Monadic Second-Order Incorrectness Logic for GP 2

Christopher M. Poskitt[a,*], Detlef Plump[b]

[a]School of Computing and Information Systems, Singapore Management University, Singapore
[b]Department of Computer Science, University of York, York, UK

Abstract

Program logics typically reason about an over-approximation of program behaviour to prove the absence of bugs. Recently, program logics have been proposed that instead prove the *presence* of bugs by means of *under-approximate reasoning*, which has the promise of better scalability. In this paper, we present an under-approximate program logic for GP 2, a rule-based programming language for manipulating graphs. We define the proof rules of this program logic extensionally, i.e. independently of fixed assertion languages, then instantiate them with a morphism-based assertion language able to specify monadic second-order properties on graphs (e.g. path properties). We show how these proof rules can be used to reason deductively about the presence of forbidden graph structure or failing executions. Finally, we prove our 'incorrectness logic' to be sound, and our extensional proof rules to be relatively complete.

Keywords: program logics, correctness proofs, under-approximate reasoning, monadic second-order logic, graph transformation

1. Introduction

Many problems in computer science and software engineering can be modelled in terms of rule-based graph transformation [1], motivating research into verifying the correctness of grammars and programs based on this form of computation. Various approaches towards this goal have been proposed, including techniques based on model checking [2], interactive proof assistants [3], unfold-

[*] Corresponding author

Email addresses: cposkitt@smu.edu.sg (Christopher M. Poskitt), detlef.plump@york.ac.uk (Detlef Plump)

ings [4, 5], *k*-induction [6], weakest preconditions [7, 8], strongest postconditions [9], abstract interpretation [10], and program logics [11, 12, 13].

Verification approaches based on program logics and proofs typically reason about over-approximations of program behaviours to prove the absence of bugs. For instance, proving a partial correctness specification $\{pre\}P\{post\}$ guarantees that for states satisfying *pre*, every successfully terminating execution of *P* ends in a state satisfying *post*. Recently, authors have begun to investigate *under-approximate* program logics that instead prove the *presence* of bugs, motivated by the promise of better scalability that may result from reasoning only about the subset of execution paths that are relevant. De Vries and Koutavas [14] proposed the first program logic of this kind, using it to reason about state reachability for randomised nondeterministic algorithms. O'Hearn [15] extended the idea to an *incorrectness logic* that tracked both successful and erroneous executions. Under-approximate program logics have also been explored for local reasoning [16], concurrency [17], and proving insecurity [18].

An under-approximate specification $[pres]P[res]$ specifies a reachability property in the reverse direction: that every state satisfying *res* ('result') is reachable by executing *P* on *some* state (not necessarily all) satisfying *pres* ('presumption'). In other words, *res* under-approximates the reachable states, allowing for sound reasoning about undesirable behaviours without any false positives, i.e. a formal logical basis for bug catching. This is one of many dualities under-approximate program logics have with Hoare logics [19]. Other important dualities include the inverted rule of consequence in which postconditions can be strenghtened (e.g. by dropping disjuncts/paths), as well as the completeness proof which relies on *weakest postconditions* rather than weakest preconditions.

In this paper, we present an under-approximate program logic for reasoning about the presence of bugs in GP 2 [20], a nondeterministic rule-based programming language for manipulating graphs. Following O'Hearn [15], we design it as an *incorrectness logic*, and show how it can be used to reason deductively about the presence of forbidden non-local graph structures or failing executions (e.g. due to the failure of finding a match for a rule). As our main technical result, we prove the soundness and relative completeness of our incorrectness logic with respect to GP 2's formal operational semantics. The work in this paper is principally a theoretical exposition, but is motivated by some possible future applications, such as the use of incorrectness logic as a basis for sound reasoning in symbolic execution tools for graph and model transformations (e.g. [21, 22, 23]).

This is a revised and extended version of the ICGT'21 paper, *"Incorrectness Logic for Graph Programs"* [24], and contains the following new content:

- We target GP 2 [20], a full-fledged graph programming language, as opposed to the rudimentary language explored in [24].

- Our under-approximate program logic is presented in an extensional style, i.e. independent of any fixed assertion language.

- We instantiate the extensional program logic using monadic second-order conditions with expressions, i.e. a morphism-based assertion language combining the ability to reason about non-local structural graph properties (as in [25]) with properties over attributes (as in [12, 13]).

- Fully revised examples and explanations throughout.

Organisation. The paper is organised as follows. In Section 2 we introduce the syntax and semantics of the graph programming language GP 2. In Section 3, we present an extensional under-approximate program logic for GP 2, proving its soundness and relative completeness. In Section 4, we define an assertion language ('ME-conditions') for specifying monadic second-order properties of GP 2 graphs, and 'plug-in' the language to our extensional program logic. Section 5 explores the utility of under-approximate reasoning in the context of some programs for recognising cycle graphs. In Section 6 we survey some related work, before concluding in Section 7.

2. The Graph Programming Language GP 2

This section introduces GP 2, a nondeterministic rule-based programming language for manipulating graphs. The language is implemented by a compiler that generates C code [26, 27], and is available to download online[1].

2.1. Underlying Theory: Graphs and Rules

We begin by introducing *graphs* and *rules*, which are respectively the program states and units of computation in GP 2.

GP 2 uses a definition of graphs in which edges are directed, nodes (resp. edges) are partially (resp. totally) labelled, and both parallel edges and loops are allowed to exist. All graphs in this paper will be totally labelled except for the interface graphs in rule applications, where they are needed for technical reasons to support relabelling.

[1] `https://github.com/UoYCS-plasma/GP2`

We remark that nodes in GP 2 can be distinguished as *rooted*, which allows for faster rule-matching in the implementation by localising the search for a match to the neighbourhoods of rooted graph nodes [28] (i.e. instead of searching the entire graph). The underlying theory utilised in this paper supports rooted graph transformation (see Appendix A and [29]). For practical purposes, in this paper, one can think of 'rootedness' as a special kind of binary label (a node is either rooted or not).

Definition 2.1 (Graph). Let C be a set of labels. A *graph* over C is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G, p_G \rangle$ comprising a finite set V_G of *nodes*, a finite set E_G of *edges*, *source* and *target functions* $s_G, t_G : E_G \to V_G$, a partial *node labelling function* $l_G : V_G \to C$, an *edge labelling function* $m_G : E_G \to C$, and a function $p_G \colon V_G \to \mathbb{B}$ determining *node rootedness*, where $\mathbb{B} = \{\text{true}, \text{false}\}$. □

If $V_G = \emptyset$, then G is the *empty graph*, which we denote by $\emptyset$. Given a node $v \in V_G$, we write $l_G(v) = \bot$ to express that $l_G(v)$ is undefined. A graph G is *totally labelled* if l_G is a total function.

We write $\mathcal{G}(C_\bot)$ (resp. $\mathcal{G}(C)$) to denote the *class* of all (resp. all *totally labelled*) graphs over label alphabet C.

GP 2 programs operate on graphs labelled over two components: a *list* of integers and/or character strings, and an optional *mark* (or colour) for convenience when implementing graph algorithms. Graphs labelled over this alphabet are known as *host graphs*.

Definition 2.2 (Label alphabet $\mathcal{L}$). We denote by $\mathcal{L}$ the *label alphabet for host graphs*:

$$\mathcal{L} = \mathbb{L} \cup (\mathbb{L} \times \mathbb{M})$$

where $\mathbb{L} = (\mathbb{Z} \cup \text{Char}^*)^*$ and $\mathbb{M} = \{\texttt{red}, \texttt{green}, \texttt{blue}, \texttt{grey}, \texttt{dashed}\}$. □

Example 2.3 (Graph). Figure 1 depicts a graph G from $\mathcal{G}(\mathcal{L})$. It consists of four nodes ($v1$ through to $v4$) and three edges ($e1$ through to $e3$), with sources $s_G(e1) = v1$, $s_G(e2) = v2$, $s_G(e3) = v4$, and targets $t_G(e1) = v2$, $t_G(e2) = v4$, and $t_G(e3) = v3$. The list component of each node/edge consists either of an atomic value (e.g. `5`) or a list of values (e.g. `8:8`). All nodes/edges are unmarked, except for $v3$ which has the mark `green`, and $e3$ which has the mark `dashed`. Node $v2$ is *rooted*, which is depicted via the double border.

We remark that node/edge identifiers will often be omitted from depictions of graphs unless they are specifically required. □

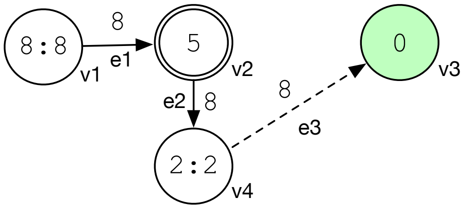


Figure 1: Example graph from $\mathcal{G}(\mathcal{L})$

Each graph G induces a predicate $\mathrm{path}_G(v, w, E)$ which holds if there is a directed path in the graph from node v to w avoiding any edges in E. This predicate will be used later (Section 4) to give a semantics to the (syntactic) path predicates of our assertion language.

Definition 2.4 (Semantic path predicate). Given a graph G, the *(semantic) path predicate)* $\mathrm{path}_G \colon V_G \times V_G \times 2^{E_G} \to \mathbb{B}$ is defined inductively for nodes $v, w \in V_G$ and sets of edges $E \subseteq E_G$. If $v = w$, then $\mathrm{path}_G(v, w, E)$ holds. If $v \neq w$, then $\mathrm{path}_G(v, w, E)$ holds if there exists an edge $e \in E_G \setminus E$ such that $s_G(e) = v$ and $\mathrm{path}_G(t_G(e), w, E)$. □

Consider graph G in Figure 1. The predicate $\mathrm{path}_G(v1, v3, \emptyset)$ holds, for example, whereas $\mathrm{path}_G(v1, v3, \{e3\})$ and $\mathrm{path}_G(v3, v1, \emptyset)$ do not.

The formal semantics of graph-matching and rule application (Appendix A) requires us to be able to relate two graphs in a formal way. For this purpose, we use graph morphisms, which are structure- and label-preserving mappings from the nodes and edges of one graph to another. (We remark that rootedness and non-rootedness are preserved in morphisms, too.)

Definition 2.5 (Graph morphism). A *graph morphism* $g \colon G \to H$ between graphs G, H in $\mathcal{G}(C_\perp)$ consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources, targets, labels, and rootedness; that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, $p_H \circ g_V = p_G$, and $l_H(g_V(v)) = l_G(v)$ for all nodes v for which $l_G(v) \neq \perp$. We call G, H respectively the *domain* and *codomain* of g, denoted by $\mathrm{dom}(g)$ and $\mathrm{codom}(g)$. □

A morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective). Injective morphisms are usually denoted by hooked arrows, $\hookrightarrow$. A morphism g is an *isomorphism* if it is injective, surjective, and satisfies $l_H(g_V(v)) = \perp$ for all nodes v with $l_G(v) = \perp$. In this case G and H are *isomorphic*, which is denoted by $G \cong H$. Finally, a morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x, and is *undefinedness-preserving* if $l_H(g_V(v)) = \bot$ for all nodes v such that $l_G(v) = \bot$.

Next, we introduce *rules*, which are the underlying unit of computation in the execution of graph programs. Rules consist of three graphs: a *left-hand* graph L, indicating the sub-graph to be matched; an *interface* graph K, indicating the elements of the match that are preserved (i.e. not deleted); and a *right-hand* graph R, indicating the labels to be manipulated and the structure to be created.

Definition 2.6 (Rule). A *(concrete) rule* $r : \langle L \hookleftarrow K \hookrightarrow R\rangle$ comprises totally labelled graphs $L, R \in \mathcal{G}(\mathcal{L})$, a partially labelled graph $K \in \mathcal{G}(\mathcal{L}_\bot)$, and inclusions $K \hookrightarrow L$, $K \hookrightarrow R$. We call L, R the *left-* and *right-hand graphs of* r, and K its *interface*. □

Intuitively, an application of a rule r to a graph $G \in \mathcal{G}(\mathcal{L})$ removes items in $L - K$, preserves those in K, adds the items in $R - K$, and relabels the unlabelled nodes in K. An injective morphism $g : L \hookrightarrow G$ is a *match* for r if it satisfies the *dangling condition*, i.e. no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case, G directly derives $H \in \mathcal{G}(\mathcal{L})$ with *comatch* $h : R \hookrightarrow H$, denoted $G \Rightarrow_{r,g,h} H$ (or just $G \Rightarrow_r H$), by: (1): removing all nodes and edges in $g(L)-g(K)$; (2) disjointly adding all nodes and edges from $R - K$, keeping their labels (for $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$; targets analogous); (3) for every node in K, $l_H(g_V(v))$ becomes $l_R(v)$. Semantically, direct derivations are constructed as two 'natural pushouts' (see Appendix A and [30] for the technical details).

2.2. Assignments and Rule Schemata

Specifying concrete rules directly over graphs in $\mathcal{G}(\mathcal{L})$ is insufficient for graph programs in practice. Suppose, for example, that one wanted to specify a rule that created a loop incident to a node labelled with any integer: this would require an infinite number of rules over each value in $\mathbb{Z}$.

To address this, GP 2 instead requires programmers to specify *(conditional) rule schemata*, which are essentially rules but labelled over expressions instead. Each rule schema represents (potentially) infinitely many concrete rules, depending on how the variables are assigned and expressions evaluated. The graphs of rule schemata are known as *rule graphs*, and the expressions they are labelled over are derived according to a grammar. We begin by defining this grammar, and show how rule graphs can be instantiated into host graphs via assignments and expression evaluation.

Label ::= List [Mark]
List ::= LVar | `empty` | Atom | List ‘:’ List
Atom ::= AVar | Integer | String
Integer ::= IVar | [‘-’] Digit {Digit} | ‘(’Integer‘)’ |
Integer (‘+’ | ‘-’ | ‘$*$’ | ‘/’) Integer |
(`indeg` | `outdeg`) ‘(’Node‘)’ |
`length` ‘(’(LVar | AVar | SVar)‘)’
String ::= SVar | Char | ‘“’{Character}‘”’ |
String ‘.’ String
Char ::= CVar | ‘“’Character‘”’
Mark ::= `red` | `green` | `blue` | `grey` | `dashed` | `any`

Figure 2: Abstract syntax of rule graph labels

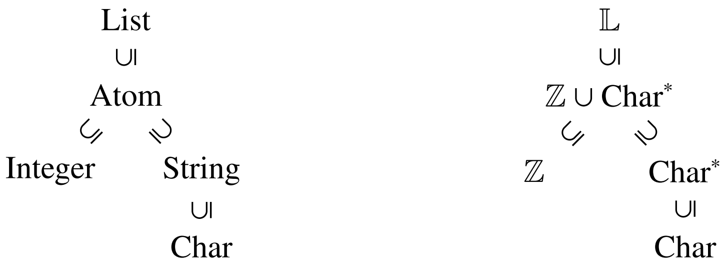


Figure 3: GP 2 subtype hierarchy

Definition 2.7 (Label alphabet RG). We denote by RG the *label alphabet for rule graphs* containing all expressions that can be derived from Label in the grammar of Figure 2. □

List expressions represent (possibly infinite) sets of lists in $\mathbb{L}$, which are obtained by assigning values to (typed) variables and evaluating expressions. We identify lists of length one with the atomic expressions they contain, allowing us to organise the list components of RG and $\mathcal{L}$ into subtype hierarchies, as shown in Figure 3. (We use the non-terminals of the grammar to denote the classes of expressions that can be derived from them.)

A graph in $\mathcal{G}(\mathcal{L})$ can be obtained from a graph in $\mathcal{G}(\text{RG})$ by means of an *assignment* (for evaluating variables in list expressions) and a morphism (for evaluating in- and outdegree expressions).

Definition 2.8 (List assignment). A *list assignment* is a family of mappings $\alpha = (\alpha_X)_{X \in \{\mathtt{L},\mathtt{A},\mathtt{I},\mathtt{S},\mathtt{C}\}}$ where $\alpha_{\mathtt{L}} : \mathrm{LVar} \to \mathbb{L}$, $\alpha_{\mathtt{A}} : \mathrm{AVar} \to \mathbb{Z} \cup \mathrm{Char}^*$, $\alpha_{\mathtt{L}} : \mathrm{IVar} \to \mathbb{Z}$, $\alpha_{\mathtt{S}} : \mathrm{SVar} \to \mathrm{Char}^*$, and $\alpha_{\mathtt{C}} : \mathrm{CVar} \to \mathrm{Char}$. For simplicity, we will omit subscripts as exactly one mapping is applicable to a given variable. □

The evaluation of a list expression l with respect to a morphism g and assignment α, denoted $l^{g,\alpha}$, is obtained by substituting $g(v)$ for node identifiers v, substituting $\alpha(\mathtt{x})$ for variables $\mathtt{x}$, and evaluating expressions in the standard way. If an integer expression i is of the form $\mathtt{indeg(}v\mathtt{)}$ (resp. $\mathtt{outdeg}$) for node identifier v, then $i^{g,\alpha}$ evaluates to the indegree (resp. outdegree) of $g(v)$, i.e. the number of edges e with v as the target (resp. v as the source). If an integer expression i is of the form $\mathtt{length(x)}$ for some variable $\mathtt{x}$, then $i^{g,\alpha}$ evaluates to the length of the list or string $\alpha(\mathtt{x})$ (note that atoms are lists of length one).

We denote the domain of α by $\mathrm{dom}(\alpha)$, and the set of variables used in a graph $G \in \mathcal{G}(\mathrm{RG})$ by $\mathrm{vars}(G)$. If $\mathrm{vars}(G) \subseteq \mathrm{dom}(\alpha)$, and g is a morphism defined for all node identifiers in G, then $G^{g,\alpha} \in \mathcal{G}(\mathcal{L})$ is the graph obtained by substituting $l^{g,\alpha}$ for every list expression l in G, and by replacing each '$\mathtt{any}$' mark with the corresponding mark in the codomain of g. Morphisms can be 'instantiated' in the same way, e.g. $p\colon P \hookrightarrow C$ can be instantiated to $p^{g,\alpha}\colon P^{g,\alpha} \hookrightarrow C^{g,\alpha}$.

At this point, we have defined the rule graphs of $\mathcal{G}(\mathrm{RG})$ and shown how they are instantiated into the host graphs of $\mathcal{G}(\mathcal{L})$. Before we define rule schemata and their application, we have two more concepts to introduce: *rule schemata conditions*, and *simple expressions*. Rule schemata conditions allow programmers to constrain the possible assignments of values to variables. For example, one may wish to be able to match an integer-labelled node, but constrain the matches to only those above a certain value. Simple expressions are a restricted form of RG that helps to reduce ambiguity in the assignment and rule-matching process.

Definition 2.9 (Rule schema condition). A *rule schema condition* Γ is an expression that can be derived from RSCon in the grammar of Figure 4. □

Given a rule schema condition Γ, an assignment α, and a morphism g defined for the node identifiers in Γ, the value of $\Gamma^{g,\alpha}$ in $\mathbb{B}$ is defined inductively. If Γ has the form $l_1 \bowtie l_2$ with l_1, l_2 in List and $\bowtie$ the symbol of a relational operator, then $\Gamma^{g,\alpha}$ has the value $l_1^{g,\alpha} \bowtie_{\mathbb{L}} l_2^{g,\alpha}$, where $\bowtie_{\mathbb{L}}$ is the relational operator corresponding to $\bowtie$. If Γ has the form $\mathit{type}\mathtt{(}l\mathtt{)}$ with type in Type and l in List, then $\Gamma^{g,\alpha}$ is true if $\mathit{type} = \mathtt{int}$ (resp. $\mathtt{char}$, $\mathtt{string}$, $\mathtt{atom}$) and $l^{g,\alpha} \in \mathbb{Z}$ (resp. Char, Char^*, $\mathbb{Z} \cup \mathrm{Char}^*$). If Γ has the form $\mathtt{edge(}v_1\mathtt{,}v_2\mathtt{)}$ with v_1, v_2 node identifiers

```
RSCon   ::=  Type ‘(’(LVar | AVar | SVar)‘)’
             | List (‘=’ | ‘!=’) List
             | Integer (‘>’ | ‘>=’ | ‘<’ | ‘<=’) Integer
             | edge ‘(’ Node ‘,’ Node [‘,’ Label] ‘)’
             | not RSCon
             | RSCon (and | or) RSCon
             | ‘(’ RSCon ‘)’
Type    ::=  int | char | string | atom
```

Figure 4: Abstract syntax of rule schema conditions

(resp. `edge`(v_1, v_2, l) with label l), then $\Gamma^{g,\alpha}$ is true if there is an edge in codom(g) with source $g(v_1)$ and target $g(v_2)$ (resp. and label $l^{g,\alpha}$). If Γ has the form `not` Γ_1 with Γ_1 a rule schema condition, then $\Gamma^{g,\alpha}$ is true if $\Gamma_1^{g,\alpha}$ is false. If Γ has the form Γ_1 `and` Γ_2 (resp. Γ_1 `or` Γ_2), then $\Gamma^{g,\alpha}$ is true if $\Gamma_1^{g,\alpha}$ and (resp. or) $\Gamma_2^{g,\alpha}$ are true. Finally, if Γ has the form `(`Γ_1`)`, then $\Gamma^{g,\alpha}$ has the value $\Gamma_1^{g,\alpha}$.

The values of variables at execution time are determined by graph matching, hence we require that expressions in the left graph of a rule schema have a simple shape.

Definition 2.10 (Simple expression). A list expression l is simple if: (1) l contains no arithmetic, degree, or length operators (with the possible exception of a unary minus preceding a sequence of digits); (2) l contains at most one occurrence of a list variable; and (3) each occurrence of a string expression in l contains at most one occurrence of a string variable. □

Finally, we introduce the conditional rule schemata of GP 2, which consist of left-hand, interface, and right-hand graphs over $\mathcal{G}$(RG), along with a rule schema condition Γ.

Definition 2.11 (Rule schema). A *(conditional) rule schema* $r\colon \langle L \Rightarrow R, \Gamma\rangle$ with $L, R \in \mathcal{G}(\mathrm{RG})$ represents concrete rules $r^{g,\alpha}\colon \langle L^{g,\alpha} \hookleftarrow K \hookrightarrow R^{g,\alpha}\rangle$ where $\mathrm{dom}(\alpha) = \mathrm{vars}(L)$ and K consists of the preserved nodes only (with all nodes unlabelled). We require that for any rule schema, $\mathrm{vars}(R) \subseteq \mathrm{vars}(L)$, $\mathrm{vars}(\Gamma) \subseteq \mathrm{vars}(L)$, and all list expressions in L are simple. Note that L and R must be totally labelled graphs. □

The application of a rule schema $r = \langle L \Rightarrow R, \Gamma\rangle$ to a host graph $G \in \mathcal{G}(\mathcal{L})$ consists of the following steps: (1) choose an assignment α with $\mathrm{dom}(\alpha) = \mathrm{vars}(L)$;

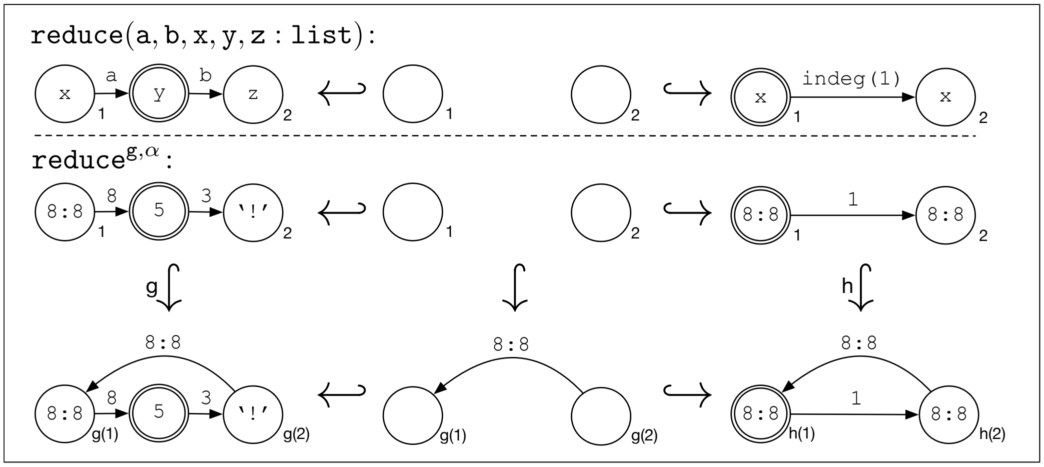


Figure 5: Example rule schema application

(2) choose a *match*, i.e. a morphism $g\colon L^{g,\alpha} \hookrightarrow G$ that satisfies the dangling condition with respect to $r^{g,\alpha}\colon \langle L^{g,\alpha} \hookleftarrow K \hookrightarrow R^{g,\alpha}\rangle$ and for which $\Gamma^{g,\alpha} = \text{true}$; (3) apply $r^{g,\alpha}$ with match g. If a graph H with comatch $h\colon R^{g,\alpha} \hookrightarrow H$ is derived from G via these steps, we write $G \Rightarrow_{r,g,h}$ (or just $G \Rightarrow_r H$). Moreover, if a graph H can be derived from a graph G via some r in a set of rule schemata $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ (i.e. nondeterministic choice of rule schema). If no rule schema in the set has a match for G, we write $G \not\Rightarrow_{\mathcal{R}}$, i.e. that $\mathcal{R}$ *fails* on G.

Example 2.12 (Rule schema application). The left-hand side, interface, and right-hand side of a rule schemata `reduce` are depicted in the top row of Figure 5. Intuitively, the rule schema matches three adjacent nodes such that: (1) the labels consist of any list components; (2) the nodes/edges are unmarked; (3) the middle node is rooted, and not incident to any edges other than those in the match. The rule schema deletes the middle node, then adds an edge between the others labelled with the indegree of node 1 in the match. Finally, node 1 is set as a rooted node, and node 2 is relabelled with the value of `x`.

The second row depicts a concrete rule instantiated from `reduce` with respect to a match g and assignment α. Here, $\texttt{indeg(1)}^{g,\alpha} = \texttt{1}$, $\alpha(\texttt{a}) = \texttt{8}$, $\alpha(\texttt{b}) = \texttt{3}$, $\alpha(\texttt{x}) = \texttt{8:8}$, $\alpha(\texttt{y}) = \texttt{5}$, and $\alpha(\texttt{z}) = $ '!'.

The bottom row shows the application of $\texttt{reduce}^{g,\alpha}$ to a graph in the double-pushout approach with relabelling [30]. □

```
Program          ::=  Declaration { Declaration }
Declaration      ::=  MainDecl | ProcedureDecl | RuleDecl
MainDecl         ::=  Main '=' CommandSeq
ProcedureDecl    ::=  ProcedureID '=' [ '[' LocalDecl ']' ] CommandSeq
LocalDecl        ::=  ( RuleDecl | ProcedureDecl ) { LocalDecl }
CommandSeq       ::=  Command {';'Command}
Command          ::=  Block
                      | if Block then Block [ else Block ]
                      | try Block [ then Block ] [ else Block ]
Block            ::=  '(' CommandSeq ')' ['!']
                      | SimpleCommand
                      | Block or Block
SimpleCommand    ::=  RuleSetCall ['!']
                      | ProcedureCall ['!']
                      | break
                      | skip
                      | fail
RuleSetCall      ::=  RuleID | '{' [ RuleID { ',' RuleID } ] '}'
ProcedureCall    ::=  ProcedureID
```

Figure 6: GP 2 Program Syntax

2.3. Programs

GP 2 provides a number of control constructs that allow for rule schemata to be applied to inputs (host graphs) programmatically. Figure 6 provides an abstract syntax of these graph programs. We shall explain the main constructs informally, before presenting a structural operational semantics that we will use as the basis for our soundness proofs.

Intuitively, a RuleSetCall, typically denoted by $\mathcal{R}$, represents a single nondeterministic application of one of the conditional rule schemata in $\mathcal{R}$. This results in failure if none of the rules are applicable to the current host graph, i.e. no matches are possible. The program $P; Q$ denotes sequential composition. The program `if` C `then` P `else` Q denotes conditional branching: if the execution of C on an input results in a graph, then P is executed on that *original input graph*; otherwise, if C fails, then Q is executed instead. (The `try` construct operates similarly, except that the effects of C are retained.) The program $P!$ denotes as-long-as-

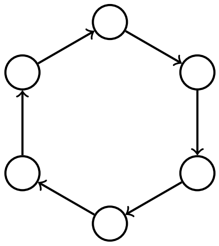
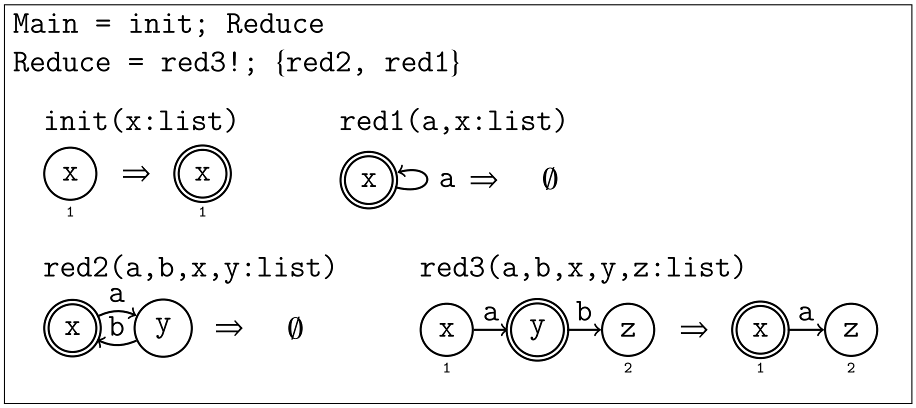


Figure 7: A cycle graph



```
Main = init; Reduce
Reduce = red3!; {red2, red1}
```

Figure 8: GP 2 program `is-cycle-buggy`

possible iteration of P, in which the iteration terminates the moment that a step of P is not applicable to the graph (note that the overall iteration $P!$ can never fail). The command `break` is used to exit such an iteration. Finally, `skip` is a null-command (i.e. the rule schema $\langle \emptyset \Rightarrow \emptyset, \texttt{true} \rangle$), and `fail` results in failure.

Example 2.13 (Graph program). A *cycle graph* is an unmarked and unrooted host graph consisting of n nodes and n edges, $n \geq 1$, which form a directed cycle. See Figure 7 for an example. The program `is-cycle-buggy` in Figure 8 is meant to recognise whether an input graph is a cycle graph or not, by reducing cycle graphs to the empty graph. (The program has a bug affecting a specific type of input that we will discuss in Section 5.)

The program runs as follows. First, `init` nondeterministically transforms a single non-rooted node into a rooted node. (Intuitively, this makes the subsequent rule applications more efficient as the rule-matching can be done in the local neighbourhood of the rooted node.) Second, the reduction rule `red3` is applied for as long as possible. The key to this rule is the dangling condition: the rooted node in the middle can only be removed if there are no other edges incident to it. Finally, the program attempts to apply exactly one more reduction: either `red2`, which reduces a cycle graph consisting of two nodes, or `red1`, which deals with the special case of a single-node cycle graph (zero iterations of `red3` would have been applied to such an input).

Executing this program on the graph of Figure 7 results in the empty graph. □

The semantics of GP 2 program constructs are given in the style of a structural operational semantics [31]. In this approach, we inductively define small-step transitions $\rightarrow$ on *configurations*, which are either command sequences together with a host graph, just a host graph, or the special element fail:

$$\rightarrow \subseteq (\text{CommandSeq} \times \mathcal{G}(\mathcal{L})) \times ((\text{CommandSeq} \times \mathcal{G}(\mathcal{L})) \cup \mathcal{G}(\mathcal{L}) \cup \text{fail})$$

Configurations in $\text{CommandSeq}\times\mathcal{G}(\mathcal{L})$ represent unfinished computations, whereas graphs in $\mathcal{G}(\mathcal{L})$ and the special element fail are terminal states that represent finished computations.

Figure 9 shows the inference rules for the core commands of GP 2. The inference rules contain meta-variables for command sequences and graphs, where $\mathcal{R}$ stands for a call of a rule schema set, C, P, P', Q stand for command sequences, and G, H for host graphs in $\mathcal{G}(\mathcal{L})$. The transitive and reflexive-transitive closures of $\rightarrow$ are written $\rightarrow^+$ and $\rightarrow^*$ respectively.

The inference rules for the remaining GP 2 commands are given in Figure 10. We call these derived commands because there are equivalent combinations of core commands that can define them.

The meaning of GP 2 programs are summarised by binary semantic relations. We associate each program P with two semantic relations: $[\![P]\!]_{\text{ok}}$ and $[\![P]\!]_{\text{fa}}$, where are *ok* and *fa* are *exit statuses*. A pair of host graphs (G, H) is in $[\![P]\!]_{\text{ok}}$ if an execution of P on G can result in H. A pair of host graphs (G, H) is in $[\![P]\!]_{\text{fa}}$, however, if executing P on G can result in failure (with H the last graph derived before failure occurred). Note that if a program P fails immediately on G, then $(G, G) \in [\![P]\!]_{\text{fa}}$.

Definition 2.14 (Semantics). The *semantics* of a graph program P is given by binary relations $[\![P]\!]_{\epsilon} \subseteq \mathcal{G}(\mathcal{L}) \times \mathcal{G}(\mathcal{L})$, where $(G, H) \in [\![P]\!]_{\text{ok}}$ if there is a sequence of configurations $\langle P, G\rangle \rightarrow^+ H$, and $(G, H) \in [\![P]\!]_{\text{fa}}$ if there is a sequence of configurations $\langle P, G\rangle \rightarrow^* \langle P', H\rangle \rightarrow \text{fail}$. □

$$[\mathrm{call}_1]\ \frac{G \Rightarrow_R H}{\langle R,\, G\rangle \to H} \qquad [\mathrm{call}_2]\ \frac{G \not\Rightarrow_R}{\langle R,\, G\rangle \to \mathrm{fail}}$$

$$[\mathrm{seq}_1]\ \frac{\langle P,\, G\rangle \to \langle P',\, H\rangle}{\langle P; Q,\, G\rangle \to \langle P'; Q,\, H\rangle} \qquad [\mathrm{seq}_2]\ \frac{\langle P,\, G\rangle \to H}{\langle P; Q,\, G\rangle \to \langle Q,\, H\rangle}$$

$$[\mathrm{seq}_3]\ \frac{\langle P,\, G\rangle \to \mathrm{fail}}{\langle P; Q,\, G\rangle \to \mathrm{fail}}$$

$$[\mathrm{if}_1]\ \frac{\langle C,\, G\rangle \to^{+} H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle P,\, G\rangle}$$

$$[\mathrm{if}_2]\ \frac{\langle C,\, G\rangle \to^{+} \mathrm{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle Q,\, G\rangle}$$

$$[\mathrm{try}_1]\ \frac{\langle C,\, G\rangle \to^{+} H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle P,\, H\rangle}$$

$$[\mathrm{try}_2]\ \frac{\langle C,\, G\rangle \to^{+} \mathrm{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \to \langle Q,\, G\rangle}$$

$$[\mathrm{alap}_1]\ \frac{\langle P,\, G\rangle \to^{+} H}{\langle P!,\, G\rangle \to \langle P!,\, H\rangle} \qquad [\mathrm{alap}_2]\ \frac{\langle P,\, G\rangle \to^{+} \mathrm{fail}}{\langle P!,\, G\rangle \to G}$$

$$[\mathrm{alap}_3]\ \frac{\langle P,\, G\rangle \to^{*} \langle \texttt{break}, H\rangle}{\langle P!,\, G\rangle \to H} \qquad [\mathrm{break}]\ \langle \texttt{break}; P,\, G\rangle \to \langle \texttt{break},\, G\rangle$$

Figure 9: Inference rules for core commands

$$[\mathrm{or}_1]\quad \langle P\ \texttt{or}\ Q,\, G\rangle \to \langle P,\, G\rangle \qquad [\mathrm{or}_2]\quad \langle P\ \texttt{or}\ Q,\, G\rangle \to \langle Q,\, G\rangle$$

$$[\mathrm{skip}]\quad \langle \texttt{skip},\, G\rangle \to G \qquad [\mathrm{fail}]\quad \langle \texttt{fail},\, G\rangle \to \mathrm{fail}$$

$$[\mathrm{if}_3]\quad \langle \texttt{if}\ C\ \texttt{then}\ P,\, G\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip},\, G\rangle$$

$$[\mathrm{try}_3]\quad \langle \texttt{try}\ C\ \texttt{then}\ P,\, G\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip},\, G\rangle$$

$$[\mathrm{try}_4]\quad \langle \texttt{try}\ C\ \texttt{else}\ P,\, G\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ P,\, G\rangle$$

$$[\mathrm{try}_5]\quad \langle \texttt{try}\ C,\, G\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip},\, G\rangle$$

Figure 10: Inference rules for derived commands

Note that divergence is treated in an implicit way: a program that always diverges is associated with empty relations. For example, $[\![\langle \emptyset \Rightarrow \emptyset\rangle!]\!]_{ok} = \emptyset$.

The purpose of defining two semantic relations is to allow for sound under-approximate proofs about both the executions that result in graphs and those that (at some point, but perhaps not immediately) result in failure. Semantically, graphs and fail are both considered proper outcomes of a program execution. However, if we model more of the GP 2 runtime, then we could also define semantic relations for tracking improper exit statuses (e.g. division by zero).

3. An Under-Approximate Program Logic

In this section, we present our under-approximate program logic for GP 2 in an *extensional* style, i.e. independent of any fixed assertion language. Instead of concrete assertions (e.g. in a first-order logic), we use semantic characterisations, which allows us to separate incompleteness due to the proof rules from incompleteness due to the assertion language (e.g. the assertion language of Section 4). We begin by defining what is meant by an assertion language, as well as some key semantic characterisations for our proof rules: SE, characterising the existence of a successful execution; and FE, characterising the existence of a failing execution.

Definition 3.1 (Assertion language). An *assertion language* is a pair $\mathfrak{A} = \langle A, \models_{\mathfrak{A}}$ $\rangle$, where A is a (possibly infinite) set of assertions, and $\models_{\mathfrak{A}} \subseteq \mathcal{G}(\mathcal{L}) \times A$ is a *satisfaction relation*. We say that a graph $G \in \mathcal{G}(\mathcal{L})$ *satisfies* an assertion $c \in A$, denoted $G \models_{\mathfrak{A}} c$, if $(G, c) \in \models_{\mathfrak{A}}$. □

Definition 3.2 (Extensional assertion false). Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ denote an assertion language, P a graph program, and c an assertion in A. We denote by false any assertion in A that expresses a property that cannot be satisfied by any graph, i.e. $\neg\exists G.\ G \models_{\mathfrak{A}} \text{false}$. □

Definition 3.3 (Extensional assertion SE). Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ denote an assertion language, P a graph program, and c an assertion in A. We denote by $\text{SE}[P, c]$ any assertion in A that expresses the *weakest property for a successful execution of P to exist on a graph satisfying c*, i.e. such that for any graph $G \in \mathcal{G}(\mathcal{L})$,

$$G \models_{\mathfrak{A}} \text{SE}[P, c] \text{ if and only if } (G \models_{\mathfrak{A}} c \text{ and } \exists H.\ (G, H) \in [\![P]\!]_{ok}).$$

□

Definition 3.4 (Extensional assertion FE). Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ denote an assertion language, P a graph program, and c an assertion in A. We denote by $\text{FE}[P, c]$ any assertion in A that expresses the *weakest property for a failing execution of P to exist on a graph satisfying c*, i.e. such that for any graph $G \in \mathcal{G}(\mathcal{L})$,

$$G \models_{\mathfrak{A}} \mathrm{FE}[P, c] \text{ if and only if } \Big(G \models_{\mathfrak{A}} c \text{ and } \exists H.\ (G, H) \in [\![P]\!]_{fa}\Big).$$

□

Before we define the proof rules of our incorrectness logic, it is important to define what an *incorrectness specification* is and what it means for it to be *valid*. In over-approximate program logics (e.g. [12, 13]) a specification is given in the form of a triple, $\{c\}P\{d\}$, which under partial correctness expresses that if a graph satisfies precondition c, and program P successfully terminates on it, then the resulting graph will always satisfy d. The postcondition d over-approximates the graphs reachable upon termination of P from graphs satisfying c.

Incorrectness logic [15], however, is based on under-approximate reasoning, for which a specification $[c]P[d]$ has a rather different meaning (and thus a different notation). Here, we call the pre-assertion c a *presumption* and the post-assertion d a *result*. The triple specifies that if a graph satisfies d, then it can be derived from *some* graph satisfying c by executing P on it. In other words, d under-approximates the states reached as a result of executing P on graphs satisfying c. It does not specify that every graph satisfying c derives a graph satisfying d, and it does not preclude graphs satisfying $\neg c$ from deriving such graphs either.

The principal benefit of proving such triples is then proving the *presence of graphs*, and can be thought of as providing a possible formal foundation for static bug catchers, e.g. symbolic execution tools. In graph programs, this amounts to formal proofs of the presence of *illegal graph structure*, but it can also facilitate proofs of the presence of *failure*. To accommodate this, we adopt O'Hearn's approach [15] of tracking exit statuses ϵ in the result, $[c]P[\epsilon : d]$, using *ok* to represent executions that result in a graph, and fa to track executions that fail.

This under-approximate style of reasoning occasionally requires us to specify the *weakest postcondition* relative to a presumption c and program P. In contrast to the more familiar weakest precondition, a weakest postcondition specifies the most general property that a graph H must satisfy to guarantee the existence of a pre-state G that satisfies c and can be transformed into H via P. Note that this is essentially a *reachability* property, as it guarantees the existence of at least one such execution (rather than guaranteeing properties of all of them).

Definition 3.5 (Extensional weakest postcondition). Let $\mathfrak{A} = \langle A, \models_{\mathfrak{A}} \rangle$ denote an assertion language, P a graph program, and c an assertion in A. We denote by

$$\textsc{RuleSetSucc}\ \vdash [\mathrm{SE}[\mathcal{R}, c]]\ \mathcal{R}\ [ok\colon \mathrm{WPOST}_{ok}[\mathcal{R}, c]][fa\colon \mathrm{false}]$$

$$\textsc{RuleSetFail}\ \vdash [\mathrm{FE}[\mathcal{R}, c]]\ \mathcal{R}\ [ok\colon \mathrm{false}][fa\colon \mathrm{FE}[\mathcal{R}, c]]$$

$$\textsc{Skip}\ \vdash [c]\ \texttt{skip}\ [ok\colon c][fa\colon \mathrm{false}]$$

$$\textsc{Fail}\ \vdash [c]\ \texttt{fail}\ [ok\colon \mathrm{false}][fa\colon c]$$

$$\textsc{IterZero}\ \vdash [\mathrm{FE}[P, c]]\ P!\ [ok : \mathrm{FE}[P, c]][fa : \mathrm{false}]$$

Figure 11: Extensional incorrectness axioms for graph programs

$\mathrm{WPOST}_\epsilon[P, c]$ any assertion in A that is a *weakest postcondition relative to c, P, and* ϵ, i.e. such that for any graph $H \in \mathcal{G}(\mathcal{L})$,

$$H \models_{\mathfrak{A}} \mathrm{WPOST}_\epsilon[P, c] \text{ if and only if } (\exists G.\ G \models_{\mathfrak{A}} c \text{ and } (G, H) \in \llbracket P \rrbracket_\epsilon)\,.$$

□

Definition 3.6 (Under-approximate validity). Let c, d denote assertions from an assertion language $\mathfrak{A} = \langle a, \models_{\mathfrak{A}} \rangle$, P a graph program, and ϵ an exit status. A specification $[c]\ P\ [\epsilon : d]$ is *valid*, denoted $\models [c]\ P\ [\epsilon : d]$, if for every graph $H \in \mathcal{G}(\mathcal{L})$ such that $H \models_{\mathfrak{A}} d$, there exists a graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \models_{\mathfrak{A}} c$ and $(G, H) \in \llbracket P \rrbracket \epsilon$. □

Figures 11 and 12 present the axioms and proof rules of our incorrectness logic for GP 2, which are adapted from O'Hearn's incorrectness logic for imperative programs [15]. We say that a triple is *provable*, denoted $\vdash [c]P[\epsilon : d]$, if it can be instantiated from any axiom, or deduced as the consequent of any proof rule with provable antecedents. We use the notation $\vdash [c]P[ok : d_1][fa : d_2]$ as shorthand for two separate triples, $\vdash [c]P[ok : d_1]$ and $\vdash [c]P[fa : d_2]$.

The axioms RuleSetSucc and RuleSetFail allow for reasoning about the most fundamental unit of graph programs: rule schemata set application. The former covers the successful case: if a graph satisfies the weakest postcondition relative to c and $\mathcal{R}$, then it can be derived from a graph satisfying the presumption $\mathrm{SE}[\mathcal{R}, c]$, i.e. which expresses that there is an applicable application of $\mathcal{R}$ on a graph satisfying c. The latter of the axioms covers the possibility that $\mathcal{R}$ cannot be applied: in this case, we have an exit condition of fa to track its failure.

The axiom Skip reflects that the program state (i.e. host graph) does not change and that the command cannot fail. The axiom Fail, on the other hand, reflects that the command's execution cannot result in a graph, but rather instantly transitions the execution to fail: whatever condition held in the pre-state is tracked as the last condition to be satisfied before the failure occurred. The final axiom, IterZero, covers the case when the iterated program immediately fails in the first iteration. This means that the pre-state already had the conditions for a failing execution of P, and as no iteration of P was able to complete, the post-state still retains this. (Note how $P!$ itself can never fail.)

Sequential composition is handled by the proof rules SeqSucc as well as SeqFail, with the latter covering the possibility of the first program resulting in failure. The conditional constructs are covered by IfElse and TryElse: note that failure can only result from failure in the two branches, and not from the guard C, which is simply tested to choose the branch (with the effects of C being retained only by `try`). The proof rule Choice covers a derived GP 2 command that nondeterministically chooses one of two programs to execute. Note that the previous three proof rules only require one of the antecedents to be proved.

It is important to highlight the rule of consequence, Cons, as the implications in the side conditions are reversed from those of the corresponding Hoare logic rule [32, 19]. In incorrectness logic, we instead weaken the precondition and strengthen the postcondition. Intuitively, this allows us to soundly drop disjuncts in the result and thus reason about *fewer paths* in the post-state, which may support better scalability in tools [15].

The proof rule IterVar expresses a triple over parameterised assertions, i.e. functions mapping natural numbers to assertions in the language. The idea of the rule is to prove a 'backwards variant' of a single iteration of P, i.e. with respect to decreasing natural numbers in the parameters of the assertions. With this backwards variant proved, one can conclude that if a graph satisfies some $c(n)$, then it can be derived by *some* graph satisfying $c(0)$.

The Break rule is similar to IterVar, except that one must prove that there is an execution of P that ends with the command `break`. The ad hoc condition in the proof rule requires one to unfold the final iteration of P into a sequence of commands Q followed by the command `break`.

$$\textsc{SeqSucc}\ \frac{\vdash [c]\ P\ [ok\colon e] \qquad \vdash [e]\ Q\ [\epsilon\colon d]}{\vdash [c]\ P;\ Q\ [\epsilon\colon d]} \qquad \textsc{SeqFail}\ \frac{\vdash [c]\ P\ [fa\colon d]}{\vdash [c]\ P;\ Q\ [fa\colon d]}$$

$$\textsc{IfElse}\ \frac{\vdash [\mathrm{SE}[C, c]]\ P\ [\epsilon\colon d] \text{ or} \vdash [\mathrm{FE}[C, c]]\ Q\ [\epsilon\colon d]}{\vdash [c]\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ [\epsilon : d]}$$

$$\textsc{TryElse}\ \frac{\vdash [\mathrm{SE}[C, c]]\ C;\ \ P\ [\epsilon\colon d] \text{ or} \vdash [\mathrm{FE}[C, c]]\ Q\ [\epsilon\colon d]}{\vdash [c]\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ [\epsilon : d]}$$

$$\textsc{Choice}\ \frac{\vdash [c]\ P\ [\epsilon\colon d] \text{ or} \vdash [c]\ Q\ [\epsilon\colon d]}{\vdash [c]\ P\ \texttt{or}\ Q\ [\epsilon : d]}$$

$$\textsc{Cons}\ \frac{c \Longleftarrow c' \quad \vdash [c']\ P\ [\epsilon\colon d']\ \ d' \Longleftarrow d}{\vdash [c]\ P\ [\epsilon\colon d]}$$

$$\textsc{IterVar}\ \frac{\vdash [c(i-1)]\ P\ [ok : c(i)] \text{ for all } i : \mathbb{N}}{\vdash [c(0)]\ P!\ [ok : \exists n : \mathbb{N}_0.\ \mathrm{FE}[P, c(n)]]}$$

$$\textsc{Break}\ \frac{\exists n : \mathbb{N}_0.\ \vdash [c(i-1)]\ P\ [ok : c(i)] \text{ for } i \leq n : \mathbb{N} \text{ and} \vdash [c(n)]\ Q\ [ok : d] \text{ for } Q; \texttt{break} \text{ `in' } P}{\vdash [c(0)]\ P!\ [ok : d]}$$

Figure 12: Extensional incorrectness proof rules for graph programs

The provability of a triple can be shown in a number of different ways. In this paper, we will visualise them using *proof trees*, in which the triple to prove is the root, the instantiations of axioms are the leafs, and applications of proof rules are everything in-between.

Example 3.7 (Proof tree). Figure 13 depicts an example proof tree for the graph program of Example 2.13. This proof tree is given independently of any particular assertion language or assertions. In fact, this proof tree can be instantiated to prove any triple $\vdash [c]$ `init`; `Reduce` $[ok : d]$ so long as the various side conditions of the

$$\dfrac{\dfrac{\vdash [\mathrm{SE}[\texttt{init}, c]]\ \texttt{init};\ [ok : \mathrm{WPOST}_{ok}[\texttt{init}, c]]}{\vdash [c]\ \texttt{init};\ [ok : g]} \qquad \dfrac{\textit{(see sub-tree below)}}{\vdash [g]\ \texttt{Reduce}\ [ok : d]}}{\vdash [c]\ \texttt{init};\ \texttt{Reduce}\ [ok : d]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash [\mathrm{SE}[\texttt{red3}, f(i-1)]]\ \texttt{red3}\ [ok : \mathrm{WPOST}[\texttt{red3}, f(i-1)]]}{\vdash [f(i-1)]\ \texttt{red3}\ [ok : f(i)]}}{\vdash [f(0)]\ \texttt{red3!}\ [ok : \exists n : \mathbb{N}_0.f(n)]}}{\vdash [g]\ \texttt{red3!}\ [ok : e]} \qquad \dfrac{\vdash [\mathrm{SE}[\mathcal{R}, e]]\ \mathcal{R}\ [ok : \mathrm{WPOST}_{ok}[\mathcal{R}, e]]}{\vdash [e]\ \mathcal{R}\ [ok : d]}}{\vdash [g]\ \texttt{Reduce}\ [ok : d]}$$

Figure 13: Proof tree for Example 2.13 ($\mathcal{R} = \{\texttt{red2}, \texttt{red1}\}$) using extensional assertions

proof rules hold. For example, it must be the case that d implies $\mathrm{WPOST}_{ok}[\mathcal{R}, e]$, e implies $\exists n : \mathbb{N}_0.f(n)$, and that $f(0)$ implies g (among other implications) with respect to the semantics of the assertion language.

(Proof trees with a specific assertion language are explored in Section 5.) □

Soundness means that any triple provable in our logic is valid in the sense of Definition 3.6, i.e. that graphs satisfying the result are reachable from some graph satisfying the presumption.

Theorem 3.8 (Soundness)**.** Let $\mathfrak{A}$ denote an assertion language. For all assertions c, d in $\mathfrak{A}$, graph programs P, and exit statuses ϵ,

$$\vdash [c]\ P\ [\epsilon : d] \text{ implies } \models [c]\ P\ [\epsilon : d].$$

□

Proof. Given $\vdash [c]P[\epsilon : d]$, we need to show that $\models [c]P[\epsilon : d]$. We consider each axiom and proof rule in turn and proceed by induction on proofs.

RuleSetSucc. Suppose $H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[\mathcal{R}, c]$. By the definition of WPOST, there exists a graph G such that $G \models_{\mathfrak{A}} c$ and $(G, H) \in [\![\mathcal{R}]\!]_{ok}$. By the definition of SE, $G \models_{\mathfrak{A}} \mathrm{SE}[\mathcal{R}, c]$. It follows that $\models [\mathrm{SE}[\mathcal{R}, c]]\mathcal{R}[ok : \mathrm{WPOST}_{ok}[\mathcal{R}, c]]$.

RuleSetFail. Suppose $H \models_{\mathfrak{A}} \mathrm{FE}[\mathcal{R}, c]$. By the definition of FE, $H \models_{\mathfrak{A}} c$, and there exists a graph H' such $(H, H') \in [\![\mathcal{R}]\!]_{fa}$. There exists a sequence of configurations $\langle \mathcal{R}, H \rangle \to^* \langle \mathcal{R}, H' \rangle \to$ fail. By Figure 9, the sequence consists of a single step, $G \not\Rightarrow_{\mathcal{R}}$, $H' = H$, and $\langle \mathcal{R}, H \rangle \to$ fail. Together, $(H, H) \in [\![\mathcal{R}]\!]_{fa}$, and so it follows that $\models [\mathrm{FE}[\mathcal{R}, c]]\mathcal{R}[fa : \mathrm{FE}[\mathcal{R}, c]]$.

Skip, Fail. Immediate from the semantic rules [skip], [fail], and the definition of $\models$.

IterZero. For every graph $H.H \models_{\mathfrak{A}} \mathrm{FE}[P, c]$, by the definition of FE, $H \models_{\mathfrak{A}} c$, and there exists some $H'.(H, H') \in [\![P]\!]_{fa}$, and thus a sequence of configurations $\langle P, H\rangle \to^{+} \langle P, H'\rangle \to$ fail. By semantic rule [alap$_2$], $\langle P!, H\rangle \to H$ and thus $(H, H) \in [\![P!]\!]_{ok}$. Together, we get the result that $\models [\mathrm{FE}[P, c]]P![ok : \mathrm{FE}[P, c]]$.

SeqSucc. Suppose that $\vdash [c]P; Q[ok : d]$. By induction, we have $\models [c]P[ok : e]$ and $\models [e]Q[ok : d]$. By definition of $\models$, for all $H.H \models_{\mathfrak{A}} d$, there exists a $G'.G' \models_{\mathfrak{A}} e$ with $(G', H) \in [\![Q]\!]_{ok}$, i.e. $\langle Q, G'\rangle \to^{+} G$. Furthermore, for all $G'.G' \models_{\mathfrak{A}} e$, there exists a $G.G \models_{\mathfrak{A}} c$ with $(G, G') \in [\![P]\!]_{ok}$, i.e. $\langle P, G\rangle \to^{+} G'$. By semantic rules [seq$_1$], [seq$_2$], $\langle P; Q, G\rangle \to^{+} \langle Q, G'\rangle \to^{+} H$, and thus $(G, H) \in [\![P; Q]\!]_{ok}$. It follows that $\models [c]P; Q[ok : d]$. Analogous for case $\vdash [c]P; Q[fa : d]$.

SeqFail. Suppose that $\vdash [c]P; Q[fa : d]$. By induction, we have $\models [c]P[fa : d]$. For all $H.H \models_{\mathfrak{A}} d$, there exists a $G.G \models_{\mathfrak{A}} c$ with $(G, H) \in [\![P]\!]_{fa}$, and thus a sequence of configurations $\langle P, G\rangle \to^{+} \langle P', H\rangle \to$ fail. By semantic rules [seq$_1$] and [seq$_3$], $\langle P; Q, G\rangle \to^{+} \langle P'; Q, H\rangle \to$ fail, i.e. $(G, H) \in [\![P; Q]\!]_{fa}$. It follows that $\models [c]P; Q[fa : d]$.

IfElse. Suppose that $\vdash [c]\texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q[\epsilon : d]$. By induction, $\models [\mathrm{SE}[C, c]]P[\epsilon : d]$ or $\models [\mathrm{FE}[C, c]]Q[\epsilon : d]$. For all $H.H \models_{\mathfrak{A}} d$, there exists a graph G that satisfies $\mathrm{SE}[C, c]$ or $\mathrm{FE}[C, c]$, and $(G, H) \in [\![P]\!]_{\epsilon}$ or $(G, H) \in [\![Q]\!]_{\epsilon}$ respectively. By the definition of SE and FE, there exists a graph H' such that $(G, H') \in [\![C]\!]_{ok}$ or $(G, H') \in [\![C]\!]_{fa}$, and thus a sequence of configurations $\langle C, G\rangle \to^{+} H'$ or $\langle C, G\rangle \to^{+} \langle C', H'\rangle \to$ fail. By semantic rules [if$_1$] or [if$_2$], $\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle$ or $\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle$. Together, we have $(G, H) \in [\![\texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q]\!]_{\epsilon}$, and thus the result that $\models [c]\texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q[\epsilon : d]$.

TryElse, Choice. The proofs follow a similar structure to that for IfElse.

Cons. Suppose that $\vdash [c]P[\epsilon : d]$. By induction, we have $\models [c']P[\epsilon : d']$, $\models d \Longrightarrow d'$, and $\models c' \Longrightarrow c$. For all graphs $H.H \models_{\mathfrak{A}} d$, H also satisfies d', and thus there exists a graph $G.G \models_{\mathfrak{A}} c'$ such that $(G, H) \in [\![P]\!]_{\epsilon}$. By assumption, $G \models_{\mathfrak{A}} c$, and thus $\models [c]P[\epsilon : d]$.

IterVar. Suppose that $\vdash [c(0)]P![ok : \exists n : \mathbb{N}_0.\mathrm{FE}[P, c(n)]]$. If $n = 0$, we are done (see the proof of IterZero). Otherwise, by induction we have $\models [c(n - 1)]P[ok : c(n)]$. If $n - 1 = 0$, we are done, otherwise, by induction we have $\models [c(n - 2)]P[ok : c(n - 1)]$, and the process repeats for a finite number of steps until a pre-state satisfying $c(0)$ is obtained. Result obtained using the semantic rules [alap$_1$], [alap$_2$], and the definition of $\models$.

Break. Suppose that $\vdash [c(0)]P![ok : d]$. By induction, there exists some natural n such that $\models [c(n)]Q[ok : d]$, i.e. for all graphs $H.H \models_{\mathfrak{A}} d$, there exists a graph $G'.G' \models_{\mathfrak{A}} c(n)$ and $(G', H) \in [\![Q]\!]_{ok}$. Here, Q is a command sequence derived from P that is followed by the command `break`. If $n = 0$, then the result is obtained using semantic rule [alap$_3$]. If $n - 1 = 0$, we get the result by the assumption that $\models [c(i-1)]P[ok : c(i)]$. Otherwise, the process repeats for a finite number of steps until a pre-state satisfying $c(0)$ is obtained (as per IterVar). □

Completeness is the other side of the coin: it means that any valid triple can be proven using our logic. As is typical, we prove *relative completeness* [33] in which completeness is relative to the existence of an oracle for deciding the validity of assertions (such as the implications in Cons). The idea is to separate incompleteness due to the incorrectness logic from incompleteness in deducing valid assertions, and thus determine that no proof rules are missing.

Theorem 3.9 (Relative completeness)**.** Let $\mathfrak{A}$ denote an assertion language. For all assertions c, d in $\mathfrak{A}$, graph programs P, and exit statuses ϵ,

$$\models [c]\ P\ [\epsilon : d] \text{ implies } \vdash [c]\ P\ [\epsilon : d].$$

□

Proof. We prove relative completeness by showing that for every program P (consisting of core commands), extensional assertion c, and exit status $\epsilon \in \{ok, fa\}$, $\vdash [c]P[\epsilon : \text{WPOST}_\epsilon[P, c]]$. Relative completeness is obtained by applying the rule of consequence to $\vdash [c]P[\epsilon : \text{WPOST}_\epsilon[P, c]]$.

Rule Application ($\epsilon = ok$). Immediate from RuleSetSucc and Cons.

Rule Application ($\epsilon = fa$). Immediate from RuleSetFail, the definition of $[\![\mathcal{R}]\!]_{fa}$, and Cons.

Sequential Composition ($\epsilon = ok$). In this case,

$$\begin{aligned}
&H \models_{\mathfrak{A}} \text{WPOST}_{ok}[P; Q, c]\\
\text{iff } &\exists G.G \models_{\mathfrak{A}} c \text{ and } (G, H) \in [\![P; Q]\!]_{ok}\\
\text{iff } &\exists G, G'.G \models_{\mathfrak{A}} c, (G, G') \in [\![P]\!]_{ok}, \text{ and } (G', H) \in [\![Q]\!]_{ok}\\
\text{iff } &\exists G'.G' \models_{\mathfrak{A}} \text{WPOST}_{ok}[P, c] \text{ and } (G', H) \in [\![Q]\!]_{ok}\\
\text{iff } &H \models_{\mathfrak{A}} \text{WPOST}_{ok}[Q, \text{WPOST}_{ok}[P, c]]
\end{aligned}$$

By induction we have $\vdash [\text{WPOST}_{ok}[P, c]]Q[ok : \text{WPOST}_{ok}[Q, \text{WPOST}_{ok}[P, c]]]$ and $\vdash [c]P[ok : \text{WPOST}_{ok}[P, c]]$. By SeqSucc we derive the triple $\vdash [c]P; Q[ok : \text{WPOST}_{ok}[Q, \text{WPOST}_{ok}[P, c]]]$, and by Cons $\vdash [c]P; Q[ok : \text{WPOST}_{ok}[P; Q, c]]$.

Sequential Composition ($\epsilon = fa$). In this case,

$$\begin{aligned}
&H \models_{\mathfrak{A}} \mathrm{WPOST}_{fa}[P; Q, c] \\
\text{iff } &\exists G. G \models_{\mathfrak{A}} c \text{ and } (G, H) \in [\![P; Q]\!]_{fa} \\
\text{iff } &\exists G. G \models_{\mathfrak{A}} c \text{ and } ((G, H) \in [\![P]\!]_{fa}) \\
&\text{or } (\exists G'. (G, G') \in [\![P]\!]_{ok} \text{ and } (G', H) \in [\![Q]\!]_{fa}) \\
\text{iff } &H \models_{\mathfrak{A}} \mathrm{WPOST}_{fa}[P, c] \\
&\text{or } H \models_{\mathfrak{A}} \mathrm{WPOST}_{fa}[Q, \mathrm{WPOST}_{ok}[P, C]]
\end{aligned}$$

If $H \models_{\mathfrak{A}} \mathrm{WPOST}_{fa}[P, c]$, then by induction we have $\vdash [c]P[fa : \mathrm{WPOST}_{fa}[P, c]]$, and by SeqFail derive $\vdash [c]P; Q[fa : \mathrm{WPOST}_{fa}[P, c]]$. With Cons we get $\vdash [c]P; Q[fa : \mathrm{WPOST}_{fa}[P; Q, c]]$.

If $H \models_{\mathfrak{A}} \mathrm{WPOST}_{fa}[Q, \mathrm{WPOST}_{ok}[P, C]]$, then by induction we get the triple $\vdash [\mathrm{WPOST}_{ok}[P, C]]Q[fa : \mathrm{WPOST}_{fa}[Q, \mathrm{WPOST}_{ok}[P, C]]]$ as well as the triple $\vdash [c]P[ok : \mathrm{WPOST}_{ok}[P, C]]$. By SeqSucc, we can derive the triple $\vdash [c]P; Q[fa : \mathrm{WPOST}_{fa}[Q, \mathrm{WPOST}_{ok}[P, C]]]$. With Cons we get the result $\vdash [c]P; Q[fa : \mathrm{WPOST}_{fa}[P; Q, c]]$.

If-then-else. In this case,

$$\begin{aligned}
&H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[\texttt{if } C \texttt{ then } P \texttt{ else } Q, c] \\
\text{iff } &\exists G. G \models_{\mathfrak{A}} c \text{ and } (G, H) \in [\![\texttt{if } C \texttt{ then } P \texttt{ else } Q]\!]_{ok} \\
\text{iff } &\exists G. G \models_{\mathfrak{A}} c \text{ and } (\exists H'. (G, H') \in [\![C]\!]_{ok} \text{ and } (G, H) \in [\![P]\!]_{ok}) \\
&\text{or } (\exists H'. (G, H') \in [\![C]\!]_{fa} \text{ and } (G, H) \in [\![Q]\!]_{ok}) \\
\text{iff } &H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[P, \mathrm{SE}[C, c]] \text{ or } H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[Q, \mathrm{FE}[C, c]]
\end{aligned}$$

If $H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[P, \mathrm{SE}[C, c]]$, then by induction we have $\vdash [\mathrm{SE}[C, c]]P[ok : \mathrm{WPOST}_{ok}[P, \mathrm{SE}[C, c]]]$. Using IfElse we derive $\vdash [c]\texttt{if } C \texttt{ then } P \texttt{ else } Q[ok : \mathrm{WPOST}_{ok}[P, \mathrm{SE}[C, c]]]$. With Cons we get the result $\vdash [c]\texttt{if } C \texttt{ then } P \texttt{ else } Q[ok : \mathrm{WPOST}_{ok}[\texttt{if } C \texttt{ then } P \texttt{ else } Q, c]]$. (Case $H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[Q, \mathrm{FE}[C, c]]$ analogous.)

Try-then-else. Analogous to the if-then-else case.

Iteration. Define $c(i) = \mathrm{WPOST}_{ok}[P, c(i-1)]$ for all $i > 0$. Then, for `break-`

free executions of $P!$,

$$\begin{aligned}
&H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[P!, c(0)] \\
\text{iff } &\exists G_0 . G_0 \models_{\mathfrak{A}} c(0) \text{ and } (G_0, H) \in [\![P!]\!]_{ok} \\
\text{iff } &\exists G_0 . G_0 \models_{\mathfrak{A}} c(0) \text{ and } (G_0, H) \in [\![P!]\!]_{ok} \text{ and } (H, H) \in [\![P]\!]_{fa} \\
\text{iff } &\exists G_0 . G_0 \models_{\mathfrak{A}} c(0) \text{ and } \exists n : \mathbb{N}_0 . \exists G_1, \ldots G_n . (G_{i-1}, G_i) \in [\![P]\!]_{ok} \text{ for all } 0 < i \leq n \\
&\quad \text{and } G_n = H \text{ and } (H, H) \in [\![P]\!]_{fa} \\
\text{iff } &H \models_{\mathfrak{A}} \exists n : \mathbb{N}_0 . \mathrm{FE}[P, c(n)]
\end{aligned}$$

By induction, $\vdash [c(i-1)]P[ok : \mathrm{WPOST}_{ok}[P, c(i-1)]]$ and thus $\vdash [c(i-1)]P[ok : c(i)]$. By IterVar and Cons we get the result $\vdash [c(0)]P![ok : \mathrm{WPOST}_{ok}[P!, c(0)]]$.

For executions of $P!$ that terminate with Q; `break`,

$$\begin{aligned}
&H \models_{\mathfrak{A}} \mathrm{WPOST}_{ok}[P!, c(0)] \\
\text{iff } &\exists G_0 . G_0 \models_{\mathfrak{A}} c(0) \text{ and } (G_0, H) \in [\![P!]\!]_{ok} \\
\text{iff } &\exists G_0 . G_0 \models_{\mathfrak{A}} c(0) \text{ and } \exists n : \mathbb{N}_0 . \exists G_1, \ldots G_n . (G_{i-1}, G_i) \in [\![P]\!]_{ok} \text{ for all } 0 < i \leq n \\
&\quad \text{and } (G_n, H) \in [\![Q]\!]_{ok} \\
\text{iff } &H \models_{\mathfrak{A}} \exists n : \mathbb{N}_0 . \mathrm{WPOST}_{ok}[Q, c(n)]
\end{aligned}$$

By induction, $\vdash [c(i-1)]P[ok : c(n)]$ and $\vdash [c(n)]Q[ok : \mathrm{WPOST}_{ok}[Q, c(n)]]$. By Break and Cons we derive the result that $\vdash [c]P![ok : \mathrm{WPOST}_{ok}[P!, c(0)]$. □

4. Verifying Monadic Second-Order Graph Properties

The extensional style of our under-approximate calculi allowed us to study proof rules for the constructs of graph programs in isolation from issues associated with particular assertion languages, such as inexpressiveness. For the calculi to be usable in verification tasks, they must of course be instantiated with an assertion language. In choosing a suitable formalism for this purpose, we aim to satisfy a number of requirements: that the language (1) can specify a broad class of properties about graphs labelled over an infinite label alphabet; (2) supports formal, logical reasoning; (3) has a decision procedure for model checking; and (4) is equipped with applicability and weakest postcondition constructions.

We propose *monadic second-order nested conditions with expressions* (or *ME-conditions* for short) as an assertion language that satisfies these requirements. ME-conditions are visual and based on morphisms, allowing for properties to be specified and reasoned about at the same level of abstraction as rule schemata and graphs. Equivalently expressive to monadic second-order logic on graphs, ME-conditions can specify non-local structural properties (e.g. the graph is connected, is bipartite, has an arbitrary-length path between two nodes) in conjunction with properties involving label expressions (e.g. every edge is labelled with the sum of its integer-labelled incident nodes). ME-conditions unify several previously proposed morphism-based assertion languages into a single formalism: nested conditions [7], for specifying local structure; M-conditions [25], for specifying non-local structure; and E-conditions [13], for specifying properties over infinite label alphabets.

4.1. MSO Conditions with Expressions

We begin by defining the syntax of ME-conditions, which are over graphs in $\mathcal{G}(\mathrm{RG})$, as well as the semantics, which are with respect to graphs and morphisms over $\mathcal{G}(\mathcal{L})$.

Similar to rule schemata, these ME-conditions can constrain the potential assignments of variables using *assignment constraints*. These differ slightly from the constraints of rule schemata as they include predicates over monadic second-order set variables (i.e. sets of nodes or sets of edges). This also requires us to extend the definition of assignment to encompass these new types of variables.

Definition 4.1 (Assignments). Let $G \in \mathcal{G}(\mathcal{L})$ denote a graph. An *assignment for G* is a family of partial functions $\alpha = (\alpha_X)_{X\in\{\mathtt{L},\mathtt{V},\mathtt{E}\}}$ where $\alpha_{\mathtt{L}}$ is a list assignment, $\alpha_{\mathtt{V}}\colon \mathrm{VSetVar} \to 2^{V_G}$, and $\alpha_{\mathtt{E}}\colon \mathrm{VSetVar} \to 2^{E_G}$. □

For notational convenience, we will omit the subscript in assignments when the context is unambiguous.

Definition 4.2 (Assignment constraint). An *assignment constraint* γ is a Boolean expression that can be derived from AssCon in the grammar of Figure 14. □

Note that Integer has been extended with integer expressions of the form $i = \mathtt{card(X)}$ with $\mathtt{X}$ in VSetVar or ESetVar. Here, `card` represents the *cardinality* function, the value of which (with respect to an assignment α) is simply $|\alpha(\mathtt{X})|$.

Given an assignment constraint γ, a morphism g with codomain G, and an assignment α, the value of $\gamma^{g,\alpha}$ in $\mathbb{B}$ is defined inductively. If γ has the form $x \in \mathtt{X}$ with x a node/edge identifier and $\mathtt{X}$ a node/edge set variable, then $\gamma^{g,\alpha}$ is true if $g(x) \in \alpha(\mathtt{X})$. If γ has the form $\mathtt{path}(v,w)$ with v, w node identifiers, then $\gamma^{g,\alpha}$

```
AssCon   ::=  List (’=’ | ’!=’) List | Integer IntRel Integer | Type ’(’ List ’)’
              | Node ’∈’ (VSetVar | ’{}’ | ’{’ Node {’,’ Node } ’}’)
              | Edge ’∈’ (ESetVar | ’{}’ | ’{’ Edge {’,’ Edge } ’}’)
              | path ’(’ Node ’,’ Node [’,’ not Edge {’|’ Edge}] ’)’
IntRel   ::=  ’>’ | ’>=’ | ’<’ | ’<=’
Type     ::=  int | char | string | atom
Integer  ::=  IVar | [‘-’] Digit {Digit} | ‘(’Integer‘)’ |
              Integer (‘+’ | ‘-’ | ‘*’ | ‘/’) Integer |
              (indeg | outdeg) ‘(’Node‘)’ |
              length ‘(’(LVar | AVar | SVar)‘)’
              card ‘(’(VSetVar | ESetVar)‘)’
```

Figure 14: Abstract syntax of assignment constraints

is true if $\text{path}_G(g(v), g(w), \emptyset)$ holds. If γ has the form $\texttt{path(}v\texttt{,}w\texttt{,not } e_1 \,|\, \ldots \,|\, e_n\texttt{)}$ with v, w node identifiers and each e_i an edge identifier, then $\gamma^{g,\alpha}$ is true if the path predicate $\text{path}_G(g(v), g(w), \{g(e_1), \ldots, g(e_n)\})$ holds. (Other cases are analogous to those for rule schema conditions.)

Definition 4.3 (ME-condition). Let P denote a graph in $\mathcal{G}(\text{RG})$. An *MSO condition with expressions (*short. *ME-condition) over P* is of the form $\texttt{true}$, γ, $\exists_{\text{L}}\texttt{x}.c$, $\exists_{\text{V}}\texttt{X}.c$, $\exists_{\text{E}}\texttt{X}.c$, or $\exists a.c'$, where γ is an assignment constraint, $\texttt{x}$ is a variable in LVar, $\texttt{X}$ is a variable in VSetVar (resp. ESetVar), c is a ME-condition over P, $a\colon P \hookrightarrow C$ is an injective graph morphism over $\mathcal{G}(\text{RG})$, and c' is a ME-condition over C. Moreover, $\neg c_1$, $c_1 \wedge c_2$, and $c_1 \vee c_2$ are ME-conditions over P if c_1, c_2 are ME-conditions over P. □

The *free variables* of a ME-condition c, denoted $\text{FV}(c)$, are those variables present in labels and assignment constraints that are not *bound* by any variable quantifier (defined in the standard way). If c is defined over the empty graph $\emptyset$ and $\text{FV}(c) = \emptyset$, we call c a *ME-constraint*. Furthermore, a mapping of free variables to expressions and node/edge sets $\sigma = (\texttt{x}_1 \mapsto e_1, \cdots, \texttt{X}_1 \mapsto V_1, \cdots)$ is called a *substitution*, and c^σ denotes the ME-condition c but with all free variables $\texttt{x}$ substituted for $\sigma(\texttt{x})$.

Definition 4.4 (Satisfaction of ME-conditions). Let c denote a ME-condition over P, α an assignment constraint with $\text{dom}(\alpha) = \text{FV}(c)$, and $p\colon P^\alpha \hookrightarrow G$ an injective morphism over $\mathcal{G}(\mathcal{L})$. The *satisfaction* relation $p \models^\alpha c$ is defined inductively.

If c has the form $\mathtt{true}$, then $p \models^{\alpha} c$ always. If c is an assignment constraint γ, then $p \models^{\alpha} c$ if $\gamma^{p,\alpha} =$ true. If c has the form $\exists_{\mathtt{L}}\mathtt{x}.c'$ where c' is a ME-condition over P, then $p \models^{\alpha} c$ if $p \models^{\alpha[\mathtt{x}\mapsto l]} c'$ for some $l \in \mathbb{L}$. If c has the form $\exists_{\mathtt{V}}\mathtt{X}.c'$ where c' is a ME-condition over P, then $p \models^{\alpha} c$ if $p \models^{\alpha[\mathtt{X}\mapsto V]} c'$ for some $V \subseteq V_G$. If c has the form $\exists_{\mathtt{E}}\mathtt{X}.c'$ where c' is a ME-condition over P, then $p \models^{\alpha} c$ if $p \models^{\alpha[\mathtt{X}\mapsto E]} c'$ for some $E \subseteq E_G$.

If c has the form $\exists a : P \hookrightarrow C.c'$ where c' is a ME-condition over C, then $p \models^{\alpha} c$ if there exists an injective morphism $q\colon C^{q,\alpha} \hookrightarrow G$ such that $q \circ a^{q,\alpha} = p$ and $q \models^{\alpha} c'$.

$$\begin{tikzcd}
P^{q,\alpha} \arrow[rr, hook, "a^{q,\alpha}"] \arrow[dr, hook, "p"'] \arrow[rr, phantom, "=" below, shift right=3] & & C^{q,\alpha} \arrow[dl, hook', "q \models^{\alpha} c'"] \\
& G &
\end{tikzcd}$$

Finally, the satisfaction of Boolean formulae over ME-conditions is defined in the standard way. □

The satisfaction of ME-constraints by graphs is defined as a special case of the general definition. That is, a graph $G \in \mathcal{G}(\mathcal{L})$ *satisfies* a ME-constraint c, denoted $G \models c$, if $i_G\colon \emptyset \hookrightarrow G \models^{\alpha_\emptyset} c$, where $\alpha_\emptyset$ is the empty assignment, i.e. with $\mathrm{dom}(\alpha_\emptyset) = \emptyset$.

For brevity, we write $\mathtt{false}$ for $\neg\mathtt{true}$, $c \Longrightarrow d$ for $\neg c \vee d$, $\forall\mathtt{x}.c$ for $\neg\exists\mathtt{x}.\neg c$, $\forall a.c$ for $\neg\exists a.\neg c$, and $\exists\mathtt{x}_1,\cdots\mathtt{x}_n.c$ for $\exists\mathtt{x}_1.\cdots\exists\mathtt{x}_n.c$ (analogous for $\forall$ and set variables $\mathtt{X}$). We also allow sub-type quantifiers such as $\exists_{\mathtt{I}}\mathtt{x}.c$ to abbreviate $\exists_{\mathtt{L}}\mathtt{x}.\mathtt{int(x)}.c$.

Furthermore, if the domain of a morphism can be unambiguously inferred from the context, we write only the codomain. For example, the ME-constraint $\exists\emptyset \hookrightarrow C.\ \exists C \hookrightarrow C'.\ \mathtt{true}$ can be written as $\exists C.\ \exists C'$.

Example 4.5 (ME-constraint). Consider the following ME-constraint:

$$\exists_{\mathtt{L}}\mathtt{c},\mathtt{d}.\exists\textcircled{c}_{v}\textcircled{d}_{w}.\mathtt{path}(v,w) \vee \mathtt{path}(w,v)$$

Intuitively, this expresses that there exists a pair of (unmarked, non-rooted) nodes labelled with any list component, that are connected by an arbitrary-length path in either direction.

With the morphism written out in full, the ME-constraint would be:

$$\exists_{\mathtt{L}}\mathtt{c},\mathtt{d}.\exists\emptyset \hookrightarrow \textcircled{c}_{v}\textcircled{d}_{w}.\mathtt{path}(v,w) \vee \mathtt{path}(w,v)$$

□

Theorem 4.6 (Equivalence to MSO on graphs)**.** ME-constraints and monadic second-order formulas (with cardinality) on graphs [34] are equivalently expressive. That is, given a ME-constraint c, there exists an MSO graph formula φ such that for all graphs G, $G \models c$ if and only if G satisfies φ, and vice versa. □

Proof. There exist sound translations between the morphisms/expressions of ME-conditions and first-order logic on graphs (see Chapter §6, [12]), as well as sound translations between the set variable quantifiers/constraints of ME-conditions and monadic-second order logic on graphs (see [25]). The result is obtained through a straightforward combination of these translations. □

4.2. Constructing SE and FE

For the remainder of the paper, we utilise ME-constraints together with the $\models$ relation as our assertion language for conducting proofs. This requires us to define transformations over ME-conditions and graph programs that produce ME-constraints characterising the extensional SE, FE, and WPOST assertions from Figures 11 and 12.

First, we consider 'App', which transforms a set of conditional rule schemata into a ME-constraint that expresses the minimum requirements on a graph for at least one of the rules to be applicable. Intuitively, the ME-constraint specifies the presence of a match for a left-hand side, including a morphism that satisfies the dangling condition. In other words, App($\mathcal{R}$) can be used to define the extensional assertions SE and FE with respect to sets of rule schemata. (In general, we cannot define SE and FE for arbitrary programs, as that would require the assertion language to be able to decide the halting problem. We remark, however, that this restriction does not affect the computational completeness of GP 2 [35].)

In order to define App, we utilise two intermediate transformations adapted from previous work [7, 12]. First, 'Dang', which is used to produce a ME-condition expressing that a morphism satisfies the dangling condition. Second, τ, which is used to transform a rule schema condition into an assignment constraint.

Lemma 4.7 (Dangling condition)**.** For every morphism $L \hookleftarrow K$ in $\mathcal{G}(\mathrm{RG})$, and every morphism $g\colon L^{g,\alpha} \hookrightarrow G$ in $\mathcal{G}(\mathcal{L})$,

$$g \models^{\alpha} \mathrm{Dang}(K \hookrightarrow L) \text{ if and only if } g \text{ satisfies the dangling condition for } L \hookleftarrow K.$$

Construction. Define $\mathrm{Dang}(K \hookrightarrow L) = \bigwedge_{a \in A} \neg\exists_{\mathrm{L}} \mathtt{x}_e, \mathtt{x}_v.\exists a$ where the index set A ranges over all injective morphisms (equated up to isomorphic codomains) $a\colon L \hookrightarrow L^{\oplus}$ such that the pair $\langle K \hookrightarrow L, a\rangle$ has no natural pushout complement and

$$\begin{tikzcd}
L^{g,\alpha} \arrow[d, "a^{g',\alpha'}"'] \arrow[dd, bend right, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l] \arrow[d] \arrow[dd, bend left] \\
(L^{\oplus})^{g',\alpha'} \arrow[d, "g'"'] \arrow[dr, phantom, "(2)"] & K' \arrow[l] \arrow[d] \\
G & D \arrow[l]
\end{tikzcd}$$

Figure 15: Diagram chasing for a contradiction

each $L^{\oplus}$ is a graph that can be obtained from L by adding either: (1) a single loop with label $\mathtt{x}_e$; (2) a single edge with label $\mathtt{x}_e$ between distinct nodes; (3) a single rooted/non-rooted node labelled with $\mathtt{x}_v$ and a non-looping edge incident to it with label $\mathtt{x}_v$; or variants of (1)–(3) but having '`any`' as the mark(s) of the new edge/node. If the index set A is empty, then $\mathrm{Dang}(r) = \mathtt{true}$. □

Proof. ($\Longrightarrow$) Assume that $g \models^{\alpha} \mathrm{Dang}(K \hookrightarrow L)$. Then $g \models^{\alpha} \bigwedge_{a \in A} \neg\exists_{\mathrm{L}} \mathtt{x}_e, \mathtt{x}_v.\exists a$ over the index set A from the construction, i.e. for each $a\colon L \hookrightarrow L^{\oplus}$, there does not exist some g', α' such that $g' \models^{\alpha'} a$. The morphism g satisfies the dangling condition for $L \hookleftarrow K$ if no edge in $G - g(L^{g,\alpha})$ is in the image of $g(L^{g,\alpha} - K)$. This can be shown by assuming the existence of such an edge, and then deriving a contradiction by the fact that the case is covered in A.

($\Longleftarrow$) Assume that g satisfies the dangling condition for $L \hookleftarrow K$. Then the pair $\langle K \hookrightarrow L^{g,\alpha}, g\rangle$ has a pushout complement $D \in \mathcal{G}(\mathcal{L})$. Assume there exists an $a \in A$ (from the construction) such that $\langle K \hookrightarrow L, a\rangle$ has no pushout complement, and there exists some assignment α' and morphism $g' : (L^{\oplus})^{g',\alpha'} \hookrightarrow G$ with $g' \circ a^{g',\alpha'} = g$. Construct (2) (Figure 15) as a pullback of $(L^{\oplus})^{g',\alpha'} \hookrightarrow G \hookleftarrow D$. By the universal property of pullbacks, there is a morphism $K \hookrightarrow K'$ such that the resulting diagrams commute. By the pushout-pullback decomposition, (1)+(2) has a decomposition into pushout (1) and (2), thus $\langle K \hookrightarrow L^{g,\alpha}, a^{g',\alpha'}\rangle$ and $\langle K \hookrightarrow L, a\rangle$ have pushout complements. A contradiction. There is no morphism in A that can be satisfied, and so $g \models^{\alpha} \mathrm{Dang}(K \hookrightarrow L)$. □

Lemma 4.8 (Encoding rule schema conditions)**.** For every conditional rule schema $r = \langle L \Rightarrow R, \Gamma\rangle$, assignment α with $\mathrm{dom}(\alpha) = \mathrm{dom}(L)$, and match $g\colon L^{g,\alpha} \hookrightarrow G$,

$$g \models^{\alpha} \tau(L, \Gamma) \text{ if and only if } \Gamma^{g,\alpha} = \text{true}.$$

Construction. The transformation is defined inductively as follows (see Figure 4 for the syntax of rule schema conditions). If Γ has the form $type\texttt{(x)}$, $l_1 = l_2$, $l_1 \ \texttt{!=}\ l_2$, or $i_1 \bowtie i_2$ (where $\bowtie$ is a relational operator on integers), then $\tau(L, \Gamma) = \Gamma$. If Γ has the form $\texttt{edge(}v_1\texttt{,}v_2\texttt{)}$ where $v_1, v_2 \in V_L$, then $\tau(L, \Gamma) = \exists_{\mathrm{L}}\mathtt{x}.\exists L \hookrightarrow L_u \vee \exists L \hookrightarrow L_m$ where $\mathtt{x}$ is a fresh variable, and where L_u, L_m are constructed from L by disjointly adding an edge e with source $s_L(v_1)$, target $t_L(v_2)$, and respectively label $\mathtt{x}$ or $(\mathtt{x}, \mathtt{any})$. If Γ has the form $\texttt{edge(}v_1\texttt{,}v_2\texttt{,}l\texttt{)}$ where $v_1, v_2 \in V_L$ and l in Label, then $\tau(L, \Gamma) = \exists L \hookrightarrow L_u \vee \exists L \hookrightarrow L_m$ where each L_u, L_m are constructed from L by disjointly adding an edge e with source $s_L(v_1)$, target $t_L(v_2)$, and respectively label l or $(l, \mathtt{any})$. If Γ has the form $\texttt{not}\ \Gamma'$, then $\tau(L, \Gamma) = \neg\Gamma$. If Γ has the form $\Gamma_1\ \texttt{and}\ \Gamma_2$, then $\tau(L, \gamma) = \Gamma_1 \wedge \Gamma_2$ (analogous for $\texttt{or}$). If Γ has the form $\texttt{(}\Gamma'\texttt{)}$, then $\tau(L, \Gamma) = \Gamma'$. □

Proof. By structural induction over the grammar defining rule schema conditions. □

Proposition 4.9 (Applicability)**.** For every graph $G \in \mathcal{G}(\mathcal{L})$, ME-constraint c, and set of rule schemata $\mathcal{R}$,

$$G \models \mathrm{App}(\mathcal{R}) \wedge c \text{ if and only if } G \models \mathrm{SE}[\mathcal{R}, c].$$

Construction. Define $\mathrm{App}(\emptyset) = \texttt{false}$ and $\mathrm{App}(\mathcal{R}) = \bigvee_{r \in \mathcal{R}} \mathrm{app}(r)$ where $\mathrm{app}(r) = \exists_{t_1}\mathtt{x}_1. \cdots \exists_{t_n}\mathtt{x}_n.\ \exists\emptyset \hookrightarrow L.\ \tau(L, \Gamma) \wedge \mathrm{Dang}(K \hookrightarrow L)$. Here, $\mathrm{vars}(L) = \{\mathtt{x}_1, \cdots \mathtt{x}_n\}$, and each t_i indicates the type of variable $\mathtt{x}_i$. □

Proof. ($\Longrightarrow$) Assume that $G \models \mathrm{App}(\mathcal{R}) \wedge c$. Then there exists some $r = \langle L \Rightarrow R, \Gamma\rangle$ such that $i_G \models^{\alpha_\emptyset} \exists_{t_1}\mathtt{x}_1. \cdots \exists_{t_n}\mathtt{x}_n.\ \exists\emptyset \hookrightarrow L.\ \tau(L, \Gamma) \wedge \mathrm{Dang}(K \hookrightarrow L)$. There exists a morphism $q \colon L^{q,\alpha} \hookrightarrow G$ defined for variables $\mathtt{x}_1, \cdots \mathtt{x}_n$ such that $q \circ \emptyset \hookrightarrow L^{q,\alpha} = i_G$, $q \models^{\alpha} \tau(L, \Gamma)$, and $q \models^{\alpha} \mathrm{Dang}(K \hookrightarrow L)$. By Lemma 4.8, $\Gamma^{q,\alpha} = \mathrm{true}$. By Lemma 4.7, q satisfies the dangling condition for $K \hookleftarrow L$. There exists a rule application $G \Rightarrow_r H$ and $G \Rightarrow_{\mathcal{R}} H$ with match q and thus $(G, H) \in [\![\mathcal{R}]\!]_{ok}$. By Definition 3.3, $G \models \mathrm{SE}[\mathcal{R}, c]$.

($\Longleftarrow$) Assume that $G \models \mathrm{SE}[\mathcal{R}, c]$. By Definition 3.3, $G \models c$ and there exists a graph H such that $(G, H) \in [\![\mathcal{R}]\!]_{ok}$. By the semantics of rule schema sets, there exists an $r \in \mathcal{R}$ such that $G \Rightarrow_r H$ for some match $g : L^{g,\alpha} \hookrightarrow G$ that satisfies the dangling condition with $\Gamma^{g,\alpha} = \mathrm{true}$. By Lemmas 4.7 and 4.8, $g \models^{\alpha} \tau(L, \Gamma) \wedge \mathrm{Dang}(K \hookrightarrow L)$. Given that $g \circ \emptyset \hookrightarrow L^{g,\alpha} = i_G$ and α is defined exactly for $\mathrm{vars}(L)$, we get $G \models \exists_{t_1}\mathtt{x}_1. \cdots \exists_{t_n}\mathtt{x}_n.\ \exists\emptyset \hookrightarrow L.\tau(L, \Gamma) \wedge \mathrm{Dang}(K \hookrightarrow L) = \mathrm{app}(r)$ and thus the result $G \models \mathrm{App}(\mathcal{R}) \wedge c$. □

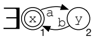
Proposition 4.10 (Non-applicability)**.** For every graph $G \in \mathcal{G}(\mathcal{L})$, ME-constraint c, and set of rule schemata $\mathcal{R}$,

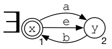
$$G \models \neg\text{App}(\mathcal{R}) \wedge c \text{ if and only if } G \models \text{FE}[\mathcal{R}, c].$$

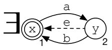
□

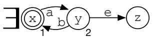
Proof. Immediate from Definition 3.4, the definition of ¬, and Proposition 4.9. □

Example 4.11 (Applicability). Consider the rule schema `red2` (Figure 8). The transformation App(`red2`) results in the following ME-constraint:

$\exists_{\mathrm{L}}\,\mathsf{a}, \mathsf{b}, \mathsf{x}, \mathsf{y}.\exists$ 

$.\neg\exists_{\mathrm{L}}\,\mathsf{e}.\exists$ 

$\wedge\ \neg\exists_{\mathrm{L}}\,\mathsf{e}.\exists$ 

$\wedge\ \neg\exists_{\mathrm{L}}\,\mathsf{e}, \mathsf{z}.\exists$ 

$\wedge \cdots$

The applicability of the rule schema rests on there being a morphism that satisfies the dangling condition. In this case, a morphism satisfies the dangling condition for `red2` if there are no additional edges incident to nodes in the codomain of the morphism. For brevity, the ME-constraint omits a number of conjuncts, e.g. those involving loops incident to nodes 1 and 2, or those involving edges between node 1 and some other node not in the match. □

4.3. Constructing Weakest Postconditions

Next, we propose a transformation 'WPost' that defines the extensional assertion WPOST for rule schemata and ME-constraints. In particular, WPost transforms a set of rule schemata and a presumption into a weakest postcondition, i.e. the weakest property a graph must satisfy to guarantee the *existence* of a pre-state that satisfies the presumption. WPost is defined via two intermediate transformations: 'Shift' and 'Right'.

We begin by defining 'Shift', which can be used to transform a ME-constraint c into a ME-condition over the left-hand side of a rule L by considering all the ways that a 'match' can overlap with c. Our definition is adapted from earlier shifting constructions in [7, 12, 24, 25] to handle the presence of list variables. Intuitively, this is handled via a disjunction over all possible substitutions of a variable in c for list expressions or variables in L, i.e. to account for assignments in which they refer to the same values.

To facilitate this, we require that the labels in c are list variables distinct from those in L. This is a mild assumption, as an arbitrary expression can be replaced with a variable that is then equated with the original expression in an assignment constraint.

Lemma 4.12 (ME-constraint to left ME-condition)**.** Let r denote a rule schema and c a ME-constraint labelled over list variables distinct from those in r. For every graph $G \in \mathcal{G}(\mathcal{L})$ and morphism $g\colon L^{g,\alpha} \hookrightarrow G$ with $\mathrm{dom}(\alpha) = \mathrm{vars}(L)$,

$$g\colon L^{g,\alpha} \hookrightarrow G \models^{\alpha} \mathrm{Shift}(r, c) \text{ if and only if } G \models c.$$

Construction. Let c denote a ME-constraint and r a rule schema with left-hand side L. Define $\mathrm{Shift}(r, c) = \mathrm{Shift}'(\emptyset \hookrightarrow L, c)$. We define Shift′ inductively for morphisms $p\colon p \hookrightarrow P'$ over $\mathcal{G}(\mathrm{RG})$ and ME-conditions over P. Let $\mathrm{Shift}'(p, \mathtt{true}) = \mathtt{true}$ and $\mathrm{Shift}'(p, \gamma) = \gamma$. Then,

$$\begin{aligned} \mathrm{Shift}'(p, \exists_{\mathtt{L}}\mathtt{x}.\ c) &= \exists_{\mathtt{L}}\mathtt{x}.\ \mathrm{Shift}'(p, c) \\ \mathrm{Shift}'(p, \exists_{\mathtt{V}}\mathtt{X}.\ c) &= \exists_{\mathtt{V}}\mathtt{X}.\ \mathrm{Shift}'(p, c) \\ \mathrm{Shift}'(p, \exists_{\mathtt{E}}\mathtt{X}.\ c) &= \exists_{\mathtt{E}}\mathtt{X}.\ \mathrm{Shift}'(p, c) \\ \mathrm{Shift}'(p, \exists a\colon P \hookrightarrow C.\ c) &= \bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_{\sigma}} \exists b\colon P' \hookrightarrow E.\ \mathrm{Shift}'(s\colon C^{\sigma} \hookrightarrow E, c^{\sigma}) \end{aligned}$$

Here, Σ is a set of substitutions, including the empty substitution, and all substitutions of the form $(\mathtt{x}_1 \mapsto e_1, \ldots \mathtt{x}_n \mapsto e_n)$, where each $\mathtt{x}_i \in \mathrm{vars}(C) \setminus \mathrm{vars}(P)$ and each e_i is a list expression present in the labels of $V_{P'}$ and $E_{P'}$. Construct pushout (1) of p and a as depicted in Figure 16. The disjunction ranges over the set ε_{σ}, which we define to contain every surjective morphism $e : (C')^{\sigma} \hookrightarrow E$ such that $b = e \circ a'$ and $s = e \circ q$ are injective morphisms. (We consider codomains of each e up to isomorphism, so the disjunction is finite.) Whenever nodes/edges are equated in this case, we substitute the node/edge identifier used in P' for the one it is equated with in C.

Shift and Shift' are defined for Boolean formulae over ME-conditions in the standard way. □

The proof relies on a more general version of the lemma given in the Appendix (Lemma B.1).

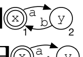
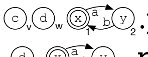
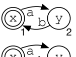
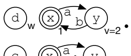
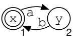
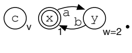
Proof. By Lemma B.1, $g\colon L^{g,\alpha} \hookrightarrow G \models^{\alpha} \mathrm{Shift}(r, c) = \mathrm{Shift}'(i_L : \emptyset \hookrightarrow L, c)$ if and only if $g \circ i_L^{g,\alpha} \models^{\alpha} c$ if and only if $i_G : \emptyset \hookrightarrow G \models^{\alpha} c$ if and only if $i_G \models^{\alpha_{\emptyset}} c$ if and only if $G \models c$. □

Example 4.13 (ME-constraint to left ME-condition). Consider the rule schema `red2` (Figure 8). Let c denote the ME-constraint from Example 4.5.

After simplification, the transformation `Shift(red2,` c`)` results in the following ME-condition over the left-hand side of `red2`:

$$\begin{tikzcd}
 & P' \arrow[d] \arrow[ddl, "a'"'] \arrow[dddl, bend right=60, "b"'] & P \arrow[l, "p"'] \arrow[d, "a"] \arrow[dl, phantom, "(1)"] \\
 & C' \arrow[dl, dashed] & C \arrow[l] \arrow[dl, dashed] \\
(C')^{\sigma} \arrow[d, "e"] & C^{\sigma} \arrow[l, "q"'] \arrow[dl, "s"] & \\
E & &
\end{tikzcd}$$

Figure 16: Pushout construction for Shift’

$\exists_{\mathrm{L}}\, \mathtt{c}, \mathtt{d}.\exists$ $\hookrightarrow$ $.\mathtt{path}(v,w) \vee \mathtt{path}(w,v)$
$\vee\ \exists_{\mathrm{L}}\, \mathtt{d}.\exists$ $\hookrightarrow$ $.\mathtt{path}(2,w) \vee \mathtt{path}(w,2)$
$\vee\ \exists_{\mathrm{L}}\, \mathtt{c}.\exists$ $\hookrightarrow$ $.\mathtt{path}(v,2) \vee \mathtt{path}(2,v)$

□

The second intermediate transformation for ‘WPost’ is ‘Right’, which transforms a ME-condition over the left-hand side of a rule to a ME-condition over the right-hand side. This construction is based on transformation ‘L’ from [7, 12] but in the reverse direction.

The transformation is made more complex by the presence of path and MSO expressions, because nodes and edges referred to on the left-hand side may no longer exist on the right. For clarity, we separate the handling of these two types of expressions, and in particular, define a decomposition RPath of path predicates according to the items that the rule is deleting or creating. For example, if an edge is deleted by a rule, a path predicate decomposes to a disjunction of path predicates collectively asserting the existence of paths to and from the nodes that, in the pre-state, were incident to it. On the other hand, if an edge is to be created, the predicate will exclude it.

Lemma 4.14 (Path decomposition)**.** For every rule schema r, direct derivation $G \Rightarrow_{r,g,h} H$, path predicate γ, and assignment α,

$$\gamma^{g,\alpha} = \mathrm{RPath}(r,\gamma)^{h,\alpha}.$$

Construction. Suppose γ has the form $\mathtt{path}(v,w,\mathtt{not}\ E)$. If $v, w \in V_K$, then $\mathrm{RPath}(r,\gamma) = \mathtt{path}(v,w,\mathtt{not}\ E^{\ominus})$ where $E^{\ominus} = E \cup E_R \setminus E_L$. If either (or both) v

or w not in V_K, then RPath(r, γ) is defined as `false` in disjunction with:

$$\bigvee_{x,y \in V_K \cdot \text{path}_L(v,x,E) \wedge \text{path}_L(y,w,E)} \texttt{path(}x\texttt{,}y\texttt{,not } E^{\ominus}\texttt{)} \vee \text{DeletedPaths}(r, x, y)$$

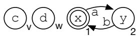
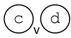
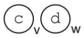
Above, DeletedPaths(r, x, w) is defined as `false` in disjunction with:

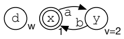
$$\bigvee_{\langle\langle a_1,b_1\rangle,\ldots\langle a_n,b_n\rangle\rangle} \Big(\texttt{path(}x\texttt{,}a_1\texttt{,not } E^{\ominus}\texttt{)} \wedge \ldots \texttt{path(}b_i\texttt{,}a_{i+1}\texttt{,not } E^{\ominus}\texttt{)} \wedge \ldots \texttt{path(}b_n\texttt{,}y\texttt{,not } E^{\ominus}\texttt{)}\Big)$$

where the non-empty sequences of distinct pairs are drawn from $\{(a, b) \mid a, b \in V_K \wedge \text{path}_L(a, b, E) \wedge \neg\text{path}_R(a, b, E^{\ominus})\}$.

If γ has the form $\texttt{path(}v\texttt{,}w\texttt{)}$, then RPath$(r, \gamma) = \text{RPath}(r, \texttt{path(}v\texttt{,}w\texttt{,not }\emptyset\texttt{))}$. □

Proof. By case analysis for each possible context of a match, following the proof structure of Proposition 1 in [25] (but in the reverse direction). □

Example 4.15 (Path decomposition). Consider the ME-condition $\texttt{Shift(red2,}c\texttt{)}$ from Example 4.13. Now, consider the derived rule schema $r_1^* =$  $\hookleftarrow$  $\hookrightarrow$ . Then, RPath$(r_1^*, \texttt{path(}v\texttt{,}w\texttt{)}) = \texttt{path(}v\texttt{,}w\texttt{)}$ and moreover, RPath$(r_1^*, \texttt{path(}w\texttt{,}v\texttt{)}) = \texttt{path(}w\texttt{,}v\texttt{)}$, since v, w are both in the interface of r_1^*.

Now consider the derived rule schema $r_2^* =$  $\hookleftarrow$ $\hookrightarrow$. In this case, RPath$(r_2^*, \texttt{path(2,}w\texttt{)})$ and RPath$(r_2^*, \texttt{path(}w\texttt{,2)})$ both evaluate to `false`, as there are no nodes in the interface that are connected by a path to node 2. □

In addition to paths, transformation Right must handle MSO expressions that refer to items present in L but absent in R. To achieve this, it computes a disjunction over all possible 'past' (i.e. immediately before the rule application) set memberships of these missing items. The idea is that if there are set memberships for deleted items in the post-state that satisfy the assignment constraints, then such a set membership would have existed in the pre-state before their deletion. The transformation keeps track of potential set memberships of deleted items via sets of pairs as follows.

Definition 4.16 (Membership set). A *membership set* M is a set of pairs $(x, \texttt{X})$ of node/edge identifiers x with set variables of the corresponding type. □

$$\begin{tikzcd}
\mathsf{L} \arrow[d, "\mathsf{a}"'] \arrow[dr, phantom, "(1)"] & \mathsf{K} \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & \mathsf{R}^{\oplus} \arrow[d, "\mathsf{b}"] \\
\mathsf{X} & \mathsf{Z} \arrow[l] \arrow[r] & \mathsf{Y}
\end{tikzcd}$$

Figure 17: Pushout construction for Right

Lemma 4.17 (Left to right ME-condition)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ denote a rule schema and c a ME-condition over L. Then for every direct derivation $G \Rightarrow_{r,g,h} H$ with $g\colon L^{g,\alpha} \hookrightarrow G$, $h\colon R^{g,\alpha} \hookrightarrow H$, and $\mathrm{dom}(\alpha) = \mathrm{vars}(L)$,

$$g\colon L^{g,\alpha} \hookrightarrow G \models^{\alpha} c \text{ if and only if } h\colon R^{g,\alpha} \hookrightarrow H \models^{\alpha} \mathrm{Right}(r,c).$$

Construction. Let $\mathrm{Right}(r,c) = \mathrm{Right}'(r,c,\emptyset)$. For such an r and a membership set M, $\mathrm{Right}'(r,\mathtt{true},M) = \mathtt{true}$ and $\mathrm{Right}'(r,\exists_{\mathtt{L}}\mathtt{x}.\ c,M) = \exists_{\mathtt{L}}\mathtt{x}.\ \mathrm{Right}'(r,c,M)$.

If γ is a path predicate, $\mathrm{Right}'(r,\gamma,M) = \mathrm{RPath}(r,\gamma)$. If γ has the form $x \in \mathtt{X}$, then $\mathrm{Right}'(r,\gamma,M) = \mathtt{true}$ if $(x,\mathtt{X}) \in M$, otherwise $\mathtt{false}$. Otherwise, if γ is a relation over list expressions, $\mathrm{Right}'(r,\gamma,M) = \gamma$, but with the following substitutions: (1) n for $\mathtt{indeg}(v)$ where $v \in V_L, v \notin V_R$, and n is the indegree of v in L; (2) $\mathtt{indeg}(v)-n$ for $\mathtt{indeg}(v)$ where $v \in V_K$ and n is the net change of edges from L to R with target v; (3–4) analogous cases for $\mathtt{outdeg}$; (5) $\mathtt{card}(X)+n$ for $\mathtt{card}(X)$ where n is the number of $(\_,\mathtt{X})$ pairs in M.

For ME-conditions quantifying set variables,

$$\mathrm{Right}'(r,\exists_{\mathtt{V}}\mathtt{X}.\ c,M) = \exists_{\mathtt{V}}\mathtt{X}.\ \bigvee_{M' \in 2^{M_{\mathtt{V}}}} \mathrm{Right}'(r,c,M \cup M')$$
$$\mathrm{Right}'(r,\exists_{\mathtt{E}}\mathtt{X}.\ c,M) = \exists_{\mathtt{E}}\mathtt{X}.\ \bigvee_{M' \in 2^{M_{\mathtt{E}}}} \mathrm{Right}'(r,c,M \cup M')$$

where $M_{\mathtt{V}} = \{(v,\mathtt{X}) \mid v \in V_L \setminus V_R\}$ and $M_{\mathtt{E}} = \{(e,\mathtt{X}) \mid e \in E_L \setminus E_R\}$.

Finally, $\mathrm{Right}'(r,\exists a.\ c,M) = \exists b.\ \mathrm{Right}'(r^*,c,M)$ if $\langle K \hookrightarrow L,a\rangle$ has a natural pushout complement (1) in Figure 17, where $R^{\oplus}$ denotes R with $\mathtt{in/outdeg}$ replaced as described earlier, and $r^* = \langle X \hookleftarrow Z \hookrightarrow Y\rangle$ denotes the rule 'derived' by also constructing natural pushout (2). If $\langle K \hookrightarrow L,a\rangle$ has no natural pushout complement, then $\mathrm{Right}'(r,\exists a.\ c,M) = \mathtt{false}$.

Right' is defined for Boolean formulae over ME-conditions as per usual. □

Proof. With the construction of Right and Lemma B.2, we have $g \models^{\alpha} c$ if and only if $h \models^{\alpha} \mathrm{Right}'(r,c,\emptyset)$ if and only if $h \models^{\alpha} \mathrm{Right}(r,c)$. □

Example 4.18 (Left to right ME-condition). Consider the rule schema `red2` (Figure 8) and the ME-condition Shift(`red2`, c) from Example 4.13.

After simplification, the transformation Right(`red2`, `Shift`(`red2`, c)) results in the following ME-condition over the right-hand side of `red2`:

$$\begin{aligned}
&\exists_{\mathrm{L}}\mathtt{c},\mathtt{d}.\exists\emptyset \hookrightarrow \textcircled{\scriptsize c}_{\mathrm{v}}\textcircled{\scriptsize d}_{\mathrm{w}}.\mathtt{path}(v,w) \vee \mathtt{path}(w,v)\\
&\vee\ \exists_{\mathrm{L}}\mathtt{d}.\exists\emptyset \hookrightarrow \textcircled{\scriptsize d}_{\mathrm{w}}.\mathtt{false} \vee \exists_{\mathrm{L}}\mathtt{c}.\exists\emptyset \hookrightarrow \textcircled{\scriptsize c}_{\mathrm{v}}.\mathtt{false}\\
&\equiv\ \exists_{\mathrm{L}}\mathtt{c},\mathtt{d}.\exists\emptyset \hookrightarrow \textcircled{\scriptsize c}_{\mathrm{v}}\textcircled{\scriptsize d}_{\mathrm{w}}.\mathtt{path}(v,w) \vee \mathtt{path}(w,v)
\end{aligned}$$

Observe how the disjuncts concerning overlaps of the rule and the original ME-constraint cannot be satisfied. This is because there cannot be a path to/from a node of the comatch to outside of it, since `red2` deletes every node/edge in the match. Hence, a path can only exist in the pre-state if it exists in the post-state *outside* of the comatch. □

Finally, we can give 'WPost' a simple definition based on the two intermediate transformations. Intuitively, it constructs a disjunction of ME-constraints that demand the existence of some co-match that can result from applying the rule schema set to a graph satisfying the presumption.

Proposition 4.19 (Weakest postcondition)**.** Let $\mathcal{R}$ denote a rule schemata set and c a ME-constraint. Then for every graph $H \in \mathcal{G}(\mathcal{L})$,

$$H \models \mathrm{WPost}(\mathcal{R}, c) \text{ if and only if } H \models \mathrm{WPOST}_{ok}[\mathcal{R}, c].$$

Construction. Define $\mathrm{WPost}(\emptyset, c) = \mathtt{false}$ and $\mathrm{WPost}(\mathcal{R}, c) = \bigvee_{r\in\mathcal{R}} \mathrm{wpost}(r, c)$. Let:

$$\mathrm{wpost}(r,c) = \exists_{t_1}\mathtt{x}_1.\cdots\exists_{t_n}\mathtt{x}_n.\exists\emptyset \hookrightarrow R^{\oplus}.\mathrm{Dang}(K \hookrightarrow R^{\oplus}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r,c) \wedge \tau(L,\Gamma))$$

where $R^{\oplus}$ is obtained from R by replacing `in`/`outdeg` expressions as in Lemma 4.17, $\{\mathtt{x}_1, \cdots, \mathtt{x}_n\} = \mathrm{vars}(R^{\oplus})$, and each t_i indicates the type of variable $\mathtt{x}_i$. □

Proof. ($\Longrightarrow$) Assume that $H \models \mathrm{WPost}(\mathcal{R}, c)$. There exists some $r \in \mathcal{R}$ such that:

$$H \models \exists_{t_1}\mathtt{x}_1.\cdots\exists_{t_n}\mathtt{x}_n.\exists\emptyset \hookrightarrow R^{\oplus}.\mathrm{Dang}(K \hookrightarrow R^{\oplus}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r,c) \wedge \tau(L,\Gamma)).$$

There exists an $h : (R^{\oplus})^{h,\alpha} \hookrightarrow H$ such that $h \models^{\alpha} \mathrm{Dang}(K \hookrightarrow R^{\oplus})$ and $h \models^{\alpha} \mathrm{Right}(r, \mathrm{Shift}(r,c) \wedge \tau(L,\Gamma))$. By Lemma 4.7, there exists a direct derivation from some graph H to G via a 'rule' $R^{\oplus} \Rightarrow L$, and thus some match $g : L^{g,\alpha} \hookrightarrow G$ from which H can be derived via a rule $L \Rightarrow R$ (note that $R^{g,\alpha} = (R^{\oplus})^{h,\alpha}$).

By Lemma 4.17, $g \models^{\alpha} \mathrm{Shift}(r, c) \wedge \tau(L, \Gamma)$. By Lemma 4.12, $G \models c$, and by Lemma 4.8, $\Gamma^{g,\alpha} = \mathrm{true}$. Together, there is a direct derivation $G \Rightarrow_r H$ for $r = \langle L \Rightarrow R, \Gamma\rangle$, i.e. $(G, H) \in [\![\mathcal{R}]\!]_{ok}$, and $H \models \mathrm{WPOST}_{ok}[\mathcal{R}, c]$.

($\Longleftarrow$) Assume that $H \models \mathrm{WPOST}_{ok}[\mathcal{R}, c]$. By Definition 3.5, there exists some $G.G \models c$ and $(G, H) \in [\![\mathcal{R}]\!]_{ok}$, i.e. there exists a conditional rule schema $r = \langle L \Rightarrow R, \Gamma\rangle$ in $\mathcal{R}$ such that $G \Rightarrow_r H$. There exists a match $g \colon L^{g,\alpha} \hookrightarrow G$ with $\Gamma^{g,\alpha} = \mathrm{true}$, and thus $g \models^{\alpha} \tau(L, \Gamma)$ by Lemma 4.8. By the definition of $\models$, Lemma 4.12, and Lemma 4.17, there exists some $h : R^{g,\alpha} \hookrightarrow G \models^{\alpha} \mathrm{Right}(r, \mathrm{Shift}(r, c) \wedge \tau(L, \Gamma))$. Using Lemma 4.7, $h \models^{\alpha} \mathrm{Dang}(K \hookrightarrow R)$. Observe that $R^{g,\alpha} = (R^{\oplus})^{h,\alpha}$. Together, we have:

$$H \models \exists_{t_1} \mathtt{x}_1. \cdots \exists_{t_n} \mathtt{x}_n. \exists \emptyset \hookrightarrow R^{\oplus}. \mathrm{Dang}(K \hookrightarrow R^{\oplus}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r, c) \wedge \tau(L, \Gamma))$$

where each $\exists_{t_i} \mathtt{x}_i$ quantifies a variable in $R^{\oplus}$ with the appropriate type t_i. We have $H \models \mathrm{wpost}(r, c)$, and being a disjunct of $\mathrm{WPost}(r, c)$, we get the result $H \models \mathrm{WPost}(r, c)$. □

Example 4.20 (Weakest postcondition). Consider the rule schema `red2` (Figure 8) and the ME-constraint c in Example 4.5. The weakest postcondition of `red2` relative to c, denoted WPost(`red2`, c), is simply:

$$\equiv \;\exists_{\mathrm{L}} \mathsf{c}, \mathsf{d}. \exists \textcircled{c}_{\mathrm{v}} \textcircled{d}_{\mathrm{w}}. \mathtt{path}(v, w) \vee \mathtt{path}(w, v)$$

(Observe that the inverse dangling condition is not an issue since the right-hand side of `red2` is $\emptyset$, and the condition of the rule schema is simply `true`.) □

4.4. Proof Rules with ME-Constraints

Having proposed an assertion language and used it to define (decidable fragments of) the extensional SE, FE, and WPOST assertions, we are now able to define an *intensional* version of the under-approximate logic with respect to that language. Figure 18 presents this, but for brevity, only displays axioms and proof rules that are different from the extensional versions (for example, the intensional version of Cons is the same as the extensional one in Figure 12).

Note that certain extensional proof rules cannot be ‘fully’ expressed with our assertion language. In particular, we cannot use ME-constraints to express the applicability of an arbitrary program. This is because programs can contain as-long-as-possible iteration which would require undecidable logics to reason about in general. Our intensional calculi restricts SE and FE to the applicability of rule

$$\textsc{RuleSetSucc}\ \vdash [c \wedge \mathrm{App}(\mathcal{R})]\ \mathcal{R}\ [ok\colon \mathrm{WPost}(\mathcal{R}, c)][fa\colon \mathtt{false}]$$

$$\textsc{RuleSetFail}\ \vdash [c \wedge \neg\mathrm{App}(\mathcal{R})]\ \mathcal{R}\ [ok\colon \mathtt{false}][fa\colon c \wedge \neg\mathrm{App}(\mathcal{R})]$$

$$\textsc{IfElse}\ \frac{\vdash [c \wedge \mathrm{App}(\mathcal{R})]\ P\ [\epsilon\colon d] \text{ or} \vdash [c \wedge \neg\mathrm{App}(\mathcal{R})]\ Q\ [\epsilon\colon d]}{\vdash [c]\ \mathtt{if}\ \mathcal{R}\ \mathtt{then}\ P\ \mathtt{else}\ Q\ [\epsilon : d]}$$

$$\textsc{TryElse}\ \frac{\vdash [c \wedge \mathrm{App}(\mathcal{R})]\ \mathcal{R};\ P\ [\epsilon\colon d] \text{ or} \vdash [c \wedge \neg\mathrm{App}(\mathcal{R})]\ Q\ [\epsilon\colon d]}{\vdash [c]\ \mathtt{try}\ \mathcal{R}\ \mathtt{then}\ P\ \mathtt{else}\ Q\ [\epsilon : d]}$$

$$\textsc{IterZero}\ \vdash [c \wedge \neg\mathrm{App}(\mathcal{R})]\ \mathcal{R}!\ [ok : c \wedge \neg\mathrm{App}(\mathcal{R})][fa : \mathtt{false}]$$

$$\textsc{IterStep}\ \frac{\vdash [c \wedge \mathrm{App}(\mathcal{R})]\ \mathcal{R};\mathcal{R}!\ [ok : d \wedge \neg\mathrm{App}(\mathcal{R})]}{\vdash [c \wedge \mathrm{App}(\mathcal{R})]\ \mathcal{R}!\ [ok : d \wedge \neg\mathrm{App}(\mathcal{R})]}$$

$$\textsc{IterVar}\ \frac{\vdash [c_{i-1}]\ \mathcal{R}\ [ok : c_i] \text{ for all } 0 < i \leq n, \text{ and } c_n \implies \neg\mathrm{App}(\mathcal{R})}{\vdash [c_0]\ \mathcal{R}!\ [ok : c_n]}$$

Figure 18: Proof rules with ME-constraints (rules with no changes from Figures 11–12 omitted)

schemata sets, but in general it would be possible to extend the transformation to a less restricted fragment (see e.g. [34]).

Note also that the IterVar rule is more restricted in that assertions are indexed rather than parameterised, as the latter would likely extend the expressive power of ME-conditions beyond monadic second-order logic on graphs.

Our proof rules for ME-constraints inherit the soundness result of the extensional proof rules presented earlier. This is due to the fact that the transformations App, ¬App, and WPost correctly implement the extensional assertions SE, FE, and WPOST for the fragments of the language (i.e. sets of conditional rule schemata) that they are defined for.

Theorem 4.21 (Soundness (for ME-constraints))**.** For all ME-constraints c, d, graph programs P, and exit statuses $\epsilon \in \{ok, fa\}$,

$$\vdash [c]\ P\ [\epsilon : d] \text{ implies } \models [c]\ P\ [\epsilon : d].$$

□

Proof. Immediate from the soundness of the extensional proof system (Theorem 3.8) and the results that App, ¬App, and WPost define SE, FE, and WPOST (Propositions 4.9, 4.10, and 4.19). □

While the extensional proof rules are relatively complete (Theorem 3.9), it remains an open problem whether or not our intensional proof rules are complete, even for programs that restrict iteration to sets of rule schemata. This is because the completeness proof depends on the *expressiveness* of the assertion language, i.e. the ability of ME-constraints to express the weakest postconditions of programs with respect to arbitrary presumptions. The expressiveness of ME-constraints, in this sense, is unknown [34].

5. Example Proofs

In this section, we demonstrate how our under-approximate logic with ME-constraints can be used to reason about incorrectness in some GP 2 programs for recognising cycle graphs.

Example 5.1 (Recognising cycle graphs I). Recall Example 2.13, which introduced cycle graphs and a graph program for recognising them. The program `is-cycle-buggy` in Figure 8 was meant to recognise whether an input graph is a cycle graph or not, by reducing cycle graphs to the empty graph.

To be more precise, call an unmarked and unrooted host graph an *input graph*. Then `is-cycle-buggy` is intended to be totally correct with respect to the following specifications:

Spec1	*Input:*	A cycle graph.
	Output:	The empty graph.
Spec2	*Input:*	An input graph that is not a cycle graph.
	Output:	A non-empty graph.

$$\dfrac{\dfrac{\vdash [\mathit{empty} \land \neg\mathrm{App}(\texttt{init})]\ \texttt{init}\ [fa : \mathit{empty} \land \neg\mathrm{App}(\texttt{init})]}{\vdash [\neg \mathit{cycle}]\ \texttt{init}\ [fa : \mathit{empty}]}}{\vdash [\neg\mathit{cycle}]\ \texttt{init};\texttt{Reduce}\ [fa : \mathit{empty}]}$$

Figure 19: Proving the presence of non-empty graphs (ME-constraints in Figure 20)

It is relatively straightforward to check that the program in Figure 8 reduces every cycle graph to the empty graph, and hence `is-cycle-buggy` is totally correct with respect to Spec1. (Moreover, one can prove that the reduction requires only linear time. We refer to [28] for a proof that the related program `is-cycle` of Example 5.2 has a linear time complexity.)

One can also show that `is-cycle-buggy` reduces input graphs that are not cycle graphs to non-empty graphs—provided that the input graphs are not empty. The program fails on the empty graph because `init` is not applicable, and thus `is-cycle-buggy` is not partially correct with respect to Spec2. We now verify this fact with the incorrectness calculus.

Figure 19 depicts a simple proof tree that shows it is possible for a non-cycle graph to result in the program failing on an empty graph (which is also the input). The assertions of the proof tree are given in Figure 20: note that we abbreviate $\texttt{indeg}(v)\texttt{=}\texttt{outdeg}(v) \land \texttt{outdeg}(v)\texttt{=1}$ to $\texttt{indeg}(v)\texttt{=}\texttt{outdeg}(v)\texttt{=1}$ in the ME-constraint *cycle*. Two side conditions $\mathit{empty} \Longrightarrow \mathit{empty} \land \neg\mathrm{App}(\texttt{init})$ and $\mathit{empty} \land \neg\mathrm{App}(\texttt{init}) \Longrightarrow \neg\mathit{cycle}$ clearly hold.

As this is an under-approximate proof, it soundly removes execution paths that are irrelevant to the post-state we want to prove the presence of. In particular, the proof focuses on failing executions of `init` that lead to a post-state violating Spec2. Instead of reasoning about every execution, we only reason about those that matter for this (un)desired post-state, quickly helping to *explain* why the triple is not provable under partial correctness. □

Example 5.2 (Recognising cycle graphs II). Consider the program `is-cycle` in Figure 21. As shown in [28], this program reduces cycle graphs to the empty graph and fails on non-cycle graphs. Hence `is-cycle` satisfies Spec1 and the specification Spec3 below.

Spec3	*Input:*	An input graph that is not a cycle graph.
	Output:	Failure.

$$
\begin{aligned}
\mathit{cycle} &= \forall_{\mathrm{L}}\mathsf{a}.\forall \textcircled{a}_{v}.\mathtt{indeg}(v)\mathtt{=outdeg}(v)\mathtt{=1} \\
&\quad \wedge \forall_{\mathrm{L}}\mathsf{a},\mathsf{b}.\forall \textcircled{a}_{v}\textcircled{b}_{w}.\mathtt{path}(v,w) \vee \mathtt{path}(w,v) \\
&\quad \wedge \exists_{\mathrm{L}}\mathsf{a}.\exists \textcircled{a} \wedge \neg\exists_{\mathrm{L}}\mathsf{a}.\exists \textcircled{\textcircled{a}} \\
\mathit{empty} &= \neg\exists_{\mathrm{L}}\mathsf{a}.\exists \textcircled{a} \wedge \neg\exists_{\mathrm{L}}\mathsf{a}.\exists \textcircled{\textcircled{a}} \\
\mathrm{App}(\mathtt{init}) &= \exists_{\mathrm{L}}\mathsf{x}.\exists \textcircled{x}
\end{aligned}
$$

Figure 20: ME-constraints used in the proof in Figure 19

```
Main = init; Reduce; if match then fail
Reduce = red3!; {red2, red1}
```

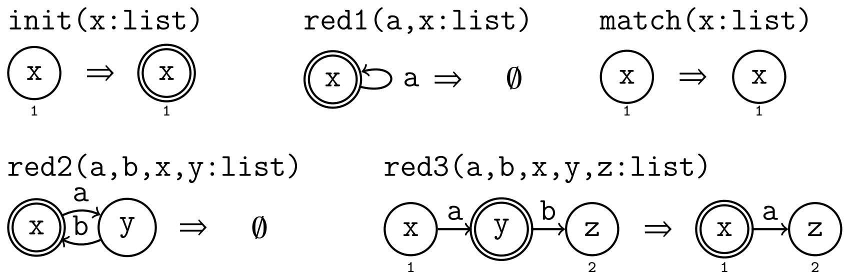


Figure 21: GP 2 program `is-cycle`

Suppose that a programmer makes the mistake to formalise cycle graphs as non-empty host graphs in which each node has exactly one incoming and one outgoing edge. Let us call such graphs *multi-cycle graphs* because they can be seen as multisets of cycle graphs. The specifications Spec4 and Spec5 below are obtained from Spec1 and Spec3 by replacing cycle graphs with multi-cycle graphs.

Spec4	*Input:*	A multi-cycle graph.
	Output:	The empty graph.
Spec5	*Input:*	An input graph that is not a multi-cycle graph.
	Output:	Failure.

Program `is-cycle` satisfies Spec5 because Spec5 is obtained from Spec3 by weakening the precondition (every non-multi-cycle graph is also a non-cycle graph). However, `is-cycle` violates Spec4 because it fails on the multi-cycle graph of Figure 22. This is because `red1` deletes only one of the nodes and its loop, after which `match` triggers failure.

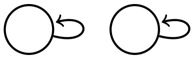


Figure 22: A multi-cycle graph showing that `is-cycle` violates Spec4

We now use our calculus to verify the incorrectness of `is-cycle` with respect to Spec4. The proof tree is given in Figure 23, and the ME-constraints utilised are given in Figure 24. Note that, for space constraints, the results of the WPost transformations are simplified to equivalent ME-constraints. Furthermore, we soundly omit a disjunct of WPost({`red2`, `red1`}, e) that is not relevant to the proof—a hallmark of under-approximate reasoning. □

$$\dfrac{\dfrac{\dfrac{\vdash [c \land \mathrm{App}(\mathtt{init})]\ \mathtt{init}\ [ok : \mathrm{WPost}(\mathtt{init}, c)]}{\vdash [c]\ \mathtt{init}\ [ok : e]} \qquad \dfrac{\textit{(see sub-tree below)}}{\vdash [e]\ \mathtt{Reduce}\ [ok : d]}}{\vdash [c]\ \mathtt{init};\ \mathtt{Reduce}\ [ok : d]} \qquad \dfrac{\dfrac{\vdash [d]\ \mathtt{fail}\ [fa : d]}{\vdash [d \land \mathrm{App}(\mathtt{match})]\ \mathtt{fail}\ [fa : d]}}{\vdash [d]\ \mathtt{if}\ \mathtt{match}\ \mathtt{then}\ \mathtt{fail}\ [fa : d]}}{\vdash [c]\ \mathtt{init}; \mathtt{Reduce};\ \mathtt{if}\ \mathtt{match}\ \mathtt{then}\ \mathtt{fail}\ [fa : d]}$$

$$\dfrac{\dfrac{\vdash [e \land \neg\mathrm{App}(\mathtt{red3})]\ \mathtt{red3}!\ [ok : e \land \neg\mathrm{App}(\mathtt{red3})]}{\vdash [e]\ \mathtt{red3}!\ [ok : e]} \qquad \dfrac{\vdash [e \land \mathrm{App}(\{\mathtt{red2}, \mathtt{red1}\})]\ \{\mathtt{red2}, \mathtt{red1}\}\ [ok : \mathrm{WPost}(\{\mathtt{red2}, \mathtt{red1}\}, e)}{\vdash [e]\ \{\mathtt{red2}, \mathtt{red1}\}\ [ok : d]}}{\vdash [e]\ \mathtt{Reduce}\ [ok : d]}$$

Figure 23: Proving the presence of failure (ME-constraints in Figure 24)

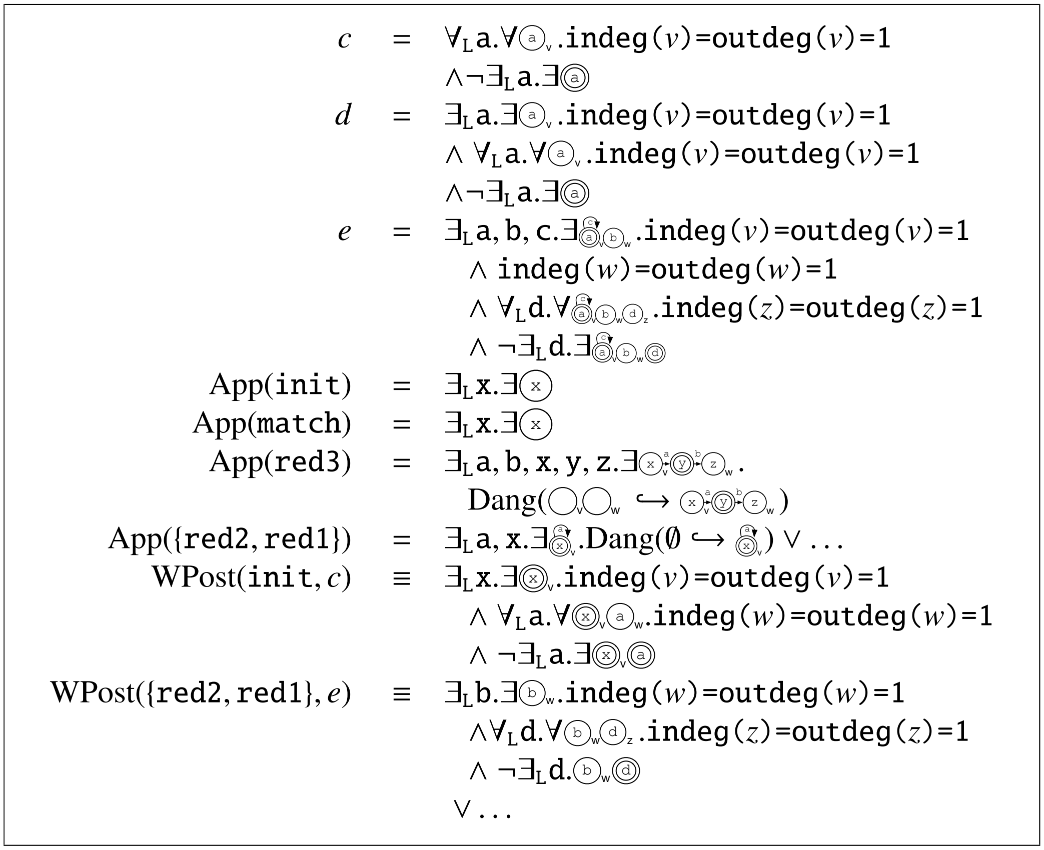

c =
∀La.∀ⓐv.indeg(v)=outdeg(v)=1
∧¬∃La.∃ⓐ
d =
∃La.∃ⓐv.indeg(v)=outdeg(v)=1
∧ ∀La.∀ⓐv.indeg(v)=outdeg(v)=1
∧¬∃La.∃ⓐ
e =
∃La, b, c.∃.indeg(v)=outdeg(v)=1
∧ indeg(w)=outdeg(w)=1
∧ ∀Ld.∀.indeg(z)=outdeg(z)=1
∧ ¬∃Ld.∃
App(init) =
∃Lx.∃ⓧ
App(match) =
∃Lx.∃ⓧ
App(red3) =
∃La, b, x, y, z.∃.
Dang(
App({red2, red1}) =
∃La, x.∃.Dang(∅ ↪) ∨ . . .
WPost(init, c) ≡
∃Lx.∃ⓧv.indeg(v)=outdeg(v)=1
∧ ∀La.∀.indeg(w)=outdeg(w)=1
∧ ¬∃La.∃
WPost({red2, red1}, e) ≡
∃Lb.∃ⓑw.indeg(w)=outdeg(w)=1
∧∀Ld.∀.indeg(z)=outdeg(z)=1
∧ ¬∃Ld.
∨ . . .

Figure 24: ME-constraints used in the proof in Figure 23

6. Related Work

In this section, we highlight some related work in two key areas: under-approximate reasoning and monadic second-order reasoning on graphs.

6.1. Under-Approximate Reasoning

Over-approximate program logics for proving the absence of bugs have been studied extensively [32]. Our program logic differs by focusing on under-approximate reasoning, i.e. proofs about the presence of bugs (in our case, forbidden graph structure or failing execution paths). The first under-approximate calculus of this kind was introduced by De Vries and Koutavas [14], who proposed the notion of under-approximate validity, and defined a ‘Reverse Hoare Logic’ for proving reachability specifications over the proper states of imperative randomised programs. O'Hearn's incorrectness logic [15] extended this program logic to support under-approximate reasoning about executions that result in errors, an idea we adopt to support reasoning about both successful computations (*ok*) and failing executions (fa). Both of these program logics use variants to reason about while-loop termination, but unlike standard Hoare logics, require that the variant decreases in the backwards direction. Our IterVar rule is similar, but requires the number of iterations to be known as ME-constraints may not be expressive enough to specify parameterised graph properties.

Raad et al. [16] combined separation logic with incorrectness logic to facilitate proofs about the presence of bugs using local reasoning, i.e. specifications that focus only on the region of memory being accessed. They found that the original model of separation logic, which does not distinguish dangling pointers from pointers we have no knowledge about, to be incompatible with the under-approximate frame rule. This was resolved by refining the model with negative heap assertions that can specify that a location has been de-allocated.

Murray [18] proposed the first under-approximate relational logic, allowing for reasoning about the behaviours of pairs of programs. As many important security properties (e.g. noninterference, function sensitivity, refinement) can be specified as relational properties, Murray's program logic can be used to provably demonstrate the presence of insecurity.

Bruni et al. [36] incorporate incorrectness logic in a proof system for abstract interpretation that combines over- and under-approximation. Given an abstraction that is 'locally complete' (i.e. complete only for some specific inputs, rather than all possible inputs), they show that it is possible to prove both the presence as well as the absence of true alerts.

Incorrectness logics allow formal reasoning about reachability specifications—in our context, the presence of failure or forbidden graph structure. A complementary approach is to find counterexamples (i.e. instances of the forbidden structure) using model checkers such as Groove [2]. Analysing graph transformation systems can be challenging, however, as they often have infinite state spaces, but this can be mitigated by using bounded model checking [37].

6.2. Monadic Second-Order Reasoning on Graphs

Habel and Radke proposed HR* conditions [38], an extension of nested conditions that embed hyperedge replacement grammars via graph variables. The formalism is more expressive than monadic second-order logic on graphs, lying somewhere between counting monadic second-order logic and second-order logic [39]. HR* conditions can be ‘plugged in’ to our extensional proof rules so long as transformations defining SE, FE, and WPOST can be provided.

Percebois et al. [40] demonstrate how one can verify global invariants involving paths (similar to our path predicates), directly at the level of rules. Rules are modelled with (a fragment of) first-order logic on graphs in the interactive theorem prover Isabelle.

Inaba et al. [41] address the verification of type-annotated Core UnCAL—a query algebra for graph-structured databases—against input/output graph schemas in MSO. They first reformulate the query algebra itself in MSO, before applying an algorithm that reduces the verification problem to the validity of MSO over trees.

Navarro et al. [42] developed a sound and complete deductive tableau method, unifying a line of several works (e.g. [43, 44, 45]). Their method targets graph navigational logic (GNL), which is also visual, and supports navigational properties, i.e. properties about the paths in a graph. Although ME-constraints are more expressive, it would be interesting to instantiate our extensional calculi using GNL so as to have a deductive system that also covers the side conditions of our proof rules, e.g. implications such as $c \Longleftarrow c'$.

Wulandari and Plump [34] propose a standard (i.e. without morphisms) monadic second-order logic with counting for graphs, and construct strongest liberal postconditions to prove the total correctness of GP 2 programs (including nested loops). Our extensional proof rules could potentially be instantiated using their logic as the assertion language.

7. Conclusion

We proposed an incorrectness logic for under-approximate reasoning about GP 2, demonstrating that the deductive rules of Hoare logics can be ‘reversed’ to prove the presence of graph transformation bugs, such as the possibility of illegal graph substructures or failing execution paths. In particular, we presented a calculus of incorrectness axioms and rules, proved them to be sound and relatively complete with respect to the semantics of GP 2, and demonstrated their use to prove the presence of various bugs in faulty programs for detecting (multi-)cycle graphs.

This paper was principally a theoretical exposition, but was motivated by some potentially interesting applications. One idea (suggested by O’Hearn [15]) is to recast static bug catchers in terms of finding under-approximation proofs. For instance, incorrectness logic might be able to provide soundness arguments for approaches that symbolically execute graph or model transformations (e.g. [21, 22, 23]). Another idea is to use it to complement over-approximate proofs: if one is unable to prove a partial correctness specification or the absence of failure [46], switch to under-approximate proofs instead and reason about the circumstances that could cause some undesirable result to be reachable. Our examples in Section 5 explored this idea in a preliminary way, proving triples in our under-approximate program logic that explain why some related partial correctness specifications cannot be proven. These applications would benefit from some automation in the construction of incorrectness proofs, which may warrant the exploration of an assertion language equipped with deductive systems, e.g. graph navigational logic [42].

Beyond exploring these potential applications, future work should also extend our logic to a larger class of programs. For example, programs with nested loops, which occur frequently in realistic graph programs (e.g. [28, 34]), or the graph programs of other languages, such as the recipes of Groove [47, 2]). It is also important to investigate how to make incorrectness reasoning for graph programs easier. This could be in the form of guidelines on how to come up with incorrectness specifications, or some derived proof rules for simplifying reasoning about common patterns. Finally, it would be interesting to model more of the GP 2 runtime so as to allow proofs with respect to improper exit statuses, such as division by zero, or perhaps even non-termination.

Appendix A. Pushouts and Pullbacks

This appendix contains some key definitions and well-known results about pushouts and pullbacks, which are used in some of the constructions and proofs in the main part of the paper. Further results in the context of classic double-pushout graph transformation and double-pushout graph transformation with relabelling can be found respectively in [48] and [30]. Readers interested in the category-theoretic background can consult a number of books, e.g. [49, 50, 51].

Note that we use the definitions of graphs and graph morphisms as presented in Section 2, which permit nodes (but not edges) to be unlabelled. Note also that we assume all morphisms to preserve and reflect rooted nodes, which ensures that all of the following categorical properties (e.g. uniqueness of direct derivations) apply in our setting [29].

Pushouts are an abstract gluing construction for two graph morphisms with a common domain. Informally, a pushout graph is formed from the disjoint union

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[d] \\
C \arrow[r] & D
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & B \arrow[d] \arrow[ddr] & \\
C \arrow[r] \arrow[drr] & D \arrow[dr] \arrow[ur, phantom, "="] \arrow[d, phantom, "="] & \\
 & & D'
\end{tikzcd}$$

Figure A.25: A pushout (left) and the universal property of pushouts (right)

of the codomains of the morphisms, but with nodes and edges identified when they are also present in the common domain.

Definition A.1 (Pushout). Given two graph morphisms $A \rightarrow B$ and $A \rightarrow C$, a graph D together with two graph morphisms $B \rightarrow D$ and $C \rightarrow D$ form a *pushout (PO)* of $A \rightarrow B$ and $A \rightarrow C$, depicted as (1) in Figure A.25, if the following properties are satisfied:

Commutativity. $A \rightarrow B \rightarrow D = A \rightarrow C \rightarrow D$.

Universal Property. For all graphs D' and graph morphisms $B \rightarrow D'$ and $C \rightarrow D'$ such that $A \rightarrow B \rightarrow D' = A \rightarrow C \rightarrow D'$, there is a unique graph morphism $D \rightarrow D'$ such that $B \rightarrow D \rightarrow D' = B \rightarrow D'$ and $C \rightarrow D \rightarrow D' = C \rightarrow D'$ (see Figure A.25). □

Definition A.2 (Pushout complement). Given two graph morphisms $A \rightarrow B$ and $B \rightarrow D$, a graph C together with two graph morphisms $A \rightarrow C$ and $C \rightarrow D$ is a *pushout complement* of $A \rightarrow B$ and $B \rightarrow D$ if the resulting diagram (i.e. (1) in Figure A.25) is a pushout. □

The dual construction of a pushout is a pullback, which can be seen as a generalisation of intersection over a common graph.

Definition A.3 (Pullback). Given two graph morphisms $B \rightarrow D$ and $C \rightarrow D$, a graph A together with two graph morphisms $A \rightarrow B$ and $A \rightarrow C$ form a *pullback (PB)* of $B \rightarrow D$ and $C \rightarrow D$, depicted as (1) in Figure A.26, if the following properties are satisfied:

Commutativity. $A \rightarrow B \rightarrow D = A \rightarrow C \rightarrow D$.

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[d] \\
C \arrow[r] & D
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
A' \arrow[dr] \arrow[drr, "="'] \arrow[ddr, "="] & & \\
& A \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & B \arrow[d] \\
& C \arrow[r] & D
\end{tikzcd}$$

Figure A.26: A pullback (left) and the universal property of pullbacks (right)

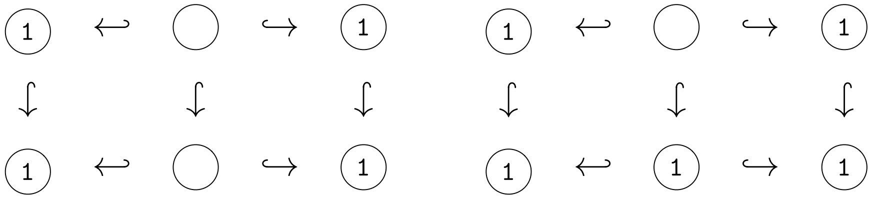


Figure A.27: Natural (left) and non-natural (right) pushouts (example from [30])

Universal Property. For all graphs A' and graph morphisms $A' \to B$ and $A' \to C$ such that $A' \to B \to D = A' \to C \to D$, there is a unique graph morphism $A' \to A$ such that $A' \to A \to B = A' \to B$ and $A' \to A \to C = A' \to C$ (see Figure A.26). □

Natural pushouts are pushouts that are simultaneously pullbacks.

Definition A.4 (Natural pushout). A pushout (1) as in Figure A.25 is a *natural pushout* if it is simultaneously a pullback. □

Natural pushouts can be characterised in terms of ordinary pushouts [30]: if $A \to B$ is injective, then (1) is natural if and only if every unlabelled node in A is mapped to an unlabelled node in B or C. See, for example, the natural and non-natural pushouts in Figure A.27.

Lemma A.5 (Existence of pullbacks)**.** Given two graph morphisms $B \to D$ and $C \to D$, there exists a graph A and two graph morphisms $A \to B$ and $A \to C$ such that the resulting diagram is a pullback. □

Proof. By Lemma 1 of [30]. □

The following lemmata describe some conditions that guarantee a pushout or pushout complement to exist, given two morphisms over graphs with partially labelled nodes. For simplicity, we give more restricted versions of the lemmata than the ones in the cited paper. This is because we only invoke them in the context of GP 2 rules, which are restricted to injective morphisms, as well as totally labelled left and right graphs.

Lemma A.6 (Existence of pushouts)**.** Given an injective graph morphism $A \hookrightarrow B$ with B a totally labelled graph, and an injective graph morphism $A \hookrightarrow C$ that preserves undefinedness, there exists a graph D and two graph morphisms $B \hookrightarrow D$ and $C \hookrightarrow D$ such that the resulting diagram is a pushout. □

Proof. By Lemma 2 of [30]. □

Lemma A.7 (Existence, uniqueness of natural PO complements)**.** Given an injective graph morphism $B \hookrightarrow D$ and inclusion $A \hookrightarrow B$ with B and D totally labelled graphs, there exists a graph C with two injective graph morphisms $A \hookrightarrow C$ and $C \hookrightarrow D$ such that the resulting diagram is a natural pushout if and only if $B \hookrightarrow D$ satisfies the dangling condition with respect to $A \hookrightarrow B$. Moreover, in this case, graph C is unique up to isomorphism. □

Proof. By Lemma 4 of [30]. □

The following lemmata are particularly useful in the correctness proofs for the various transformations of ME-conditions (see Section 4.3).

Lemma A.8 (Basic (de)compositions)**.** The following (de)compositions relate to the commutative diagram in Figure A.28, in which all the graph morphisms are injective.

Pushout/pullback composition. If (1) and (2) are pushouts (resp. pullbacks) then the composite diagram (1)+(2) is a pushout (resp. pullback).

Pushout decomposition. If (1)+(2) and (1) are pushouts, then (2) is also a pushout.

Pullback decomposition. If (1)+(2) and (2) are pullbacks, then (1) is also a pullback. □

Proof. Diagram chase (see e.g. [50]). □

$$\begin{tikzcd}
A \arrow[r, hook] \arrow[d, hook] \arrow[dd, hook, bend right=40, "="'] \arrow[dr, phantom, "(1)"] & B \arrow[d, hook] \arrow[dd, hook, bend left=40, "="] \\
C \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(2)"] & D \arrow[d, hook] \\
E \arrow[r, hook] & F
\end{tikzcd}$$

Figure A.28: Commutative diagram of injective graph morphisms

The following decompositions are more specialised to our context, in that they consider whether morphisms preserve undefinedness and also the naturalness of pushouts.

Lemma A.9 (Special decompositions)**.** The following decompositions relate to the commutative diagram in Figure A.28, in which all the graph morphisms are injective, and B, D, F are totally labelled graphs.

Pushout-pullback decomposition. If (1)+(2) is a pushout, $A \to C \to E$ is undefinedness preserving, and (2) is a pullback, then (1) and (2) are natural pushouts.

Pullback decomposition. If (1)+(2) and (1) are pushouts and $A \to C \to E$ is undefinedness preserving, then (1) and (2) are pullbacks. □

Proof. Habel and Plump [52] prove that the category of partially labelled graphs and their morphisms form a so-called $\mathcal{M}, \mathcal{N}$-adhesive category. The decompositions are properties of $\mathcal{M}, \mathcal{N}$-adhesive categories (see Theorem 1 in [52][2]). □

B. Proofs for Weakest Precondition Lemmata

The proof of Lemma 4.12 relies on the following more general lemma, stating that a ME-condition can be shifted along an injective morphism. The proof approach is based on that of transformation A from [7].

Lemma B.1 (Shifting ME-conditions)**.** Let $p : P \hookrightarrow P'$ and $p'' : (P')^{p'',\alpha} \hookrightarrow G$ denote injective morphisms with $P, P' \in \mathcal{G}(\mathrm{RG})$, $G \in \mathcal{G}(\mathcal{L})$, and $\mathrm{dom}(\alpha) = \mathrm{vars}(P')$. Let c denote a ME-condition over P labelled over lists of variables. Then,

[2] The second decomposition is given only in the long version of the authors' paper.

$$p'' \models^{\alpha} \mathrm{Shift}'(p, c) \text{ if and only if } p'' \circ p^{p'',\alpha} \models^{\alpha} c.$$

□

Proof. (*Induction basis*) Suppose c has the form $\mathtt{true}$. Then $p'' \models^{\alpha} \mathrm{Shift}'(p, c) = \mathtt{true}$ and $p'' \circ p^{p'',\alpha} \models^{\alpha} \mathtt{true}$.

Suppose c is an assignment constraint γ. Then $p'' \models^{\alpha} \mathrm{Shift}'(p, c) = \gamma$ if and only if $\gamma^{p'',\alpha} = \text{true}$ if and only if $p'' \circ p^{p'',\alpha} \models^{\alpha} \gamma$.

(*Induction step*; $\Longrightarrow$) Assume that $p'' \models^{\alpha} \mathrm{Shift}'(p, c)$.

Suppose c has the form $\exists_{\mathtt{L}}\mathtt{x}.c'$. Then $p'' \models^{\alpha} \exists_{\mathtt{L}}\mathtt{x}.\mathrm{Shift}'(p, c')$. By definition of $\models$, there exists some $l \in \mathbb{L}$ such that $p'' \models^{\alpha[\mathtt{x}\mapsto l]} \mathrm{Shift}'(p, c')$ By induction hypothesis, $p'' \circ p^{p'',\alpha} \models^{\alpha[\mathtt{x}\mapsto l]} c'$. By definition of $\models$, we get the result that $p'' \circ p^{p'',\alpha} \models^{\alpha} \exists_{\mathtt{L}}\mathtt{x}.c'$ and thus $p'' \circ p^{p'',\alpha} \models^{\alpha} c$. (Analogous for case $\exists_{\mathtt{V}}$ and $\exists_{\mathtt{E}}$.)

Finally, suppose that c has the form $\exists a\colon P \hookrightarrow C.c'$. There exists some $\sigma \in \Sigma$ and $e \in \varepsilon_{\sigma}$ such that:

$$p''\colon (P')^{p'',\alpha} \hookrightarrow G \models^{\alpha} \exists b\colon P' \hookrightarrow E.\ \mathrm{Shift}'(s\colon C^{\sigma} \hookrightarrow E, c'^{\sigma})$$

There exists a morphism $q'' : E^{q'',\alpha} \hookrightarrow G$ with $p'' = q'' \circ b^{q'',\alpha}$. Define $q' = q'' \circ s^{q'',\alpha}$ and $p' = p'' \circ p^{p'',\alpha}$. By construction and the definition of list expression evaluation, $(a')^{q'',\alpha} \circ p^{p'',\alpha} = q^{q'',\alpha} \circ a^{q'',\alpha}$ is a pushout (1) (Figure B.29), $s^{q'',\alpha} = e^{q'',\alpha} \circ q^{q'',\alpha}$, and $b^{q'',\alpha} = e^{q'',\alpha} \circ (a')^{q'',\alpha}$. Together, $p'' \circ p^{p'',\alpha} = p' = q' \circ a^{q'',\alpha}$ and $p'' \circ p^{p'',\alpha} \models^{\alpha} \exists a$. By assumption, $q'' \models^{\alpha} \mathrm{Shift}'(s : C^{\sigma} \hookrightarrow E, c'^{\sigma})$. By induction hypothesis, $q' = q'' \circ s^{q'',\alpha} \models^{\alpha} c'^{\sigma}$. By the assumption and definition of $\models$, we get the result that $p'' \circ p^{p'',\alpha} \models^{\alpha} \exists a.c'$.

For Boolean formulae over ME-conditions, the result is obtained from the definitions and induction hypothesis.

(*Induction step*; $\Longleftarrow$) Assume that $p'' \circ p^{p'',\alpha} \models^{\alpha} c$. Define $p' = p'' \circ p^{p'',\alpha}$.

Suppose c has the form $\exists_{\mathtt{L}}\mathtt{x}.c'$. By assumption and definition of $\models$, for some $l \in \mathbb{L}$, $p' \models^{\alpha[\mathtt{x}\mapsto l]} c'$. By induction hypothesis, $p'' \models^{\alpha[\mathtt{x}\mapsto l]} \mathrm{Shift}'(p, c')$. By definition of $\models$, we get $p'' \models^{\alpha} \exists_{\mathtt{L}}\mathtt{x}.\mathrm{Shift}'(p, c')$ and thus the result that $p'' \models^{\alpha} \mathrm{Shift}'(p, c)$. (Analogous for case $\exists_{\mathtt{V}}$ and $\exists_{\mathtt{E}}$.)

Finally, suppose that c has the form $\exists a : P \hookrightarrow C.c'$. By assumption, there exists a morphism $q' : C^{q',\alpha} \hookrightarrow G$ such that $q' \circ a^{q',\alpha} = p'$. Let σ denote a substitution from Σ for which $\alpha(\mathtt{x}_i) = e_i^{q',\alpha}$ for each $\mathtt{x}_i \mapsto e_i$. (Given that $() \in \Sigma$, such a substitution will always exist.) From the construction and definition of list evaluation, we obtain pushout graph $(C'^{\sigma})^{q',\alpha}$ along with the morphisms $(a')^{q',\alpha} : (P')^{p'',\alpha} \hookrightarrow (C'^{\sigma})^{q',\alpha}$ and $q^{q',\alpha} : (C^{\sigma})^{q',\alpha} \hookrightarrow (C'^{\sigma})^{q',\alpha}$. As $q' \circ a^{q',\alpha} = p' = p'' \circ p^{p'',\alpha}$,

$$\begin{tikzcd}
(P')^{p'',\alpha} \arrow[d, "(a')^{q'',\alpha}"'] \arrow[dd, bend right, "b^{q'',\alpha}"'] \arrow[ddd, bend right=60, "p''"'] & P^{p'',\alpha} \arrow[l, "p^{p'',\alpha}"'] \arrow[d, "a^{q'',\alpha}"] \arrow[dddl, bend left=50, "p'"] \arrow[dl, phantom, "(1)"] \\
(C'^{\sigma})^{q'',\alpha} \arrow[d, "e^{q'',\alpha}"] \arrow[dd, bend right, "g"'] & (C^{\sigma})^{q'',\alpha} \arrow[l, "q^{q'',\alpha}"'] \arrow[dl, "s^{q'',\alpha}"] \arrow[ddl, "q'"] \\
E^{q'',\alpha} \arrow[d, "q''"] & \\
G &
\end{tikzcd}$$

Figure B.29: Instantiating the construction with assignments

the pushout guarantees the existence of a unique morphism $g\colon (C'^{\sigma})^{q',\alpha} \to G$ with $p'' = g \circ (a')^{q',\alpha}$ and $q' = g \circ q^{q',\alpha}$. Consider $y \circ x = g$, a surjective-injective factorisation of g with $x\colon (C'^{\sigma})^{q',\alpha} \to X$ surjective and $y\colon X \hookrightarrow G$ injective. There is a surjective morphism $e : (C')^{\sigma} \to E$ such that $E^{q'',\alpha} = X$, as the substitution ensures list expressions equal under α can be merged at the syntactic level. Thus, there is a morphism $q'' : E^{q'',\alpha} \hookrightarrow G$ such that $q'' = y$ and $(C'^{\sigma})^{q',\alpha} = (C'^{\sigma})^{q'',\alpha}$. Define $s^{q'',\alpha} = e^{q'',\alpha} \circ q^{q'',\alpha}$ and $b^{q'',\alpha} = e^{q'',\alpha} = (a')^{q'',\alpha}$. Together, $p'' = g \circ (a')^{q'',\alpha}$, $g = q'' \circ e^{q'',\alpha}$, and $b = e^{q'',\alpha} \circ (a')^{q'',\alpha}$ yield $p'' = q'' \circ b^{q'',\alpha}$, i.e. $p'' \models^{\alpha} \bigvee_{e \in \mathcal{E}_{\sigma}} \exists b\colon P' \hookrightarrow E$. By assumption ($q' = q'' \circ s^{q'',\alpha} \models^{\alpha} (c')^{\sigma}$) and induction hypothesis, we have $q'' \models^{\alpha} \mathrm{Shift}'(s, (c')^{\sigma})$. Together, we get $p'' \models^{\alpha} \bigvee_{\sigma \in \Sigma} \bigvee_{e \in \mathcal{E}_{\sigma}} \exists b : P' \hookrightarrow E.\mathrm{Shift}'(s, (c')^{\sigma})$, and finally the result, $p'' \models^{\alpha} \mathrm{Shift}'(p, c)$.

For Boolean formulae over ME-conditions, the result is obtained from the definitions and induction hypothesis. □

The proof of Lemma 4.17 relies on a more general lemma about the Right'. The proof approach is based on that of transformation L' from [25].

Lemma B.2 (Transformation Right')**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ denote a rule schema and c a ME-condition over L. Then, for every direct derivation $G \Rightarrow_{r,g,h} H$ with $g : L^{g,\alpha} \hookrightarrow G$, $h\colon R^{g,\alpha} \hookrightarrow H$, $\mathrm{dom}(\alpha) = \mathrm{FV}(c)$, and membership set M for items in $L \setminus R$,

$$g\colon L^{g,\alpha_M} \hookrightarrow G \models^{\alpha_M} c \text{ if and only if } h\colon R^{g,\alpha} \hookrightarrow H \models^{\alpha} \mathrm{Right}'(r, c, M).$$

Here, α_M is defined as α except for items $x \in L \setminus R$, where $g(x) \in \alpha_M(\mathtt{X})$ if and only if $(x, \mathtt{X}) \in M$. □

Proof. (*Induction basis*) Suppose c has the form $\mathtt{true}$. Then $g \models^{\alpha_M} c$ if and only if $h \models^{\alpha} \mathrm{Right}'(r, c, M) = \mathtt{true}$.

Suppose c is an assignment constraint γ. If γ is a path predicate, then the result follows from Lemma 4.14. If γ has the form $x \in \mathtt{X}$, then $g \models^{\alpha_M} c$ if and only if γ^{g,α_M} if and only if $(x, \mathtt{X}) \in M$ if and only if $h \models^{\alpha} \mathrm{Right}'(r, c, M)$. If γ is a (relation over) a list expression, then the statement is clear other than for expressions e interpreted with respect to a morphism (i.e. `indeg`, `outdeg`, and `card`). This can be proven by showing that e^{g,α_M} evaluates to the same integer after applying h, α to the substituted expression (clear by case analysis).

(*Induction step;* $\Longrightarrow$) Suppose c has the form $\exists_{\mathrm{L}}\mathtt{x}.c'$. By assumption, $g \models^{\alpha_M[\mathtt{x}\mapsto l]} c'$ for some $l \in \mathtt{L}$. By induction hypothesis, $h \models^{\alpha[\mathtt{x}\mapsto l]} \mathrm{Right}'(r, c', M)$. By the definition of $\models$, we get the result that $h \models^{\alpha} \exists_{\mathrm{L}}\mathtt{x}.\mathrm{Right}'(r, c', M)$.

Suppose c has the form $\exists_{\mathrm{V}}\mathtt{X}.c'$. By the definition of $\models$, there exists some $V \subseteq V_G$ such that $g \models^{\alpha_M[\mathtt{X}\mapsto V]} c'$. Define $M' = \{(v, \mathtt{X}) \mid v \in V_L \setminus V_R \wedge g(v) \in V\}$ and $V^{\ominus} = V \setminus (V_G \setminus V_D)$. Define $\alpha' = \alpha[\mathtt{X} \mapsto V^{\ominus}]$. We have $\alpha'_{M\cup M'} = \alpha_M[\mathtt{X} \mapsto V^{\ominus} \cup \{g(v) \mid (v, \mathtt{X}) \in M'\}]$. Observe that $g \models^{\alpha'_{M\cup M'}} c'$. By induction hypothesis, $h \models^{\alpha'} \mathrm{Right}'(r, c', M \cup M')$. Clearly, $M' \in 2^{M_{\mathrm{V}}}$, and so $h \models^{\alpha'} \bigvee_{M' \in 2^{M_{\mathrm{V}}}} \mathrm{Right}'(r, c', M \cup M')$. By definition of $\models$, we get the result $h \models^{\alpha} \exists_{\mathrm{V}}\mathtt{X}. \bigvee_{M' \in 2^{M_{\mathrm{V}}}} \mathrm{Right}'(r, c', M \cup M') = \mathrm{Right}(r, c, M)$.

Finally, suppose c has the form $\exists a : P \hookrightarrow C.c'$. By assumption, there is a morphism $q : X^{g,\alpha_M} \hookrightarrow G$ with $q \circ a = g$ and $q \models^{\alpha_M} c'$. Following the proof of Theorem 6 in [7], and with the fact that $R^{g,\alpha_M} = (R^{\oplus})^{h,\alpha_M}$, we derive a morphism $q' : Y^{q',\alpha_M} \hookrightarrow H$ with $q' \circ b^{q',\alpha_M} = h$, i.e. $h \models^{\alpha_M} \exists b$. As the morphism b does not contain any node/edge-set variables, $h \models^{\alpha} \exists b$. By induction hypothesis, $q' \models^{\alpha} \mathrm{Right}'(r^*, c', M)$. Together, we have the result that $h \models^{\alpha} \exists b.\mathrm{Right}'(r^*, c', M) = \mathrm{Right}'(r, c, M)$. (For the case where $\langle K \hookrightarrow L, a\rangle$ has no natural pushout complement, the result can be proven by contradiction [7].)

(*Induction step;* $\Longleftarrow$) Suppose c has the form $\exists_{\mathrm{L}}\mathtt{x}.c'$. By assumption and construction, $h \models^{\alpha} \exists_{\mathrm{L}}\mathtt{x}.\mathrm{Right}'(r, c', M)$. By definition of $\models$, $h \models^{\alpha[\mathtt{x}\mapsto l]} \mathrm{Right}'(r, c', M)$ for some $l \in \mathbb{L}$. By induction hypothesis, $g \models^{\alpha_M[\mathtt{x}\mapsto l]} c'$. By definition of $\models$, we get the result that $g \models^{\alpha_M} \exists_{\mathrm{L}}\mathtt{x}.\mathrm{Right}'(r, c', M)$.

Suppose c has the form $\exists_{\mathtt{V}}\mathtt{X}.c'$. By assumption and construction, there exists some M' such that $h \models^{\alpha} \exists_{\mathtt{V}}\mathtt{X}.\mathrm{Right}'(r, c', M \cup M')$. By definition of $\models$, there exists some $V \subseteq V_G$ such that $h \models^{\alpha[\mathtt{X}\mapsto V]} \mathrm{Right}'(r, c', M \cup M')$. By induction hypothesis, $g \models^{\alpha_{M\cup M'}[\mathtt{X}\mapsto V]} c'$. Observe that $\alpha_{M\cup M'}[\mathtt{X} \mapsto V] = \alpha_M[\mathtt{X} \mapsto V \cup \{g(v) \mid (v, \mathtt{X}) \in M'\}]$. By definition of $\models$, get the result that $g \models^{\alpha_M} \exists_{\mathtt{V}}\mathtt{X}.c'$. (Analogous for case $\exists_{\mathtt{E}}$.)

Finally, suppose c has the form $\exists a : P \hookrightarrow C.c'$. By assumption, $h \models^{\alpha} \exists b.\mathrm{Right}'(r^*, c', M)$. There is a morphism $q' : Y^{q',\alpha} \hookrightarrow H$ with $q' \circ b^{q',\alpha} = h$ and $q' \models^{\alpha} \mathrm{Right}'(r^*, c', M)$. Following the proof of Theorem 6 in [7], we derive a morphism $q : X^{q,\alpha} \hookrightarrow G$ with $q \circ a^{q,\alpha} = g$, i.e. $g \models^{\alpha} \exists a$. As the morphism does not contain any node/edge-set variables, $g \models^{\alpha_M} \exists a$. By induction hypothesis, $q \models^{\alpha_M} c'$. Together, we get the result that $g \models^{\alpha_M} \exists a.c'$. (For the case where $\langle K \hookrightarrow L, a\rangle$ has no natural pushout complement, the result can be proven by contradiction [7].) □

References

[1] R. Heckel, G. Taentzer, Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering, Springer, 2020.

[2] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova, Modelling and analysis using GROOVE, International Journal on Software Tools for Technology Transfer 14 (1) (2012) 15–40.

[3] M. Strecker, Interactive and automated proofs for graph transformations, Mathematical Structures in Computer Science 28 (8) (2018) 1333–1362.

[4] P. Baldan, A. Corradini, B. König, A framework for the verification of infinite-state graph transformation systems, Information and Computation 206 (7) (2008) 869–907.

[5] B. König, J. Esparza, Verification of graph transformation systems with context-free specifications, in: ICGT 2010, Vol. 6372 of LNCS, Springer, 2010, pp. 107–122.

[6] S. Schneider, J. Dyck, H. Giese, Formal verification of invariants for attributed graph transformation systems based on nested attributed graph conditions, in: ICGT 2020, Vol. 12150 of LNCS, Springer, 2020, pp. 257–275.

[7] A. Habel, K.-H. Pennemann, Correctness of high-level transformation systems relative to nested conditions, Mathematical Structures in Computer Science 19 (2) (2009) 245–296.

[8] A. Habel, K.-H. Pennemann, A. Rensink, Weakest preconditions for high-level programs, in: Proc. International Conference on Graph Transformation (ICGT 2006), Vol. 4178 of LNCS, Springer, 2006, pp. 445–460.

[9] G. Wulandari, D. Plump, Verifying graph programs with first-order logic, in: Proc. International Workshop on Graph Computation Models (GCM 2020), Revised Selected Papers, Vol. 330 of EPTCS, 2020, pp. 181–200.

[10] A. Makhlouf, C. Percebois, H. N. Tran, Two-level reasoning about graph transformation programs, in: ICGT 2019, Vol. 11629 of LNCS, Springer, 2019, pp. 111–127.

[11] J. H. Brenas, R. Echahed, M. Strecker, Verifying graph transformation systems with description logics, in: Proc. International Conference on Graph Transformation (ICGT 2018), Vol. 10887 of LNCS, Springer, 2018, pp. 155–170.

[12] C. M. Poskitt, Verification of graph programs, Ph.D. thesis, University of York (2013).

[13] C. M. Poskitt, D. Plump, Hoare-style verification of graph programs, Fundamenta Informaticae 118 (1-2) (2012) 135–175.

[14] E. de Vries, V. Koutavas, Reverse Hoare logic, in: SEFM 2011, Vol. 7041 of LNCS, Springer, 2011, pp. 155–171.

[15] P. W. O'Hearn, Incorrectness logic, Proceedings of the ACM on Programming Languages 4 (POPL) (2020) 10:1–10:32.

[16] A. Raad, J. Berdine, H. Dang, D. Dreyer, P. W. O'Hearn, J. Villard, Local reasoning about the presence of bugs: Incorrectness separation logic, in: CAV 2020, Vol. 12225 of LNCS, Springer, 2020, pp. 225–252.

[17] A. Raad, J. Berdine, D. Dreyer, P. W. O'Hearn, Concurrent incorrectness separation logic, Proc. ACM Program. Lang. 6 (POPL) (2022) 1–29.

[18] T. Murray, An under-approximate relational logic: Heralding logics of insecurity, incorrect implementation & more, CoRR abs/2003.04791 (2020). URL `https://arxiv.org/abs/2003.04791`

[19] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM (CACM) 12 (10) (1969) 576–580.

[20] D. Plump, The design of GP 2, in: Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011), Vol. 82 of Electronic Proceedings in Theoretical Computer Science, 2012, pp. 1–16.

[21] A. S. Al-Sibahi, A. S. Dimovski, A. Wasowski, Symbolic execution of high-level transformations, in: SLE 2016, ACM, 2016, pp. 207–220.

[22] B. Azizi, B. Zamani, S. K. Rahimi, SEET: symbolic execution of ETL transformations, Journal of Systems and Software 168 (2020) 110675.

[23] B. J. Oakes, J. Troya, L. Lúcio, M. Wimmer, Full contract verification for ATL using symbolic execution, Software and Systems Modeling 17 (3) (2018) 815–849.

[24] C. M. Poskitt, Incorrectness logic for graph programs, in: Proc. International Conference on Graph Transformation (ICGT 2021), Vol. 12741 of LNCS, Springer, 2021, pp. 81–101.

[25] C. M. Poskitt, D. Plump, Verifying monadic second-order properties of graph programs, in: Proc. International Conference on Graph Transformation (ICGT 2014), Vol. 8571 of LNCS, Springer, 2014, pp. 33–48.

[26] C. Bak, D. Plump, Compiling graph programs to C, in: Proc. International Conference on Graph Transformation (ICGT 2016), Vol. 9761 of LNCS, Springer, 2016, pp. 102–117.

[27] G. Campbell, J. Romo, D. Plump, The improved GP 2 compiler, CoRR abs/2010.03993 (2020).

[28] G. Campbell, B. Courtehoute, D. Plump, Fast rule-based graph programs, Science of Computer Programming 214, 32 pages (2022). doi:10.1016/j.scico.2021.102727.

[29] G. Campbell, Efficient graph rewriting, CoRR abs/1906.05170 (2019).

[30] A. Habel, D. Plump, Relabelling in graph transformation, in: Proc. International Conference on Graph Transformation (ICGT 2002), Vol. 2505 of LNCS, Springer, 2002, pp. 135–147.

[31] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebraic Methods Program. 60-61 (2004) 17–139.

[32] K. R. Apt, F. S. de Boer, E. Olderog, Verification of Sequential and Concurrent Programs, Texts in Computer Science, Springer, 2009.

[33] S. A. Cook, Soundness and completeness of an axiom system for program verification, SIAM Journal of Computing 7 (1) (1978) 70–90.

[34] G. S. Wulandari, D. Plump, Verifying graph programs with monadic second-order logic, in: ICGT, Vol. 12741 of Lecture Notes in Computer Science, Springer, 2021, pp. 240–261.

[35] A. Habel, D. Plump, Computational completeness of programming languages based on graph transformation, in: Proc. International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001), Vol. 2030 of LNCS, Springer, 2001, pp. 230–245.

[36] R. Bruni, R. Giacobazzi, R. Gori, F. Ranzato, A logic for locally complete abstract interpretations, in: LICS 2021, IEEE, 2021, pp. 1–13.

[37] T. Isenberg, D. Steenken, H. Wehrheim, Bounded model checking of graph transformation systems via SMT solving, in: FMOODS/FORTE 2013, Vol. 7892 of LNCS, Springer, 2013, pp. 178–192.

[38] A. Habel, H. Radke, Expressiveness of graph conditions with variables, in: Proc. International Colloquium on Graph and Model Transformation (GraMoT 2010), Vol. 30 of Electronic Communications of the EASST, 2010.

[39] H. Radke, A theory of HR* graph conditions and their application to meta-modeling, Ph.D. thesis, University of Oldenburg, Germany (2016).

[40] C. Percebois, M. Strecker, H. N. Tran, Rule-level verification of graph transformations for invariants based on edges' transitive closure, in: Proc. Software Engineering and Formal Methods (SEFM 2013), Vol. 8137 of LNCS, Springer, 2013, pp. 106–121.

[41] K. Inaba, S. Hidaka, Z. Hu, H. Kato, K. Nakano, Graph-transformation verification using monadic second-order logic, in: Proc. Principles and Practice of Declarative Programming (PPDP 2011), ACM, 2011, pp. 17–28.

[42] M. Navarro, F. Orejas, E. Pino, L. Lambers, A navigational logic for reasoning about graph properties, J. Log. Algebraic Methods Program. 118 (2021) 100616.

[43] S. Schneider, L. Lambers, F. Orejas, Symbolic model generation for graph properties, in: Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2017), Vol. 10202 of LNCS, Springer, 2017, pp. 226–243.

[44] S. Schneider, L. Lambers, F. Orejas, Automated reasoning for attributed graph properties, International Journal on Software Tools for Technology Transfer (2018).

[45] L. Lambers, M. Navarro, F. Orejas, E. Pino, Towards a navigational logic for graphical structures, in: Graph Transformation, Specifications, and Nets, Vol. 10800 of LNCS, Springer, 2018, pp. 124–141.

[46] C. M. Poskitt, D. Plump, Verifying total correctness of graph programs, in: Revised Selected Papers, Graph Computation Models (GCM 2012), Vol. 61 of Electronic Communications of the EASST, 2013.

[47] C. Corrodi, A. Heußner, C. M. Poskitt, A semantics comparison workbench for a concurrent, asynchronous, distributed programming language, Formal Aspects of Computing 30 (1) (2018) 163–192.

[48] A. Habel, J. Müller, D. Plump, Double-pushout graph transformation revisited, Mathematical Structures in Computer Science 11 (5) (2001) 637–688.

[49] S. Mac Lane, Categories for the Working Mathematician, 2nd Edition, Springer, 1998.

[50] S. Awodey, Category Theory, 2nd Edition, Oxford Logic Guides, Oxford University Press, 2010.

[51] E. Riehl, Category Theory in Context, Dover Publications, 2016.

[52] A. Habel, D. Plump, *M,N*-adhesive transformation systems, in: Proc. International Conference on Graph Transformation (ICGT 2012), Vol. 7562 of LNCS, Springer, 2012, pp. 218–233.