Graphs and Decidable Transductions based on Edge Constraints
(Extended Abstract)

Nils Klarlund* & Michael I. Schwartzbach**

Aarhus University, Department of Computer Science,
Ny Munkegade, DK-8000 Århus, Denmark
{klarlund,mis}@daimi.aau.dk

Abstract. We give examples to show that not even **c-edNCE**, the most general known notion of context-free graph grammar, is suited for the specification of some common data structures.
To overcome this problem, we use monadic second-order logic and introduce *edge constraints* as a new means of specifying a large class of graph families. Our notion stems from a natural dichotomy found in programming practice between ordinary pointers forming spanning trees and auxiliary pointers cutting across.
Our main result is that for certain transformations of graphs definable in monadic second-order logic, the question of whether a graph family given by a specification $\mathcal{A}$ is mapped to a family given by a specification $\mathcal{B}$ is decidable. Thus a decidable Hoare logic arises.

1 Introduction

Graphs are complicated objects to describe. Thus various grammars and logics have emerged for their representation, see the chapter by Courcelle [1]. The *monadic second-order logic of graphs* (M2L-G) allows a very large class of graph families to be described. The first-order terms of the logic denote nodes. The second-order terms denote sets of nodes. Nodes and edges are related by built-in predicates. The M2L-G formalism is very well-suited for describing properties of some common data structures, see our earlier paper [5].

Some authors consider logics that comprise quantification over edges. For these logics, a fundamental result is that a family of graphs allows a decidable M2L if and only if the family is specified by a *hyperedge-replacement grammar* [2]. Such grammars constitute a natural generalization of context-free grammars for string languages.

An even larger class of context-free grammars is known as **c-edNCE**. The monadic logic of graph families thus given is undecidable, but certain other questions, such a non-emptiness of a specification, are decidable, see [4].

* The author is supported by a fellowship from the Danish Research Council.
** The author is partially supported by the BRICS Center under the Danish Research Foundation.

For programming purposes, we would like to describe common data structures found in the store such as trees and doubly-linked lists. Indeed, this is possible within the framework of decidable formalisms as e.g. hyperedge-replacement grammars. Many other graph shapes are not representable. But whatever specification formalism we choose, we should be able to represent trees with additional, unconstrained pointers—reflecting a situation where almost nothing is said about the store, as is the case with type systems of most imperative programming languages.

We show in this paper that not even **c-edNCE** grammars are able to define such families of graphs.

To reason about data structures, it is vital to model the execution of programs. Therefore, we must formulate ways of transforming graphs corresponding to statements in a programming language. For program correctness, we would use Hoare logic to show that the store transformations leave the graph specifications satisfied.

In this paper we consider restricted graph transformations, called *transductions*, which are based on the method of *semantic interpretation* [7] and studied in [3]. Given logical graph specifications $\mathcal{A}$ and $\mathcal{B}$ and a transduction, we address the problem of verifying what we call *transductional correctness*: for any graph satisfying $\mathcal{A}$, any graph resulting from the transduction satisfies $\mathcal{B}$. This informal definition omits the difficulty of having shared logical variables in $\mathcal{A}$ and $\mathcal{B}$—a problem that is explicitly solved in this paper. Decidability of transductional correctness amounts to decidability of the corresponding Hoare logic.

Contributions of this paper

We devise a class of graph specifications

- that may model loosely restrained edges, and
- for which transductional correctness is decidable.

Our graphs consist of *ordinary edges* constituting an underlying spanning forest, called the *backbone*, and *auxiliary edges* cutting across the backbone.

These notions stem from a natural dichotomy found in programming practice between ordinary pointers forming spanning trees and auxiliary pointers cutting across as used for short-cuts (such as extra links pointing backward to previous elements) or for indexing into other data structures using unrestrained pointers.

Our graph specifications are based on combining the full M2L in form of a *backbone formula* for specifying ordinary edges together with a special M2L syntax, called *edge constraints*, for specifying auxiliary edges. The formulas in an edge constraint involve only the backbone to specify the sources and destinations of auxiliary edges. The resulting class of graph families thus definable is called **EC**. We show that the classes **c-edNCE** and **EC** are incomparable.

We next introduce a class of transductions. They are formulated in M2L and are similar to the ones considered in [3]. We use extra logical variables to model edges that are followed, deleted, or added during the transformation of the graph.

Our main result is that the transduction problem is decidable for **EC**. This result is based on a rather complicated encoding of the effects of the transduction within M2L on the backbone alone. The obstacle that we overcome is that it is impossible to directly represent all auxiliary edges in the logic of the backbone. The key idea is to distinguish between the bounded number of auxiliary edges that are explicitly manipulated by the transduction and the others, which are represented by a universal quantification in the logic.

Our other work

In an accompanying paper [6], we outline a typing system for data structures and define a programming language. The typing information is expressed in a logic on the underlying recursive data types. The programming language provides assignment, dereference, allocation, deallocation, and limited forms of iterations based on regular walks. We show in [6] that the operational semantics is captured by transductions and that by the results in this paper the resulting Hoare logic on data structures is decidable.

In [5], we also used monadic second-order logic to reason about data structures as graphs, but we restricted ourselves to trees with auxiliary edges that are functionally determined by the backbone in terms of regular walks.

2 Rooted Graphs

A *graph alphabet* Λ consists of a finite set $\Lambda^{\mathbf{V}}$ of node labels (which include a special label **spare**) and a finite set $\Lambda^{\mathbf{E}}$ of edge labels. Usually, we denote a node label by v. There are two kinds of edge labels: *ordinary* and *auxiliary*. Usually, an ordinary edge label is denoted f and an auxiliary edge label is denoted a. An edge label that is either ordinary or auxiliary is denoted n.

A *rooted graph* G over Λ consists of a finite set $G^{\mathbf{V}}$ of labeled nodes; a finite set $G^{\mathbf{E}}$ of labeled edges; and a finite set of node variables x, called *roots*, denoting nodes in GThe label of node $v \in G^{\mathbf{V}}$ is denoted $G^{\mathbf{L}}(v)$. Nodes are either *ordinary* or *spare* according to their label. An edge from v to w labeled n is denoted (v, n, w). For each v and n, there is at most one such edge. Loops are allowed. The edges of G are divided into *ordinary* and *auxiliary* ones according to their label. The node denoted by root x is written x^G.

The set of all graphs over Λ is denoted $\mathbf{GR}(\Lambda)$. An *edge set* $\mathcal{E}$ is a set of edges such that $(v, \mathsf{n}, w) \in \mathcal{E}$ and $(v, \mathsf{n}, u) \in \mathcal{E}$ implies $w = u$.

We sometimes view G as consisting of $\overline{G}$, called the *backbone*, which is all of G except for the auxiliary edges, and $\overline{\overline{G}}$, which is the edge set of auxiliary edges in G. Thus, G may be written as $(\overline{G}, \overline{\overline{G}})$.

The spare nodes model free memory cells in programming language applications. They are essential to allow addition and deletion of nodes by transductions.

Figure 1 shows a sketch of a rooted graph. The ordinary edges are drawn as solid arrows, whereas the auxiliary edges are dashed; spare nodes are black; the roots are called x_1, x_2, and x_3.

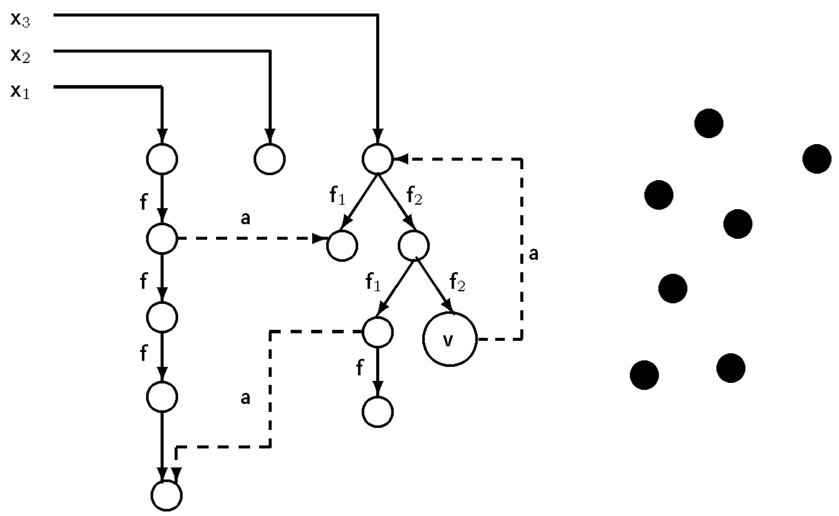

Fig. 1. A rooted graph.

3 The Logic M2L-BB

The key to specifying data structures is the *Monadic Second-Order of Backbones*, abbreviated *M2L-BB*. First-order terms range over nodes in the graph. Second-order terms range over *sets* of nodes.

Syntax

Assume a graph alphabet $\mathit{\Lambda}$. The logic of rooted graphs over $\mathit{\Lambda}$ is denoted M2L-BB($\mathit{\Lambda}$). Its syntax is as follows.

Address terms $\boldsymbol{A}$ denote nodes in the graph.

$\boldsymbol{A}$	::=	x	root
		src	source
		dst	destination
		$\alpha, \beta, \ldots$	first-order variable

The terms **src** and **dst** are special variables used in certain assertions. *Address set terms* $\boldsymbol{\Sigma}$ denote sets of nodes.

$\boldsymbol{\Sigma}$	::=	$\emptyset$	empty set
		$\boldsymbol{\Sigma}_1 \cup \boldsymbol{\Sigma}_2$	set union
		$\boldsymbol{\Sigma}_1 \setminus \boldsymbol{\Sigma}_2$	set difference
		$S, T, \ldots$	second-order variable

Formulas $\boldsymbol{\Phi}$ denote **true** or **false**.

$\boldsymbol{\Phi}$	::=	$\boldsymbol{A}_1 = \boldsymbol{A}_2$	equality

$\boldsymbol{A} \in \boldsymbol{\Sigma}$	set membership
$\boldsymbol{\Sigma}_1 \subseteq \boldsymbol{\Sigma}_2$	set inclusion
$\boldsymbol{A}_1 \xrightarrow{\mathsf{f}} \boldsymbol{A}_2$	successor relation, where $\mathsf{f} \in \Lambda^{\mathbf{E}}$ is ordinary
$\mathsf{v}?\boldsymbol{A}$	test for node label, where $\mathsf{v} \in \Lambda^{\mathbf{V}}$
$\neg\boldsymbol{\Phi}$	negation
$\boldsymbol{\Phi}_1 \wedge \boldsymbol{\Phi}_2$	conjunction
$\exists^0 \alpha : \boldsymbol{\Phi}$	first-order quantification over all nodes
$\exists^0 S : \boldsymbol{\Phi}$	second-order quantification over all nodes

Note that the syntax does not allow references to auxiliary edges. We also use unmarked quantifiers that range only over ordinary nodes. They can be viewed as abbreviations according to the following.

$$\exists\alpha : \boldsymbol{\Phi} \equiv \exists^\circ\alpha : \neg\mathbf{spare}?\alpha \wedge \boldsymbol{\Phi}$$
$$\exists S : \boldsymbol{\Phi} \equiv \exists^\circ S : (\neg\exists^\circ\alpha : \alpha \in S \wedge \mathbf{spare}?\alpha) \wedge \boldsymbol{\Phi}$$

We also assume abbreviations $\forall$, $\Rightarrow$, $\vee$, etc.

Semantics

M2L-BB is interpreted relative to a backbone $\overline{G}$. The interpretation of x is given by $\overline{G}$ as $\mathsf{x}^{\overline{G}}$. The constants **dst** and **src** are used as variables. The semantics of variables is formulated below by substitution for values in $\overline{G}^{\mathbf{V}}$. A value v is interpreted as itself, i.e. $v^{\overline{G}} = v$. A non-variable address set term $\boldsymbol{\Sigma}$ is interpreted as follows.

$$\begin{array}{ll}
\emptyset^{\overline{G}} & = \emptyset \\
(\boldsymbol{\Sigma}_1 \cup \boldsymbol{\Sigma}_2)^{\overline{G}} & = \boldsymbol{\Sigma}_1^{\overline{G}} \cup \boldsymbol{\Sigma}_2^{\overline{G}} \\
(\boldsymbol{\Sigma}_1 \backslash \boldsymbol{\Sigma}_2)^{\overline{G}} & = \boldsymbol{\Sigma}_1^{\overline{G}} \backslash \boldsymbol{\Sigma}_2^{\overline{G}}
\end{array}$$

The semantics of formulas is as follows.

$$\begin{array}{ll}
\overline{G} \vDash \boldsymbol{A}_1 = \boldsymbol{A}_2 & \text{if } \boldsymbol{A}_1^{\overline{G}} = \boldsymbol{A}_2^{\overline{G}} \\
\overline{G} \vDash \boldsymbol{A} \in \boldsymbol{\Sigma} & \text{if } \boldsymbol{A}^{\overline{G}} \in \boldsymbol{\Sigma}^{\overline{G}} \\
\overline{G} \vDash \boldsymbol{\Sigma}_1 \subseteq \boldsymbol{\Sigma}_2 & \text{if } \boldsymbol{\Sigma}_1^{\overline{G}} \subseteq \boldsymbol{\Sigma}_2^{\overline{G}} \\
\overline{G} \vDash \boldsymbol{A}_1 \xrightarrow{\mathsf{f}} \boldsymbol{A}_2 & \text{if } (\boldsymbol{A}_1^{\overline{G}}, \mathsf{f}, \boldsymbol{A}_2^{\overline{G}}) \in \overline{G}^{\mathbf{E}} \\
\overline{G} \vDash \mathsf{v}?\boldsymbol{A} & \text{if } \overline{G}^{\mathbf{L}}(\boldsymbol{A}^{\overline{\mathcal{G}}}) = \mathsf{v} \\
\overline{G} \vDash \neg\boldsymbol{\Phi} & \text{if not } \overline{G} \vDash \boldsymbol{\Phi} \\
\overline{G} \vDash \boldsymbol{\Phi}_1 \wedge \boldsymbol{\Phi}_2 & \text{if } \overline{G} \vDash \boldsymbol{\Phi}_1 \text{ and } \overline{G} \vDash \boldsymbol{\Phi}_2 \\
\overline{G} \vDash \exists^\circ\alpha : \boldsymbol{\Phi} & \text{if there is } v \in \overline{G}^{\mathbf{V}} \text{ such that } \overline{G} \vDash \boldsymbol{\Phi}(\alpha \mapsto v) \\
\overline{G} \vDash \exists^\circ S : \boldsymbol{\Phi} & \text{if there is } V \subseteq \overline{G}^{\mathbf{V}} \text{ such that } \overline{G} \vDash \boldsymbol{\Phi}(S \mapsto V),
\end{array}$$

If $\boldsymbol{\Phi}$ has free variables $\mathfrak{F}$ and $\underline{\mathfrak{F}}$ is an interpretation of these variables in $\overline{G}^{\mathbf{V}}$, then

$$\overline{G}, \underline{\mathfrak{F}} \vDash \boldsymbol{\Phi} \;\text{ if } \overline{G} \vDash \boldsymbol{\Phi}(\mathfrak{F} \mapsto \underline{\mathfrak{F}}).$$

If $\overline{G} \vDash \boldsymbol{\Phi}$ holds for all $\overline{G}$, then we say that $\boldsymbol{\Phi}$ is *valid* and we write $\vDash \boldsymbol{\Phi}$. A graph G is *tree-formed* if

– all edges are between ordinary nodes; and
– the graph induced by ordinary nodes and ordinary edges is a directed forest such that each root is the value of some root variable.

Note that the graph depicted in Figure 1 is tree-formed.

Lemma 1. *There is a formula $\boldsymbol{\Phi}$ such that G is tree-formed if and only if $G \vDash \boldsymbol{\Phi}$.*

Proof Among other conditions, acyclicity and reachability can be encoded in M2L-BB. □

We say that $\boldsymbol{\Phi}$ is *tree-valid* and we write $\Vdash \boldsymbol{\Phi}$ if $\overline{G} \vDash \boldsymbol{\Phi}$ holds for all tree-formed $\overline{G}$.

Theorem 2. *Validity is undecidable, but tree-validity is decidable.*

Proof The first result follows from the undecidability of the first-order logic of finite graphs. The second result follows from the decidability of the monadic second-order logic of finite trees. □

Edge Constraints and Assertions

Constraints on auxiliary edges cannot just be formulas, since the logic refers only to ordinary edges. Instead, an *edge constraint* is of the form $[\boldsymbol{\sigma} \xrightarrow{\mathsf{a}} \boldsymbol{\delta}]$, where $\boldsymbol{\sigma}$ is a formula involving **src** as a free variable, and $\boldsymbol{\delta}$ is a formula with free variables **src** and **dst**. The edge constraint is *valid* for a given graph if whenever $\boldsymbol{\sigma}$ is valid with a node v in place of **src**, then there is an a-edge (which is unique by definition of a rooted graph) from v to some node w and $\boldsymbol{\delta}$ is valid with v and w in place of **src** and **dst**. Note that the edge constraint does not describe any a-edges outside where $\boldsymbol{\sigma}$ holds.

Formally, let $[\boldsymbol{\sigma} \xrightarrow{\mathsf{a}} \boldsymbol{\delta}]$ be an edge constraint with free variables $\mathfrak{F}$. We say that G and $\underline{\boldsymbol{\mathfrak{F}}}$ *satisfy* $[\boldsymbol{\sigma} \xrightarrow{\mathsf{a}} \boldsymbol{\delta}]$, and we write $G, \underline{\boldsymbol{\mathfrak{F}}} \vDash [\boldsymbol{\sigma} \xrightarrow{\mathsf{a}} \boldsymbol{\delta}]$ if:

$$\begin{array}{l} \text{for all } v \in G^{\mathsf{V}},\ G, \underline{\boldsymbol{\mathfrak{F}}} \vDash \boldsymbol{\sigma}(\mathbf{src} \mapsto v) \text{ implies} \\ \qquad \text{for some } (v, \mathsf{a}, w) \in \overline{G},\ G, \underline{\boldsymbol{\mathfrak{F}}} \vDash \boldsymbol{\delta}(\mathbf{src} \mapsto v, \mathbf{dst} \mapsto w). \end{array}$$

An *assertion* $\mathcal{A} = \boldsymbol{\Phi}[\boldsymbol{\sigma}_1 \xrightarrow{\mathsf{a}_1} \boldsymbol{\delta}_1] \ldots [\boldsymbol{\sigma}_n \xrightarrow{\mathsf{a}_n} \boldsymbol{\delta}_n]$ consists of a formula $\boldsymbol{\Phi}$, called the *backbone formula*, and a number of edge constraints $[\boldsymbol{\sigma}_i \xrightarrow{\mathsf{a}_i} \boldsymbol{\delta}_i]$. These components are connected through free variables, which are implictly existentially quantified.

Let $\mathfrak{F}$ be a list containing the free variables and let $\underline{\boldsymbol{\mathfrak{F}}}$ be a value assignment to these variables. An assertion $\mathcal{A}$ is *satisfied* in G with $\underline{\boldsymbol{\mathfrak{F}}}$, and we write $G, \underline{\boldsymbol{\mathfrak{F}}} \vDash \mathcal{A}$, if $\overline{G}, \underline{\boldsymbol{\mathfrak{F}}} \vDash \boldsymbol{\Phi}$ and for all i, $G, \underline{\boldsymbol{\mathfrak{F}}} \vDash [\boldsymbol{\sigma}_i \xrightarrow{\mathsf{a}_i} \boldsymbol{\delta}_i]$.

An assertion $\mathcal{A}$ specifies the language of graphs

$$\{G \mid G \text{ is tree-formed and for some } \underline{\boldsymbol{\mathfrak{F}}},\ G, \underline{\boldsymbol{\mathfrak{F}}} \vDash \mathcal{A}\}$$

The class of such graph languages is called **EC**.

Example

Consider the common data structure, shown in Figure 2, of linked lists with a head node that points both to the first element of the list and to some designated element. The f- and n-edges are ordinary; the s-edge is auxiliary.

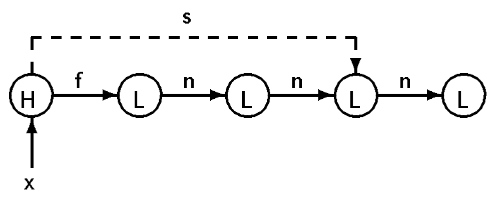

Fig. 2. A list structure

The corresponding backbone formula contains these clauses.

$\mathsf{H}?\mathsf{x}$	*The head node has label* H
$\exists\alpha : \mathsf{x} \xrightarrow{\mathsf{f}} \alpha$	*and an outgoing* f*-edge;*
$\forall\beta,\beta' : \beta \xrightarrow{\mathsf{f}} \beta' \Rightarrow \beta = \mathsf{x}$	*no other node has an outgoing* f*-edge;*
$\forall\beta : \neg\beta{=}\mathsf{x} \Rightarrow \mathsf{L}?\beta$	*all other nodes have label* L*;*
$\forall\beta,\beta' : \beta \xrightarrow{\mathsf{n}} \beta' \Rightarrow \beta \neq \mathsf{x}$	*the head node has no outgoing* n*-edge;*
$\mathsf{L}?\gamma$	*and there is a designated* L*-node...*

Note that we quantify only over ordinary nodes. There is only a single edge constraint.

$[\mathsf{H}?\mathbf{src} \xrightarrow{\mathsf{s}} \gamma = \mathbf{dst}]$	*that is the destination of the* s*-edge.*

Here the free variable γ connects the backbone formula and the edge constraint. In conjunction with the general requirement of tree-formedness, this assertion describes backbones that are lists with a head node. Note that the assertion does not eliminate extraneous s-edges from nodes other than the one marked H. In a programming language application these are avoided through elementary type-checking of the transductions that build graphs [6].

4 Relations to Other Formalisms

It is interesting to compare the expressive power of this graph specification formalism with those of other proposals. In particular we show in this section that the set of trees with unrestrained auxiliary edges is not representable as a context-free graph grammar.

We look at the most general class known of context-free graphs languages: **c-edNCE**, which stands for "**c**onfluent **e**dge and node labeled,**d**irected graphs given by **N**eighborhood **C**ontrolled **E**mbedding." The grammars that define such languages are complicated. Instead we shall use a result by Engelfriet that these languages are exactly the images of trees under functions definable in monadic second-order logic [4]. The following definition is from [4] (but changed as to allow loops in graphs):

Let Λ_1 and Λ_2 be alphabets. An *M2L-definable function* $f : \mathbf{GR}(\Lambda_1) \rightarrow \mathbf{GR}(\Lambda_2)$ is given by the following formulas in M2L-BB(Λ_1):

- a closed formula ϕ_{dom}, called the *domain formula*;
- for every $\mathsf{v} \in \Lambda_2^{\mathbf{V}}$, a formula ϕ_{v}, called a *node formula*, with one free variable **src**; and
- for every $\mathsf{n} \in \Lambda_2^{\mathbf{E}}$, a formula ϕ_{n}, called an *edge formula*, with two free variables **src** and **dst**.

The domain of f is $\{G \in \mathbf{GR}(\Lambda_1) \mid G \vDash \phi_{dom}\}$. For every $G \in \mathrm{dom}(f)$, the graph $G' = f(G) \in \mathbf{GR}(\Lambda_2)$ is given by

$$G'^{\mathbf{V}} = \{v \in G^{\mathbf{V}} \mid \text{there is exactly one } \mathsf{v} \in \Lambda_1^{\mathbf{V}} \text{such that } G \vDash \phi_{\mathsf{v}}(\mathbf{src} \mapsto v)\}$$

$$G'^{\mathbf{E}} = \{(v, \mathsf{n}, w) \mid v, w \in G^{\mathbf{V}} \text{and } G \vDash \phi_{\mathsf{n}}(\mathbf{src} \mapsto v, \mathbf{dst} \mapsto w)\}.$$

(For simplicity, we ignore roots in this section.)

Theorem 3. *[4] A language of graphs is* **c-edNCE** *if and only if it is the image of an M2L-definable function* $f : \mathbf{GR}(\Lambda_1) \rightarrow \mathbf{GR}(\Lambda_2)$ *applied to the set of directed trees over* Λ_1.

Such a language is then said to be *f-definable*.

Theorem 4. *[4] It is decidable whether a function* f *defines a finite language of graphs.*

Lemma 5. *[4] The class of M2L-definable functions is closed under composition.*

Now fix $\Lambda_T^{\mathbf{V}} = \{\mathsf{v}\}$, $\Lambda_T^{\mathbf{E}} = \{\mathsf{f}_1, \mathsf{f}_2, \mathsf{a}\}$. A *tree with equi-level edges* is a graph G over Λ_T such that G restricted to f-edges is a directed tree and such that $(v, \mathsf{a}, w) \in G^{\mathbf{E}}$ if and only if w is the left-most node to the right of v at the same level as v, as shown in Figure 3.

Lemma 6. *The set of trees over* Λ_T *with equi-level edges is not* **c-edNCE**.

Proof Suppose for a contradiction that the set is **c-edNCE** by means of an M2L-definable function f. Then there would be a uniform way of obtaining an M2L-definable function f_i whose graph language represents all finite sequences of configurations that TM (Turing Machine) i may produce with an empty input tape. In fact we may choose $\Lambda^{\mathbf{V}} = \{0, 1, \#\}$ and construct f_i' such that it maps trees with equi-level edges into trees whose $\Lambda^{\mathbf{V}}$ labels at level k encode the configuration of TM i after the k'th step (details are omitted). By Lemma 5, the set of graphs representing finite configuration sequences is then definable by a function $f_i = f_i' \circ f$. But then the Halting Problem would be decidable by Theorem 4, which is a contradiction. □

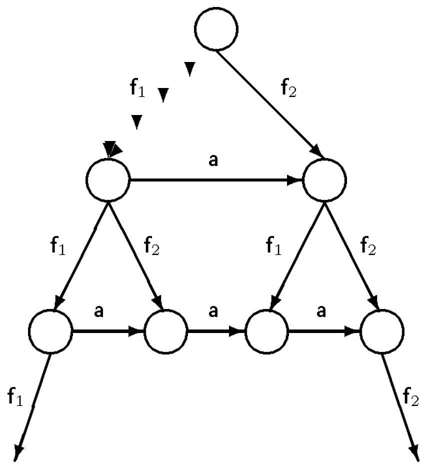

Fig. 3. A tree with equi-level edges.

Lemma 7. *The set of trees over A_T with unrestrained* a*-edges is not* **c-edNCE**.

Proof If it was we could use Lemmas 5 and 6 to show that also the set of trees with equi-level edges is **c-edNCE**. (We would construct a domain formula checking, among other things, that whenever (v, a, w) and (v', a, w') are edges and v' is a child of v, then w' is a child of w.) □

Theorem 8. **c-edNCE** *and* **EC** *are incomparable.*

Proof EC $\not\subseteq$ **c-edNCE**: The set of trees with unrestrained a-edges is certainly **EC**, but not **c-edNCE** by Lemma 7.

c-edNCE $\not\subseteq$ **EC**: The set of cyclic graphs over singleton node and edge alphabets is **c-edNCE**, but not **EC** (in fact, since the edge label determines whether an edge is ordinary or auxiliary, only list-like structures and certain degenerate structures can be described with singleton edge alphabets). □

5 Transductions

We are interested in graph transformations that model pointer manipulations in programs. These can be specified through a *transduction*, which is defined to be of the form $\mathcal{T} =< L, \mathcal{E}, \rho >$. The component L is a list of labeled *entries*. An entry t defines one or two first-order variables, called *transduction variables*, according to its label as follows.

– **add**-n: this indicates the creation of an n-edge between two nodes denoted by first-order terms $\mathbf{src}(t)$ and $\mathbf{dst}(t)$; an existing n-edge from the source is deleted.

- **del**-n: this indicates the deletion of the n-edge whose origin is denoted by the first-order term $\mathbf{src}(t)$.
- **foll**-a: this indicates the existence of an a-edge which has been followed between two cells denoted by first-order terms $\mathbf{src}(t)$ and $\mathbf{dst}(t)$; this makes for an explicit representation of auxiliary edges that are followed and, therefore, known to exist in the original graph.
- v: this indicates that a node denoted by the first-order logical variable $\mathbf{src}(t)$ is marked with label v (which may be **spare**); if an ordinary node is marked **spare**, then its outgoing and incoming edges are deleted.

The component $\mathcal{E}$ is an environment, which maps root variables to address terms denoting their values. The component ρ is a formula which must hold in order for the free variables in L and $\mathcal{E}$ to denote a transformation. The formula ρ may contain other transduction variables than those defined by L. Together they are designated $\boldsymbol{\mu}$.

The formula ρ must ensure that the entries are consistent with each other. Thus if a graph G and a value assignment $\underline{\boldsymbol{\mu}}$ are such that $G, \underline{\boldsymbol{\mu}} \models \rho$, then some examples of technical relationsships that most hold are:

- given any v and a, there are at most one **foll**-a entry t such that $G, \underline{\boldsymbol{\mu}} \models \mathbf{src}(t) = v$; and
- given any (v, a, w) that is marked by a **del**-a entry before any **add**-a entry, there is a **foll**-a entry, which makes explicit the assumption that (v, a, w) is an edge in G.

6 Predicate Transformers

Each transduction $\mathcal{T}$ determines a predicate transformer $\mathbf{Tr}_{\mathcal{T}}$. A formula $\boldsymbol{\Phi}$ is translated into $\mathbf{Tr}_{\mathcal{T}}\boldsymbol{\Phi}$ according to the following rules.

$$
\begin{aligned}
\mathbf{Tr}_{\mathcal{T}}(\mathsf{x}) &= \mathcal{T}.\mathcal{E}(\mathsf{x}) \\
\mathbf{Tr}_{\mathcal{T}}(\alpha) &= \alpha \\
\mathbf{Tr}_{\mathcal{T}}(\boldsymbol{A}_1 = \boldsymbol{A}_2) &= \mathbf{Tr}_{\mathcal{T}}(\boldsymbol{A}_1) = \mathbf{Tr}_{\mathcal{T}}(\boldsymbol{A}_2)
\end{aligned}
$$

$$
\mathbf{Tr}_{\mathcal{T}}(\alpha \xrightarrow{\mathsf{f}} \beta) = \begin{cases}
\beta = \mathbf{dst}(t) & \text{if } t \text{ is an } \mathbf{add}\text{-}\mathsf{f} \text{ entry in } \mathcal{T}.L, \ \alpha = \mathbf{src}(t), \ t \text{ is the last such entry, and no later } \mathbf{spare} \text{ entry } t' \text{ is such that } \mathbf{src}(t') \in \{\alpha, \beta\} \text{ and no later } \mathbf{del}\text{-}\mathsf{f} \text{ entry } t' \text{ is such that } \mathbf{src}(t') = \alpha \\
\mathbf{false} & \text{if there is a } \mathbf{spare} \text{ entry } t \text{ with } \mathbf{src}(t) \in \{\alpha, \beta\} \text{ or there is a } \mathbf{del}\text{-}\mathsf{f} \text{ entry } t \text{ with } \mathbf{src}(t) = \alpha, \text{ and no later } \mathbf{add}\text{-}\mathsf{f} \text{ entry } t' \text{ is such that } \mathbf{src}(t') = \alpha\} \\
\alpha \xrightarrow{\mathsf{f}} \beta & \text{otherwise}
\end{cases}
$$

$$\mathbf{Tr}_{\mathcal{T}}(\mathsf{v}?\alpha) = \begin{cases} \mathbf{true} & \text{if there is an } \mathsf{v}\text{-entry } t \text{ in } \mathcal{T}.L \text{ such that } \mathbf{src}(t) = \alpha \text{ and no later } \mathsf{v}'\text{-entry } t' \text{ is such that } \mathbf{src}(t') = \alpha \\ \mathsf{v}?\alpha & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathbf{Tr}_{\mathcal{T}}(\boldsymbol{A} \in \boldsymbol{\Sigma}) &= \mathbf{Tr}_{\mathcal{T}}(\boldsymbol{A}) \in \boldsymbol{\Sigma} \\
\mathbf{Tr}_{\mathcal{T}}(\boldsymbol{\Sigma}_1 \subseteq \boldsymbol{\Sigma}_2) &= \boldsymbol{\Sigma}_1 \subseteq \boldsymbol{\Sigma}_2 \\
\mathbf{Tr}_{\mathcal{T}}(\neg\boldsymbol{\Phi}) &= \neg\mathbf{Tr}_{\mathcal{T}}\boldsymbol{\Phi} \\
\mathbf{Tr}_{\mathcal{T}}(\boldsymbol{\Phi}_1 \wedge \boldsymbol{\Phi}_2) &= \mathbf{Tr}_{\mathcal{T}}(\boldsymbol{\Phi}_1) \wedge \mathbf{Tr}_{\mathcal{L}}(\boldsymbol{\Phi}_2) \\
\mathbf{Tr}_{\mathcal{T}}(\exists^\circ \alpha : \boldsymbol{\Phi}) &= \exists^\circ \alpha : \mathbf{Tr}_{\mathcal{T}}\boldsymbol{\Phi} \\
\mathbf{Tr}_{\mathcal{T}}(\exists^\circ S : \boldsymbol{\Phi}) &= \exists^\circ S : \mathbf{Tr}_{\mathcal{T}}\boldsymbol{\Phi}
\end{aligned}$$

The *transformed backbone*, denoted $\mathbf{BB}_{\mathcal{T}}(\overline{\mathcal{G}}, \underline{\boldsymbol{\mu}})$, according to $\mathcal{T}$ on $\overline{G}$ with transduction values $\underline{\boldsymbol{\mu}}$ is the graph $\overline{G}'$ defined as follows.

- $\overline{G}'^{\mathbf{V}} = \overline{G}^{\mathbf{V}}$;
- $(v, \mathsf{f}, w) \in \overline{G}'^{\mathbf{E}}$ iff $\overline{G}, \underline{\boldsymbol{\mu}} \vDash \mathbf{Tr}_{\mathcal{T}}(v \xrightarrow{\mathsf{f}} w)$;
- $\overline{G}'^{\mathbf{L}}(v) = \mathsf{v}$ iff $\overline{G}, \underline{\boldsymbol{\mu}} \vDash \mathbf{Tr}_{\mathcal{T}}(\mathsf{v}?v)$; and
- $\mathsf{x}^{\overline{G}'}$ is the node v such that $\overline{G}, \underline{\boldsymbol{\mu}} \vDash v = \mathbf{Tr}_{\mathcal{T}}(\mathcal{T}.\mathcal{E}(\mathsf{x}))$.

Lemma 9. *(Faithfulness) Let* $\overline{G}' = \boldsymbol{BB}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})$ *and let* $\underline{\mathfrak{F}}$ *be a value assignment to the free variables of* $\boldsymbol{\Phi}$*. Then,*

$$\overline{G}', \underline{\mathfrak{F}} \vDash \boldsymbol{\Phi}$$

if and only if

$$\overline{G}, \underline{\mathfrak{F}}, \underline{\boldsymbol{\mu}} \vDash \boldsymbol{Tr}_{\mathcal{T}}\boldsymbol{\Phi}$$

Proof (Sketch) By a straightforward structural induction. □

We say that $\overline{G}$, $\boldsymbol{\mu}$, and $\mathcal{T}$ determine a *transformation*. In addition to the transformed backbone, the transformation also determines:

- $\mathbf{Foll}_{\mathcal{T}}$-$\mathsf{a}(\overline{G}, \underline{\boldsymbol{\mu}})$, the set of a-edges in the old graph G that were followed;
- $\mathbf{Del}_{\mathcal{T}}$-$\mathsf{a}(\overline{G}, \underline{\boldsymbol{\mu}})$, the set of a-edges in the old graph G that were both followed and deleted; and
- $\mathbf{Add}_{\mathcal{T}}$-$\mathsf{a}(\overline{G}, \underline{\boldsymbol{\mu}})$, the set of a-edges in the new graph G' that were added.

To specify $\mathbf{Foll}_{\mathcal{T}}$-$\mathsf{a}(\overline{G}, \underline{\boldsymbol{\mu}})$, we define a predicate $\mathbf{Foll}_{\mathcal{T}}$-$\mathsf{a}$ with free variables $\mathbf{src}$ and $\mathbf{dst}$ expressing that an a-edge from $\mathbf{src}$ to $\mathbf{dst}$ was followed. Informally,

$\mathbf{Foll}_{\mathcal{T}}$-$\mathsf{a}$ $\equiv$ "for some $\mathbf{foll}$-a entry in $\mathcal{T}.L$, $\mathbf{src} = \mathbf{src}(t)$ and $\mathbf{dst} = \mathbf{dst}(t)$,"

which can be encoded as a formula. Now,

$$\mathbf{Foll}_{\mathcal{T}}\text{-}\mathsf{a}(\overline{G}, \underline{\boldsymbol{\mu}}) = \{(v, \mathsf{a}, w) \mid \overline{G}, \underline{\boldsymbol{\mu}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \mathbf{Foll}_{\mathcal{T}}\text{-}\mathsf{a}\}.$$

Similarly, we define the two other sets by defining predicates $\mathbf{Del}_{\mathcal{T}}$-$\mathsf{a}$ and $\mathbf{Add}_{\mathcal{T}}$-$\mathsf{a}$:

$\mathbf{Del}_{\mathcal{T}}\text{-a} \equiv$ "$\mathbf{Foll}_{\mathcal{T}}\text{-a}$ and there is some **spare** entry with $\mathbf{src} = \mathbf{src}(t)$ or $\mathbf{dst} = \mathbf{src}(t)$, or some **del**-a or **add**-a entry t with $\mathbf{src} = \mathbf{src}(t)$."

$\mathbf{Add}_{\mathcal{T}}\text{-a} \equiv$ "if there is an **add**-a entry t such that $\mathbf{src}(t) = \mathbf{src}$ and $\mathbf{dst}(t) = \mathbf{dst}$, and no later entries delete this edge."

Lemma 10. $\mathbf{Del}_{\mathcal{T}}\text{-a}(\overline{G}, \underline{\boldsymbol{\mu}}) \subseteq \mathbf{Foll}_{\mathcal{T}}\text{-a}(\overline{G}, \underline{\boldsymbol{\mu}})$ *if* $G, \underline{\boldsymbol{\mu}} \vDash \rho$.

Proof By the definitions and imposed technical relationships. □

The *transformation relation* induced by $\mathcal{T}$ is:

$$G \longrightarrow_{\mathcal{T}} G'$$

if and only if

$$\begin{aligned}&\text{for some } \underline{\boldsymbol{\mu}}:\\ &\quad \overline{G}, \underline{\boldsymbol{\mu}} \models \mathcal{T}.\rho,\\ &\quad \mathbf{Foll}\text{-a}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}}) \subseteq \overline{\overline{G}},\\ &\quad \overline{G}' = \mathbf{BB}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}}), \text{ and}\\ &\quad \overline{\overline{G}}' = (\overline{\overline{G}} \backslash \mathbf{Del}\text{-a}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})) \cup \mathbf{Add}\text{-a}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})\end{aligned}$$

Example (continued)

Consider the linked list with a designated element from Section 4. A common transduction on such structures is the insertion of an new element just before the head. This is realized by the following transduction.

L: $\mathsf{L}(\mu').\mathbf{del}\text{-}\mathsf{f}(\mathsf{x}, \mu).\mathbf{add}\text{-}\mathsf{f}(\mathsf{x}, \mu').\mathbf{add}\text{-}\mathsf{n}(\mu', \mu)$
$\mathcal{E}$: $\mathsf{x} \mapsto \mathsf{x}$
ρ: $\mathsf{x} \xrightarrow{\mathsf{f}} \mu \wedge \mathbf{spare?}\mu'$

Notice how this closely mimics the code that one would write in a conventional programming language. The expressive power of transductions goes beyond mere straight-line code, since regular control structures can be encoded in formulas [5].

7 Transductional Invariance

Let $\mathfrak{A}$ be the free variables in the assertion $\mathcal{A}$ and let $\mathfrak{B}$ be the free variables in the assertion $\mathcal{B}$ that are not already free in $\mathcal{A}$. The problem of transductional correctness is:

> Given assertions $\mathcal{A}$, $\mathcal{B}$, and a transduction $\mathcal{T}$. Does it hold for all G, G', and $\underline{\mathfrak{A}}$ that if G is tree-formed and satisfies $\mathcal{A}$ with $\underline{\mathfrak{A}}$, and if $G \longrightarrow_{\mathcal{T}} G'$, then G' is tree-formed and satisfies $\mathcal{B}$ for some $\underline{\mathfrak{B}}$?

Since tree-formedness by Lemma 1 can be encoded as a backbone formula, we can without loss of generality rephrase the question as follows. We say that the triple $\mathcal{A}\{\mathcal{T}\}\mathcal{B}$ is *tree-valid*, and write $\Vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$, if:

$$\begin{array}{l}\text{for all tree-formed } G, \text{all } G'\text{ , and all } \underline{\mathfrak{A}},\;\; G, \underline{\mathfrak{A}} \vDash \mathcal{A} \text{ and } G \longrightarrow_{\mathcal{T}} G' \\ \qquad \text{implies there is } \underline{\mathfrak{B}} \text{ such that } G', \underline{\mathfrak{B}} \vDash \mathcal{B}\end{array}$$

Note that triple tree-validity concerns only transformations of tree-formed graphs.

Our main result is to demonstrate that tree triple validity can be encoded in M2L-BB. For simplicity we assume in what follows that an assertion now contains only one edge constraint, and that $\mathcal{A} = \boldsymbol{\Phi}[\boldsymbol{\sigma} \xrightarrow{\mathsf{a}} \boldsymbol{\delta}]$ and $\mathcal{B} = \boldsymbol{\Phi}'[\boldsymbol{\sigma}' \xrightarrow{\mathsf{a}} \boldsymbol{\delta}']$. Then we say that triple $\mathcal{A}\{\mathcal{T}\}\mathcal{B}$ is *provable* and write $\vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$ if

$$\begin{array}{l}\Vdash \forall^\circ \mathfrak{A} : \forall^\circ \boldsymbol{\mu} : \\ \quad (\boldsymbol{\Phi} \;\wedge\; \rho \;\wedge\; \forall^\circ \mathbf{src} \exists^\circ \mathbf{dst} : (\boldsymbol{\sigma} \Rightarrow \;(\boldsymbol{\delta} \;\wedge\; (\neg \mathbf{Foll}_{\mathcal{T}} \Rightarrow (\forall^\circ \mathbf{dst} : \neg \mathbf{Foll}_{\mathcal{T}})))) \\ \qquad \Rightarrow \exists^\circ \mathfrak{B} : (\mathbf{Tr}_{\mathcal{T}} \boldsymbol{\Phi}' \\ \qquad\qquad\qquad \wedge \forall^\circ \mathbf{src} : \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\sigma}' \;\Rightarrow \\ \qquad\qquad\qquad\qquad\qquad ((\exists^\circ \mathbf{dst} : \mathbf{Add}_{\mathcal{T}} \;\wedge\; \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}') \\ \qquad\qquad\qquad\qquad\qquad \vee (\exists^\circ \mathbf{dst} : \mathbf{Foll}_{\mathcal{T}} \;\wedge\; \neg \mathbf{Del}_{\mathcal{T}} \;\wedge\; \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}') \\ \qquad\qquad\qquad\qquad\qquad \vee (\boldsymbol{\sigma} \;\wedge\; \forall^\circ \mathbf{dst} : \neg \mathbf{Add}_{\mathcal{T}} \wedge \neg \mathbf{Foll}_{\mathcal{T}} \wedge\; (\boldsymbol{\delta} \Rightarrow \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}'))))\end{array}$$

8 Soundness, Completeness, and Decidability

Theorem 11. *(Soundness)* $\vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$ *implies* $\Vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$.

Proof Assume

(1) $\vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$.

Fix a tree-formed G, a G', and a value assignment $\underline{\mathfrak{A}}$ to the free variables $\mathfrak{A}$ of $\mathcal{A}$ such that

(2) $G, \underline{\mathfrak{A}} \vDash \mathcal{A}$, and

(3) $G \longrightarrow_{\mathcal{T}} G'$.

To establish $\Vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$, we only need to find a value assignment $\underline{\mathfrak{B}}$ to the remaining free variables $\mathfrak{B}$ such that

(4) $G', \underline{\mathfrak{A}}, \underline{\mathfrak{B}} \vDash \mathcal{B}$.

Now by (3) and the definition of transductions, there is a value assignment $\underline{\boldsymbol{\mu}}$ to the transduction variables $\boldsymbol{\mu}$ of $\mathcal{T}$ such that

(5) $\overline{G}, \underline{\boldsymbol{\mu}} \models \mathcal{T}.\rho$

(6) $\mathbf{Foll}_{\mathcal{T}}(\overline{S}, \underline{\boldsymbol{\mu}}) \subseteq \overline{\overline{G}}$,

(7) $\overline{G}' = \mathbf{BB}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})$, and

(8) $\overline{\overline{G}}' = (\overline{\overline{G}} \backslash \mathbf{Del}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})) \cup \mathbf{Add}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})$

In order to apply (1), we would like to show that

(9) $\overline{G}, \underline{\mathfrak{A}}, \underline{\boldsymbol{\mu}} \vDash \boldsymbol{\Phi} \ \wedge\ \rho \ \wedge\ \forall^\circ \mathbf{src} \exists^\circ \mathbf{dst} : \boldsymbol{\sigma} \Rightarrow \ (\boldsymbol{\delta} \ \wedge\ (\neg \mathbf{Foll}_{\mathcal{T}} \Rightarrow (\forall^\circ \mathbf{dst} : \neg \mathbf{Foll}_{\mathcal{T}})))$

holds. Now by (2), we have $\overline{G}, \underline{\mathfrak{A}} \vDash \boldsymbol{\Phi}$ and $\overline{G}, \underline{\mathfrak{A}} \vDash [\boldsymbol{\sigma} \xrightarrow{a} \boldsymbol{\delta}]$. Thus it is sufficient to find for each v such that $\overline{G}, \underline{\mathfrak{A}}, \mathbf{src} \mapsto v \vDash \boldsymbol{\sigma}$ some w satisfying

(10)$\overline{G}, \underline{\mathfrak{A}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \boldsymbol{\delta} \ \wedge\ (\neg \mathbf{Foll}_{\mathcal{T}} \Rightarrow (\forall^\circ \mathbf{dst} : \neg \mathbf{Foll}_{\mathcal{T}}))$

The w we choose is the one such that $(v, \mathsf{a}, w) \in \overline{\overline{G}}$. This w exists by virtue of (2) and the definition of edge constraint satisfaction. Moreover, $\overline{G}, \underline{\mathfrak{A}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \boldsymbol{\delta}$. Thus in order to establish (10), it suffices to suppose that

(11)$\overline{G}, \underline{\mathfrak{A}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \neg \mathbf{Foll}_{\mathcal{T}}$

and to prove that no u exists such that

(12)$\overline{G}, \underline{\mathfrak{A}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto u \vDash \mathbf{Foll}_{\mathcal{T}}$.

For a contradiction, assume that some u does satisfy (12). Then $(v, \mathsf{a}, u) \in \mathbf{Foll}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}})$. But by (5), $\mathbf{Foll}_{\mathcal{T}}(\overline{G}, \underline{\boldsymbol{\mu}}) \subseteq \overline{\overline{G}}$, and thus $u = w$, which contradicts our supposition (11). It follows that (9) holds, and by (1) we then obtain a $\underline{\mathfrak{B}}$ such that

$$\begin{aligned} &\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \underline{\boldsymbol{\mu}} \vDash \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\Phi}' \\ &\qquad \wedge \forall^\circ \mathbf{src} : \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\sigma}' \ \Rightarrow \\ &\qquad\qquad ((\exists^\circ \mathbf{dst} : \mathbf{Add}_{\mathcal{T}} \ \wedge\ \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}') \\ &\qquad\qquad \vee(\exists^\circ \mathbf{dst} : \mathbf{Foll}_{\mathcal{T}} \ \wedge\ \neg \mathbf{Del}_{\mathcal{T}} \ \wedge\ \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}') \\ &\qquad\qquad \vee(\boldsymbol{\sigma} \ \wedge\ \forall^\circ \mathbf{dst} : \neg \mathbf{Add}_{\mathcal{T}} \wedge \neg \mathbf{Foll}_{\mathcal{T}} \wedge\ (\boldsymbol{\delta} \Rightarrow \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}'))) \end{aligned} \tag{13}$$

holds. From (13) and Lemma 9 (Faithfulness), it follows that

(14)$\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}} \vDash \boldsymbol{\Phi}'$

We thus only need to show that also the edge constraint $[\boldsymbol{\sigma}' \xrightarrow{\mathsf{a}} \boldsymbol{\delta}']$ holds. To do this, we consider $v \in \overline{G}'$ such that

(15)$\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \mathbf{src} \mapsto v \vDash \boldsymbol{\sigma}'$.

We must then prove that there is w such that $(v, \mathsf{a}, w) \in \overline{\overline{G}}'$ and

(16)$\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \boldsymbol{\delta}'$.

Now by (15) and Lemma 9 (Faithfulness), we have

(17)$\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \underline{\boldsymbol{\mu}}, \mathbf{src} \mapsto v \vDash \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\sigma}'$.

Discharging the hypothesis in (13) by means of (17) gives us three cases:

(18) $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \underline{\boldsymbol{\mu}}, \mathbf{src} \mapsto v \vDash \exists^\circ \mathbf{dst} : \mathbf{Add}_{\mathcal{T}} \ \wedge\ \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}'$
(19) $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \underline{\boldsymbol{\mu}}, \mathbf{src} \mapsto v \vDash \exists^\circ \mathbf{dst} : \mathbf{Foll}_{\mathcal{T}} \ \wedge\ \neg \mathbf{Del}_{\mathcal{T}} \ \wedge\ \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}'$
(20) $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \underline{\boldsymbol{\mu}}, \mathbf{src} \mapsto v \vDash \boldsymbol{\sigma} \ \wedge\ \forall^\circ \mathbf{dst} : \neg \mathbf{Add}_{\mathcal{T}} \wedge \neg \mathbf{Foll}_{\mathcal{T}} \wedge\ (\boldsymbol{\delta} \Rightarrow \mathbf{Tr}_{\mathcal{T}} \boldsymbol{\delta}'))$

In case (18) there is a w such that

$$(21)\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \underline{\boldsymbol{\mu}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \mathbf{Add}_{\mathcal{T}} \ \wedge \ \mathbf{Tr}_{\mathcal{T}}\boldsymbol{\delta}'$$

By (8), $(v, \mathsf{a}, w) \in \overline{\overline{G}}'$, and by Lemma 9 (Faithfulness) (16) holds. Case (19) is handled by a similar argument. Finally, in Case (20) we have by Lemma 9 (Faithfulness) that $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \mathbf{src} \mapsto v \vDash \boldsymbol{\sigma}$ and $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \neg\mathbf{Add}_{\mathcal{T}} \wedge \neg\mathbf{Foll}_{\mathcal{T}} \wedge (\boldsymbol{\delta} \Rightarrow \mathbf{Tr}_{\mathcal{T}}\boldsymbol{\delta}')$, where w is the node such that $(v, \mathsf{a}, w) \in \overline{\overline{G}}$ (this node exists by virtue of (2)). By (8), (20), and Lemma 10, we infer that $(v, \mathsf{a}, w) \in \overline{G}'$ and by (2) that $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \mathbf{Tr}_{\mathcal{T}}\boldsymbol{\delta}$. Thus $\overline{G}, \underline{\mathfrak{A}}, \underline{\mathfrak{B}}, \mathbf{src} \mapsto v, \mathbf{dst} \mapsto w \vDash \mathbf{Tr}_{\mathcal{T}}\boldsymbol{\delta}'$ holds, whence (16) holds by Lemma 9 (Faithfulness). $\square$

Theorem 12. *(Completeness)* $\Vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$ *implies* $\vdash \mathcal{A}\{\mathcal{T}\}\mathcal{B}$.

Proof Proof can be found in full paper.

$\square$

Theorem 13. *Transductional correctness is decidable for* **EC**.

Proof By Theorems 2, 11, and 12. $\square$

References

1. B. Courcelle. Graph rewriting: an algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 193–242. Elsevier Science Publishers, 1990.
2. B. Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Information and computation*, 85:12–75, 1990.
3. B. Courcelle. Monadic second-order definable graph transductions. In J.C. Raoult, editor, *CAAP '92, Colloquium on Trees in Algebra and Programming, LNCS 581*, pages 124–144. Springer Verlag, 1992.
4. J. Engelfriet. A characterizarion of context-free NCE graph languages by monadic second-order logic on trees. In H. Ehrig, H.J. Kreowski, and G. Rozenberg, editors, *Graph grammars and their applications to computer science, 4th International Workshop, LNCS 532*, pages 311–327. Springer Verlag, 1990.
5. N. Klarlund and M. Schwartzbach. Graph types. In *Proc. 20th Symp. on Princ. of Prog. Lang.*, pages 196–205. ACM, 1993.
6. N. Klarlund and M. Schwartzbach. Invariants as data types. Unpublished, 1993.
7. M. Rabin. A simple method for undecidability proofs and some applications. In *Logic, Methodology and Philosophy of Science II*, pages 58–68. North-Holland, 1965.