*, vol. 2694 of *LNCS*, pp. 255–272. Springer, 2003.
3. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In S. D. Swierstra, ed., *European Symposium on Programming*, vol. 1576 of *LNCS*, pp. 2–19. Springer, 1999.
4. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In P. Widmayer et al., ed., *Automata, Languages and Programming*, vol. 2380 of *LNCS*, pp. 597–610. Springer, 2002.
5. B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, vol. A, pp. 193–239. Elsevier, 1990.
6. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro, eds., *Foundations of Information Technology in the Era of Network and Mobile Computing*, vol. 223 of *IFIP Conference Proceedings*, pp. 435–447. Kluwer Academic Publishers, 2002.

7. D. Distefano, A. Rensink, and J.-P. Katoen. Who is pointing when to whom: On model-checking pointer structures. CTIT Technical Report TR–CTIT–03–12, Department of Computer Science, University of Twente, Sept. 2003.
8. A. P. L. Ferreira and L. Ribeiro. Towards object-oriented graphs and grammars. In U. Nestmann and P. Stevens, eds., *Formal Methods for Open Object-Oriented Distributed Systems*, vol. 2884 of *LNCS*, pp. 16–31. Springer, 2003.
9. P. Fradet and D. Le Métayer. Shape types. In *Principles of Programming Languages*, pp. 27–39. ACM Press, 1997.
10. E. Grädel, P. G. Kolatis, and M. Y. Vardi. On the decision problem for two-variable first-order logic. *The Bulletin of Symbolic Logic*, 3(1):53–69, Mar. 1997.
11. E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Logic in Computer Science*, pp. 306–317. Computer Society Press, 1997.
12. N. Klarlund and M. I. Schwartzbach. Graph types. In *Principles of Programming Languages*, pp. 196–205. ACM Press, Jan. 1993.
13. M. Mortimer. On languages with two variables. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:135–140, 1975.
14. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, ed., *CSL 2001*, vol. 2142 of *LNCS*, pp. 1–19. Springer, 2001.
15. C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.
16. A. Rensink. A logic of local graph shapes. CTIT Technical Report TR–CTIT–03–35, Faculty of Informatics, University of Twente, Aug. 2003.
17. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*. Computer Society Press, 2002.
18. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, 20(1):1–50, Jan. 1998.
19. D. Scott. A decision method for validity of sentences in two variables. *J. Symb. Log.*, 27:477, 1962.
20. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, eds., *Foundations of Software Science and Computation Structures*, vol. 2303 of *LNCS*, pp. 402–416. Springer, 2002.