Incorrectness Logic for Graph Programs

Christopher M. Poskitt

Singapore Management University, Singapore
cposkitt@smu.edu.sg

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 a nondeterministic graph programming language, and show how it can be used to reason deductively about program incorrectness, whether defined by the presence of forbidden graph structure or by finitely failing executions. We prove this 'incorrectness logic' to be sound and complete, and speculate on some possible future applications of it.

Keywords: Program logics · Under-approximate reasoning · Bugs

1 Introduction

Many problems in computer science and software engineering can be modelled in terms of rule-based graph transformations [13], motivating research into verifying the correctness of grammars and programs based on this unit of computation. Various approaches towards this goal have been proposed, with techniques including model checking [9], unfoldings [4, 16], k-induction [29], weakest preconditions [10, 11], abstract interpretation [17], and program logics [5, 24, 25].

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 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 paths that matter. De Vries and Koutavas [30] proposed the first program logic of this kind, using it to reason about state reachability for randomised nondeterministic algorithms. O'Hearn [21] 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 [28] 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 [14]. 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 nondeterministic attribute-manipulating graph programs. Following O'Hearn [21], we design it as an *incorrectness logic*, and show how it can be used to reason deductively about the presence of forbidden graph structures or finitely 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 a relational denotational 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. [1, 3, 20]).

The paper is organised as follows. In Section 2 we provide preliminary definitions of graphs and graph morphisms. In Section 3 we define graph programs using a relational denotational semantics, as well as an assertion language ('E-conditions') for specifying properties of program states. In Section 4, we present an incorrectness logic for graph programs and demonstrate it on some examples. In Section 5, we formally define the assertion transformations used in our incorrectness logic, and present our main soundness and completeness results. Finally, we review some related work in Section 6 before concluding in Section 7.

2 Preliminaries

We use a definition of graphs in which edges are directed, nodes (resp. edges) are partially (resp. totally) labelled, and parallel edges are allowed to exist. All graphs in this paper will be totally labelled except for the interface graphs in rule applications (for technical reasons to support relabelling [12]).

A *graph* over a label alphabet $\mathcal{C}$ is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ 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* $l_G : V_G \to \mathcal{C}$, and a total *edge labelling function* $m_G : E_G \to \{\square\}$. 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. Note that for simplicity of presentation, in this paper, we label all edges with a 'blank' label denoted by $\square$ and rendered as $\longrightarrow$ in diagrams. Note also that we use an undirected edge $(8)\!-\!(8)$ to represent a pair of edges $(8)\rightleftarrows(8)$.

We write $\mathcal{G}(\mathcal{C}_\bot)$ (resp. $\mathcal{G}(\mathcal{C})$) to denote the *class* of all (resp. all totally labelled) graphs over label alphabet $\mathcal{C}$. Let $\mathcal{L}$ denote the label alphabet $\mathbb{Z}^+$, i.e. all non-empty sequences of integers. In diagrams we will delimit the integers of the sequence using colons, e.g. 5:6:7:8.

A *graph morphism* $g\colon G \to H$ between graphs G, H in $\mathcal{G}(\mathcal{C}_\perp)$ 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 and labels; that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, 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.

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.

3 Graph Programs and Assertions

We begin by introducing the graph programs that will be the target of our incorrectness logic, as well as an assertion language ('E-conditions') that will be used for specifying properties of the program states (which consist of graphs). To allow for a self-contained presentation, our programs are a simplified 'core' of full-fledged graph programming languages (e.g. GP 2 [23]) which have several more features for practicality (e.g. additional types, negative application conditions).

First, we define the underlying unit of computation in graph programs: the application of a graph transformation rule with relabelling.

Definition 1 (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 $K \in \mathcal{G}(\mathcal{L}_\perp)$, 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* $h\colon R \hookrightarrow H$, 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 [12] for the technical details).

In practical graph programming languages, we need a more powerful unit of computation—the rule schema—which describes (potentially) infinitely many concrete rules by labelling the graphs over expressions. We define a simple abstract syntax 'Exp' (Figure 1) which derives a label alphabet of (lists of) integer expressions, including variables ('Var') of type integer.

A graph in $\mathcal{G}(\mathcal{L})$ can be obtained from a graph in $\mathcal{G}(\mathrm{Exp})$ by means of an *interpretation*, which is a partial function $I\colon \mathrm{Var} \to \mathbb{Z}$. We denote the domain of

```
Exp      ::= Integer | Integer ':' Exp
Integer  ::= Digit {Digit} | Var | '−' Integer | Integer ArithOp Integer
ArithOp ::= '+' | '−' | '∗' | '/'
```

Fig. 1: Abstract syntax of rule schema labels

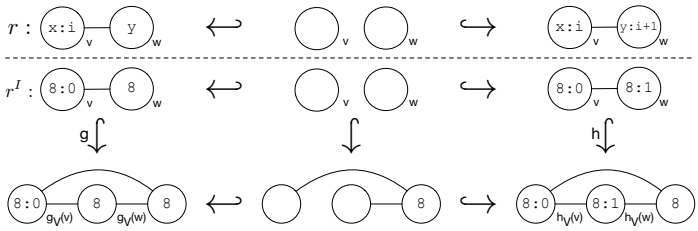


Fig. 2: Example rule schema application

I by $\mathrm{dom}(I)$, and the set of variables used in a graph $G \in \mathcal{G}(\mathrm{Exp})$ by $\mathrm{vars}(G)$. If $\mathrm{vars}(G) \subseteq \mathrm{dom}(I)$, then $G^I \in \mathcal{G}(\mathcal{L})$ is the graph obtained by evaluating the expressions in the standard way, with variables $\mathtt{x}$ substituted for $I(\mathtt{x})$. Interpretations may also be applied to morphisms, e.g. $p\colon P \hookrightarrow C$ becomes $p^I : P^I \hookrightarrow C^I$.

Definition 2 (Rule schema). A *rule schema* $r\colon \langle L \Rightarrow R \rangle$ with $L, R \in \mathcal{G}(\mathrm{Exp})$ represents concrete rules $r^I : \langle L^I \hookleftarrow K \hookrightarrow R^I \rangle$ where $\mathrm{dom}(I) = \mathrm{vars}(L)$ and K consists of the preserved nodes only (with all nodes unlabelled). Note that we assume for any rule schema, $\mathrm{vars}(R) \subseteq \mathrm{vars}(L)$. □

The application of a rule schema $r = \langle L \Rightarrow R \rangle$ to a graph $G \in \mathcal{G}(\mathcal{L})$ consists of the following steps: (1) choose an interpretation I with $\mathrm{dom}(I) = \mathrm{vars}(L)$; (2) choose a *match*, i.e. a morphism $g : L^I \hookrightarrow G$ that satisfies the dangling condition with respect to $r^I : \langle L^I \hookleftarrow K \hookrightarrow R^I \rangle$; (3) apply r^I with match g. If a graph H with comatch $h : R^I \hookrightarrow H$ 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. finite failure).

Example 1 (Rule schema application). Figure 2 displays a rule schema $r\colon \langle L \hookleftarrow K \hookrightarrow R \rangle$ with its interface (top row), a possible instantiation r^I where $I(\mathtt{x}) = I(\mathtt{y}) = 8$ and $I(\mathtt{i}) = 0$ (middle row). Finally, the bottom row depicts a direct derivation from G (bottom left) to H (bottom right) via r^I. □

Definition 3 (Graph programs). *(Graph) programs* are defined inductively. Given a set of rule schemata $\mathcal{R}$, $\mathcal{R}$ and $\mathcal{R}!$ are programs. If P, Q are programs and $\mathcal{R}$ a set of rule schemata, then $P; Q$ and $\mathtt{if}\ \mathcal{R}\ \mathtt{then}\ P\ \mathtt{else}\ Q$ are programs. □

$$
\begin{array}{rcl}
[\![\mathcal{R}]\!]ok & = & \{(G,H) \mid G \Rightarrow_{\mathcal{R}} H\} \\
[\![\mathcal{R}]\!]er & = & \{(G,G) \mid G \not\Rightarrow_{\mathcal{R}}\} \\
[\![P;Q]\!]\epsilon & = & \{(G,H) \mid \exists G'.(G,G') \in [\![P]\!]ok \text{ and } (G',H) \in [\![Q]\!]\epsilon\} \\
 & & \cup \ (\text{if } \epsilon = er \text{ then } \{(G,H) \mid (G,H) \in [\![P]\!]er\}) \\
[\![\mathcal{R}!]\!]ok & = & [\![\mathcal{R}]\!]er \cup [\![\mathcal{R};\mathcal{R}!]\!]ok \\
[\![\mathcal{R}!]\!]er & = & \emptyset \\
[\![\texttt{if } \mathcal{R} \texttt{ then } P \texttt{ else } Q]\!]\epsilon & = & \{(G,H) \mid \exists G'.(G,G') \in [\![\mathcal{R}]\!]ok \text{ and } (G,H) \in [\![P]\!]\epsilon\} \\
 & & \cup \{(G,H) \mid (G,G) \in [\![\mathcal{R}]\!]er \text{ and } (G,H) \in [\![Q]\!]\epsilon\}
\end{array}
$$

Fig. 3: A relational denotational semantics for graph programs

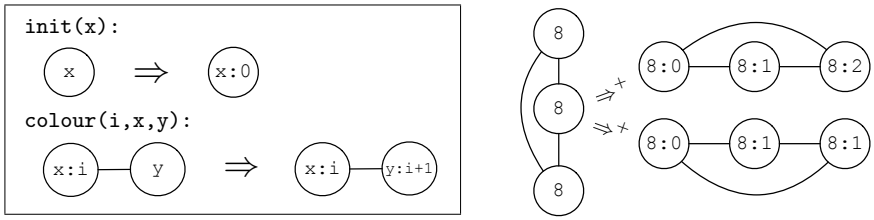


Fig. 4: Rules for the program `init; colour!` and two possible executions

Intuitively, $\mathcal{R}$ denotes a single nondeterministic application of a rule schemata set. This results in failure if none of the rules are applicable to the current graph. The program $\mathcal{R}!$ denotes as-long-as-possible iteration of $\mathcal{R}$, in which the iteration terminates the moment that $\mathcal{R}$ is no longer applicable to the current graph (the program never fails). Finally, the program $P;Q$ denotes sequential composition, and `if` $\mathcal{R}$ `then` P `else` Q denotes conditional branching, determined by testing the applicability of $\mathcal{R}$ (note that $\mathcal{R}$ will not transform the current graph).

Each graph program is given a simple relational denotational semantics (in the style of [21]). We associate each program P with two semantic functions, $[\![P]\!]ok$ and $[\![P]\!]er$, which respectively describe state (i.e. graph) transitions for successful and finitely failing computations. Unlike operational semantics for graph programs (e.g. [23]), we do not explicitly track a 'fail' state, but rather return pairs (G,H) where H is the last graph derived from G before the failure.

Definition 4 (Semantics). The *semantics* of a graph program P is given by a binary relation $[\![P]\!]\epsilon \subseteq \mathcal{G}(\mathcal{L}) \times \mathcal{G}(\mathcal{L})$, defined according to Figure 3. □

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

Example 2 (Buggy colouring). Figure 4 contains an example graph program $P =$ `init;colour!` that purportedly computes a graph colouring, i.e. an association of integers ('colours') with nodes such that no two adjacent nodes are associated with the same colour. The program nondeterministically assigns a colour of '0'

to a node, encoding it as the second element of the label's sequence, before iteratively matching adjacent pairs of coloured/uncoloured nodes and assigning a colour to the latter obtained by incrementing the colour of the former. Note that the edges are undirected for simplicity.

Two possible executions are shown in Figure 4, the first of which leads to a correct colouring, and the second of which leads to an illegal one. Moreover, the program can finitely fail on input graphs for which `init` has no match. (We shall use incorrectness logic to logically prove the presence of such outcomes.)

Before we can define an incorrectness logic for graph programs, we require an assertion language for expressing properties of the states, i.e. graphs in $\mathcal{G}(\mathcal{L})$. For this purpose we shall use nested conditions with expressions ('E-conditions'), which allow for the specification of properties at the same level of abstraction, i.e. by graph morphisms annotated with expressions. The concept of E-conditions was introduced in prior work [24, 25], but we shall present an alternative definition that more cleanly separates the quantification of graph structure and integer variables (the latter was handled implicitly in previous work, which led to more complicated assertion transformations).

Definition 5 (E-condition). Let P denote a graph in $\mathcal{G}(\text{Exp})$. A *nested condition with expressions (*short. *E-condition) over* P is of the form $\texttt{true}$, γ, $\exists \texttt{x}.c$, or $\exists a.c'$, where γ is an interpretation constraint (i.e. a Boolean expression over 'Exp'), $\texttt{x}$ is a variable in Var, c is an E-condition over P, $a\colon P \hookrightarrow C$ is an injective graph morphism over $\mathcal{G}(\text{Exp})$, and c' is an E-condition over C. Moreover, $\neg c_1$, $c_1 \wedge c_2$, and $c_1 \vee c_2$ are E-conditions over P if c_1, c_2 are E-conditions over P. □

The *free variables* of an E-condition c, denoted $\text{FV}(c)$, are those variables present in node labels and interpretation 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 an *E-constraint*. Furthermore, a mapping of free variables to expressions $\sigma = (\texttt{x}_1 \mapsto e_1, \cdots)$ is called a *substitution*, and c^σ denotes the E-condition c but with all free variables $\texttt{x}$ substituted for $\sigma(\texttt{x})$.

Definition 6 (Satisfaction of E-conditions). Let c denote an E-condition over P, I an interpretation with $\text{dom}(I) = \text{FV}(c)$, and $p\colon P^I \hookrightarrow G$ an injective morphism over $\mathcal{G}(\mathcal{L})$. The *satisfaction* relation $p \models^I c$ is defined inductively.

If c has the form $\texttt{true}$, then $p \models^I c$ always. If c is an interpretation constraint γ, then $p \models^I c$ if $\gamma^I = \text{true}$ (defined in the standard way). If c has the form $\exists \texttt{x}.c'$ where c' is an E-condition over P, then $p \models^I c$ if $p \models^{I[\texttt{x}\mapsto v]} c'$ for some $v \in \mathbb{Z}$. If c has the form $\exists a\colon P \hookrightarrow C.c'$ where c' is an E-condition over C, then $p \models^I c$ if there exists an injective morphism $q\colon C^I \hookrightarrow G$ such that $q \circ a^I = p$ and $q \models^I c'$.

$$\begin{array}{ccc} P^I & \xhookrightarrow{a^I} & C^I \\ & {\scriptstyle p}\searrow \quad \swarrow{\scriptstyle q} & \\ & G & \end{array} \quad (=) \quad q \models^I c'$$

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

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

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

Example 3 (E-constraint). The following E-constraint expresses that for every pair of integer-labelled nodes, if the labels differ, then the nodes are adjacent:

$$\forall \mathtt{x}, \mathtt{y}.\ \forall \textcircled{\mathtt{x}}_v\ \ \textcircled{\mathtt{y}}_w\ .\ \mathtt{x} \neq \mathtt{y} \Longrightarrow \exists \textcircled{\mathtt{x}}_v \rightarrow \textcircled{\mathtt{y}}_w\ \vee \exists \textcircled{\mathtt{x}}_v \leftarrow \textcircled{\mathtt{y}}_w$$

Note that v, w are node identifiers to indicate which nodes are the same along the chain of nested morphisms, as can be seen when denoting them in full:

$$\forall \mathtt{x}, \mathtt{y}.\ \forall \emptyset \hookrightarrow \textcircled{\mathtt{x}}_v\ \ \textcircled{\mathtt{y}}_w\ .\ \mathtt{x} \neq \mathtt{y} \Longrightarrow \exists \textcircled{\mathtt{x}}_v\ \ \textcircled{\mathtt{y}}_w\ \hookrightarrow \textcircled{\mathtt{x}}_v \rightarrow \textcircled{\mathtt{y}}_w \vee \exists \textcircled{\mathtt{x}}_v\ \ \textcircled{\mathtt{y}}_w\ \hookrightarrow \textcircled{\mathtt{x}}_v \leftarrow \textcircled{\mathtt{y}}_w$$

These node identifiers may be omitted when the mappings are unambiguous.

4 Proving the Presence of Bugs

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. [24, 25]) 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 [21], 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 bugs*, 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

$$\textsc{RuleSetSucc}\ \ \vdash [c \wedge \mathrm{App}(\mathcal{R})]\ \mathcal{R}\ [ok : \mathrm{WPost}(\mathcal{R}, c)][er : \texttt{false}]$$

$$\textsc{RuleSetFail}\ \ \vdash [c \wedge \neg\mathrm{App}(\mathcal{R})]\ \mathcal{R}\ [ok : \texttt{false}][er : c \wedge \neg\mathrm{App}(\mathcal{R})]$$

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

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

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

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

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

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

Fig. 5: Incorrectness axioms and proof rules for graph programs

formal proofs of the presence of *illegal graph structure*, but it can also facilitate proofs of the presence of *finite failure*. To accommodate this, we adopt O'Hearn's approach [21] of tracking exit conditions ϵ in the result, $[c]P[\epsilon : d]$, using ok to represent normal executions and er to track finite failures.

Definition 7 (Under-approximate validity). Let c, d denote E-constraints, P a graph program, and ϵ an exit condition. 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 $H \models d$, there exists a graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \models c$ and $(G, H) \in [\![P]\!]\epsilon$. □

Figure 5 presents the axioms and proof rules of our incorrectness logic for graph programs, which are adapted from O'Hearn's incorrectness logic for imperative programs [21]. 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 $\vdash [c]P[ok : d_1][er : d_2]$ as shorthand for two separate triples, $\vdash [c]P[ok : d_1]$ and $\vdash [c]P[er : d_2]$.

Note that a number of axioms and proof rules rely on some transformations that we have not yet defined: $\mathrm{App}(\mathcal{R})$, which expresses the existence of a match for $\mathcal{R}$, and $\mathrm{WPost}(\mathcal{R}, c)$, which expresses the *weakest postcondition* that must

be satisfied to guarantee the existence of a pre-state satisfying c. These transformations will be formally defined in Section 5.

The axioms RuleSetSucc and RuleSetFail allow for reasoning about the most fundamental unit of graph programs: rule schema application. The former covers the successful case: if a graph satisfies the weakest postcondition for rule schemata set $\mathcal{R}$ and E-constraint c, then it can be derived from some graph satisfying the presumption $c \wedge \mathrm{App}(\mathcal{R})$. The latter of the axioms covers the possibility that $\mathcal{R}$ cannot be applied: in this case, we have an exit condition of er to track its finite failure.

Sequential composition is handled by SeqSucc as well as SeqFail (to cover the possibility of the first program resulting in failure). The conditional construct is covered by IfElse: note that failure can only result from failure in the two branches, and not from the guard $\mathcal{R}$, which is simply tested to choose the branch.

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 [2, 14]. 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 [21].

For the iteration of rule schemata sets, we have a number of cases. The axiom IterZero covers the case when a rule schemata set is no longer applicable (note that this does not result in failure). The proof rule Iter unrolls a step of the iteration. Traditional loop invariants are less important in these proof rules than they are for Hoare logic, as we are reasoning about a subset of paths rather than *all* of them. To see this, consider the triple $\models [inv]\mathcal{R}![ok : inv \wedge \neg\mathrm{App}(\mathcal{R})]$ with invariant inv. Under-approximate validity requires every graph H satisfying inv and $\neg\mathrm{App}(\mathcal{R})$ to be derivable by applying $\mathcal{R}!$ to some graph G satisfying inv. One can always find such a graph by taking $G = H$.

Finally, IterVar combines IterZero and Iter into one rule. It expresses that a triple $\vdash [c_0]\mathcal{R}![ok : c_n]$ can be proven if: (1) c_n implies the termination of the iteration (i.e. the non-applicability of $\mathcal{R}$); and (2) if triples can be proven for the n iterations of $\mathcal{R}$. IterVar is a stricter version of the backwards variant rule for while-loops in [21, 30]: had we adopted the rule in full, we would be able to prove triples such as $\vdash [c(0)]\mathcal{R}![ok : \exists n.n \geq 0.c(n) \wedge \neg\mathrm{App}(\mathcal{R})]$. Here, $c(i)$ denotes a parameterised predicate, i.e. in our case, a function mapping expressions to E-constraints. Unfortunately, these are not possible to express using E-constraints, and including them would strictly increase their expressive power beyond first-order graph properties and the current capabilities of 'WPost'.

Example 4 (Colouring: finite failure). In our first example, we prove the incorrectness specification $\vdash [\neg\exists \mathtt{x}.\exists(\times)]\ \mathtt{init}; \mathtt{colour}!\ [er : \neg\exists \mathtt{x}.\exists(\times)]$ for the program of Figure 4. This triple specifies that if a graph does not contain any integer-labelled nodes, then it can be derived from another graph satisfying the same condition that the program finitely fails on. Since `init` would fail on any such graph, this specification is valid: the graph in the post-state is exactly the graph in the pre-state. Figure 6 proves this triple using incorrectness logic.

$$\textsc{RuleSetFail}\ \frac{\textsc{Cons}\ \dfrac{\textsc{SeqFail}\ \dfrac{\square}{\vdash [\texttt{true} \wedge \neg\mathrm{App}(\texttt{init})]\ \texttt{init}\ [er : \texttt{true} \wedge \neg\mathrm{App}(\texttt{init})]}}{\vdash [\neg\mathrm{App}(\texttt{init})]\ \texttt{init}\ [er : \neg\mathrm{App}(\texttt{init})]}}{\vdash [\neg\mathrm{App}(\texttt{init})]\ \texttt{init};\texttt{colour!}\ [er : \neg\mathrm{App}(\texttt{init})]}$$

Fig. 6: Proving the presence of failure (E-constraints in Figure 8)

$$\dfrac{\dfrac{\dfrac{\square}{\vdash [\texttt{true} \wedge \mathrm{App}(\texttt{init})]\ \texttt{init}\ [ok : \mathrm{WPost}(\texttt{init}, \texttt{true})]}}{\vdash [\mathrm{App}(\texttt{init})]\ \texttt{init}\ [ok : \mathit{illegal}]} \quad \dfrac{\dfrac{\square}{\vdash [\mathit{illegal} \wedge \neg\mathrm{App}(\texttt{colour})]\ \texttt{colour!}\ [ok : \mathit{illegal} \wedge \neg\mathrm{App}(\texttt{colour})]}}{\vdash [\mathit{illegal}]\ \texttt{colour!}\ [ok : \mathit{illegal} \wedge \neg\mathrm{App}(\texttt{colour})]}}{\vdash [\mathrm{App}(\texttt{init})]\ \texttt{init};\texttt{colour!}\ [ok : \mathit{illegal} \wedge \neg\mathrm{App}(\texttt{colour})]}$$

Fig. 7: Proving the presence of an illegal graph (E-constraints in Figure 8)

Example 5 (Colouring: illegal graph). While proving the presence of failure for the program of Figure 4 is simple, there are some interesting subtleties involved in proving the presence of illegal graph structure. Let us consider:

$$\vdash [\exists \mathtt{x}.\exists(\mathtt{x})]\ \texttt{init};\texttt{colour!}\ [ok : \big(\exists \mathtt{a},\mathtt{b},\mathtt{j}.\exists(\mathtt{a}:\mathtt{j})\!-\!(\mathtt{b}:\mathtt{j})\big) \wedge (\exists \mathtt{x}.\exists(\mathtt{x}:0)) \wedge \big(\neg\exists \mathtt{x}.\exists(\mathtt{x}:1)\!-\!(\mathtt{y})\big)]$$

which specifies that if a graph has an illegal colouring, at least one node coloured '0', and `colouring` is no longer applicable, then it can be derived by applying the program to some graph containing an integer-labelled node (i.e. that `init` does not fail on). This triple is provable (Figure 7) and valid, but not because of any problem with `colour`. Consider, for example, the graph $(8:0)\ (8:8)\!-\!(8:8)$. This is trivially reachable from graphs that already contain the illegal structure, e.g. $(8)\ (8:8)\!-\!(8:8)$, thus we are able to complete the proof using the ITERZERO rule.

Finally, we strengthen the condition on the result to try and prove the presence of an illegal colouring that is created by the program itself (see Figure 8 for the E-constraints):

$$\vdash [c]\ \texttt{init};\texttt{colour!}\ [ok : d \wedge \neg\mathrm{App}(\texttt{colour})]$$

The E-constraint c expresses that there exists at least one node and that no node is coloured (instead of using conjunction, we express this more compactly using nesting). The E-constraint d expresses that there are three coloured nodes (with colours $0, 1, 1$). Together, the triple specifies that every graph satisfying $d \wedge \neg\mathrm{App}(\texttt{colour})$ can be derived from at least one graph satisfying c. This triple is valid and provable (Figure 9) as the illegal colouring is a logical possibility of some executions of `colour!`. Note that we cannot use an assertion such as $\exists \mathtt{a},\mathtt{b}.\exists(\mathtt{a}:1)\!-\!(\mathtt{b}:1)$ in place of d, as this is satisfied by the graph $(8:1)\!-\!(8:1)$ which is impossible to derive from any graph satisfying c.

As E-constraints are equivalent to first-order logic on graphs [24], we are precluded from proving a more general non-local condition, e.g. "there exists a

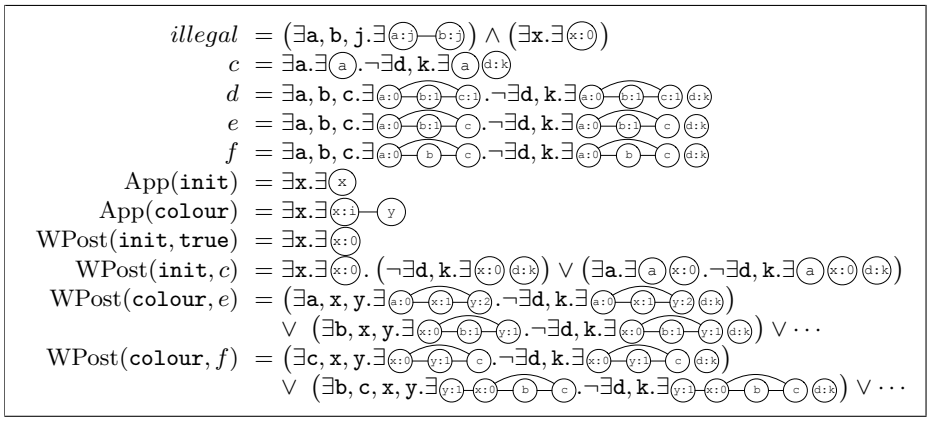


Fig. 8: E-constraints used in the proofs of Figures 6, 7, and 9

cycle with an illegal colouring". However, there are more powerful logics equipped with similar transformations that may be possible to use instead [19, 27].

5 Transformations, Soundness, and Completeness

This section presents formal definitions and characterisations of the transformations that are used in some of our incorrectness axioms and proof rules. Following this, we present our main technical result: the soundness and completeness of our incorrectness logic with respect to the denotational semantics.

First, we consider 'App', which transforms a set of rule schemata into an E-constraint that expresses the minimum requirements on a graph for at least one of the rules to be applicable. Intuitively, the E-constraint expresses the presence of a match for a left-hand side, i.e. a morphism that satisfies the dangling condition. This transformation is adapted from similar transformations in [10, 24].

Proposition 1 (Applicability). For every graph $G \in \mathcal{G}(\mathcal{L})$ and set of rule schemata $\mathcal{R}$,

$$G \models \mathrm{App}(\mathcal{R}) \text{ if and only if } \exists H.\ G \Rightarrow_{\mathcal{R}} H.$$

Construction. Define $\mathrm{App}(\emptyset) = \texttt{false}$ and then $\mathrm{App}(\{r_1, \cdots r_n\}) = \mathrm{app}(r_1) \vee \cdots \mathrm{app}(r_n)$. Given a rule schema $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ over variables $\mathtt{x}_1, \cdots \mathtt{x}_m$, define $\mathrm{app}(r) = \exists \mathtt{x}_1, \cdots \mathtt{x}_m.\ \exists \emptyset \hookrightarrow L.\ \mathrm{Dang}(r)$.

Finally, define $\mathrm{Dang}(r) = \bigwedge_{a \in A} \neg \exists \mathtt{x}_a . \exists a$ 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 each $L^{\oplus}$ is a graph that can be obtained from L by adding either: (1) a single loop with label $\square$; (2) a single edge with label $\square$ between distinct nodes; or (3) a single node labelled with fresh variable $\mathtt{x}_a$ and a non-looping edge incident to it with label $\square$. If the index set A is empty, then $\mathrm{Dang}(r) = \texttt{true}$. $\square$

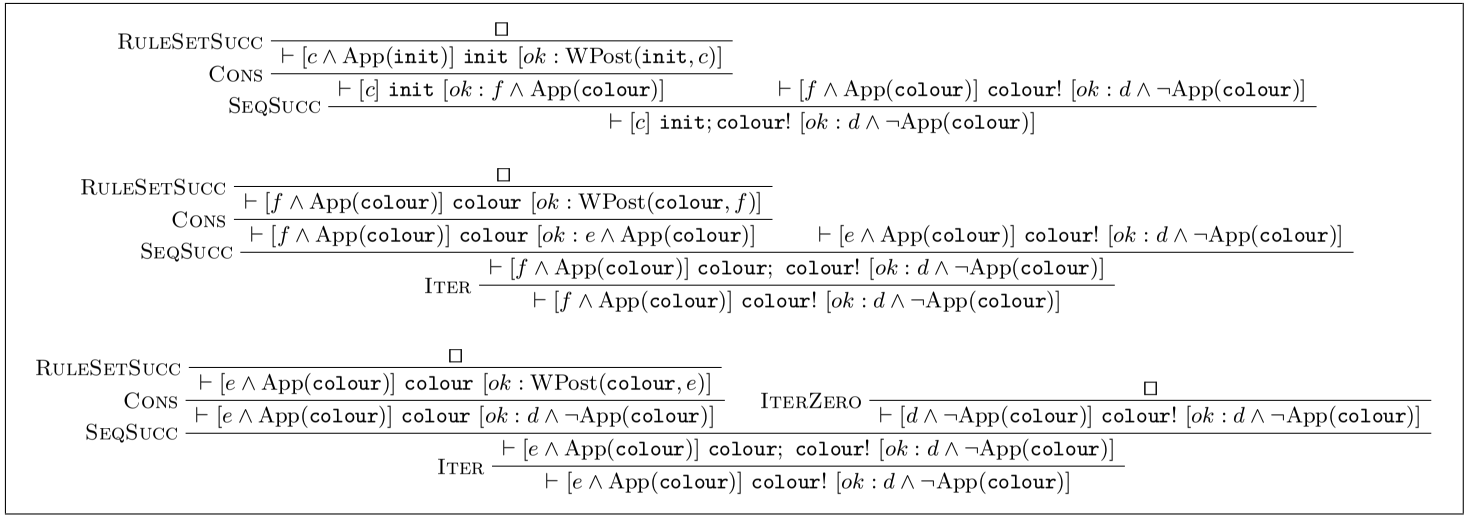


Fig. 9: Proving the presence of an illegal colouring (E-constraints in Figure 8)

Next, we consider 'WPost', which 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 an E-constraint c into an E-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 the shifting constructions of [10, 24] to handle the explicit quantification of label variables. Intuitively, this step is handled via a disjunction over all possible substitutions of a variable in c for integer expressions or variables in L, i.e. to account for interpretations in which they refer to the same values.

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

Lemma 1 (E-constraint to left E-condition). Let r denote a rule schema and c an E-constraint labelled over lists of variables distinct from those in r. For every graph $G \in \mathcal{G}(\mathcal{L})$ and morphism $g\colon L^I \hookrightarrow G$ with $\mathrm{dom}(I) = \mathrm{vars}(L)$,

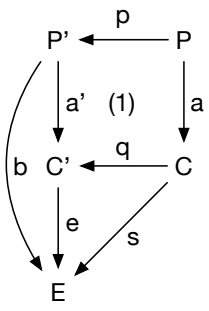
$$g\colon L^I \hookrightarrow G \models^I \mathrm{Shift}(r, c) \text{ if and only if } G \models c.$$

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

$$\mathrm{Shift}'(p, \exists \mathtt{x}.\ c) = \big(\exists \mathtt{x}.\ \mathrm{Shift}'(p, c)\big) \bigvee_{l \in \Sigma_{P'}} \mathrm{Shift}'(p, c^{(\mathtt{x} \mapsto l)})$$

$$\mathrm{Shift}'(p, \exists a\colon P \hookrightarrow C.\ c) = \bigvee_{e \in \varepsilon} \exists b\colon P' \hookrightarrow E.\ \mathrm{Shift}'(s\colon C \hookrightarrow E, c)$$

In the third case, $\Sigma_{P'}$ is the set of all variables and integer expressions present in the labels of $V_{P'}$. In the fourth case, construct pushout (1) of p and a as depicted in the diagram. The disjunction ranges over the set ε, which we define to contain every surjective morphism $e\colon C' \hookrightarrow E$ 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.)



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

Example 6 (Shift). Consider the rule schema `init` (Figure 4) and E-constraint c (Figure 8). After simplification, the transformation $\mathrm{Shift}(\mathtt{init}, c)$ results in:

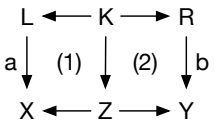
$$\big(\exists \textcircled{\mathtt{x}} \hookrightarrow \textcircled{\mathtt{x}}.\neg\exists \mathtt{d}, \mathtt{k}.\exists \textcircled{\mathtt{x}} \hookrightarrow \textcircled{\mathtt{x}}\ \textcircled{\mathtt{d:k}}\big) \vee \big(\exists \mathtt{a}.\exists \textcircled{\mathtt{x}} \hookrightarrow \textcircled{\mathtt{a}}\ \textcircled{\mathtt{x}}.\neg\exists \mathtt{d}, \mathtt{k}.\exists \textcircled{\mathtt{a}}\ \textcircled{\mathtt{x}} \hookrightarrow \textcircled{\mathtt{a}}\ \textcircled{\mathtt{x}}\ \textcircled{\mathtt{d:k}}\big)$$

The second intermediate transformation for 'WPost' is 'Right', which transforms an E-condition over the left-hand side of a rule to an E-condition over the right-hand side. This construction is based on transformation 'L' from [10, 24] but in the reverse direction.

Lemma 2 (Left to right E-condition). Let $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ denote a rule schema and c an E-condition over L. Then for every direct derivation $G \Rightarrow_{r,g,h} H$ with $g\colon L^I \hookrightarrow G$ and $h\colon R^I \hookrightarrow H$,

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

Construction. We define $\mathrm{Right}(r, \mathtt{true}) = \mathtt{true}$, $\mathrm{Right}(r, \gamma) = \gamma$, and $\mathrm{Right}(r, \exists \mathtt{x}.\ c) = \exists \mathtt{x}.\ \mathrm{Right}(r, c)$. Let $\mathrm{Right}(r, \exists a.\ c) = \exists b.\ \mathrm{Right}(r^*, c)$ if $\langle K \hookrightarrow L, a \rangle$ has a natural pushout complement (1), where $r^* = \langle X \hookleftarrow Z \hookrightarrow Y \rangle$ denotes the rule 'derived' by also constructing natural pushout (2). If $\langle K \hookrightarrow L, a \rangle$ has no natural pushout complement, then $\mathrm{Right}(r, \exists a.\ c) = \mathtt{false}$.



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

Example 7 (Right). Continuing from Example 6, applying the transformation $\mathrm{Right}(\mathtt{init}, \mathrm{Shift}(\mathtt{init}, c))$ results in the E-condition:

$$\big(\exists \textcircled{x:0} \hookrightarrow \textcircled{x:0}.\neg\exists \mathtt{d}, \mathtt{k}.\exists \textcircled{x:0} \hookrightarrow \textcircled{x:0}\ \textcircled{d:k}\big) \vee \big(\exists \mathtt{a}.\exists \textcircled{x:0} \hookrightarrow \textcircled{a}\ \textcircled{x:0}.\neg\exists \mathtt{d}, \mathtt{k}.\exists \textcircled{a}\ \textcircled{x:0} \hookrightarrow \textcircled{a}\ \textcircled{x:0}\ \textcircled{d:k}\big)$$

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

Proposition 2 (Weakest postcondition). Let $\mathcal{R}$ denote a rule schemata set and c an E-constraint. Then for every graph $H \in \mathcal{G}(\mathcal{L})$,

$$H \models \mathrm{WPost}(\mathcal{R}, c) \text{ if and only if } \exists G.\ G \models c \text{ and } G \Rightarrow_{\mathcal{R}} H.$$

Construction. Define $\mathrm{WPost}(\emptyset, c) = \mathtt{false}$ and $\mathrm{WPost}(\mathcal{R}, c) = \bigvee_{r \in \mathcal{R}} \mathrm{wpost}(r, c)$. Let $\mathrm{wpost}(r, c) = \exists \mathtt{x}_1, \cdots \mathtt{x}_n.\exists \emptyset \hookrightarrow R.\mathrm{Dang}(r^{-1}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r, c))$ where $\{\mathtt{x}_1, \cdots, \mathtt{x}_n\} = \mathrm{vars}(R)$ and r^{-1} is the reversal of rule r. □

Example 8 (WPost). Continuing from Example 7, applying the transformation $\mathrm{WPost}(\mathtt{init}, c)$ results in the E-constraint given in Figure 8.

Finally, using the characterisations of 'App' and 'WPost', we can present the main technical results of our paper: the soundness and completeness of our incorrectness logic for graph programs. Soundness means that any triple provable in our logic is valid in the sense of Definition 7, i.e. that graphs satisfying the result are reachable from some graph satisfying the presumption. The proof of this theorem is by structural induction on triples.

Theorem 1 (Soundness). For all E-constraints c, d, graph programs P, and exit conditions ϵ,

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

□

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* [7] in which completeness is relative to the existence of an oracle for deciding the validity of assertions (as in Cons). The idea is to separate incompleteness due to the incorrectness logic from incompleteness in deducing valid assertions, and determine that no proof rules are missing. Our proof relies on some semantically (or extensionally) defined assertions, WPOST$[P, c]$, that characterise exactly the weakest postcondition of an arbitrary program P relative to an E-constraint c.

Theorem 2 (Relative completeness). For all E-constraints c, d, graph programs P, and exit conditions ϵ,

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

□

It is important to remark that it is unknown whether E-constraints are *expressive* enough to specify precisely the assertion WPOST$[P, c]$ in general; in fact, there is evidence to suggest they may not be [31]. This is, however, a limitation of the logic and not the incorrectness proof rules, and expressiveness may not be a problem faced by stronger assertion languages for graphs, such as those supporting non-local properties [19, 22, 27].

6 Related Work

Over-approximate program logics for proving the absence of bugs have been studied extensively [2]. 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 finitely failing execution paths). The first under-approximate calculus of this kind was introduced by De Vries and Koutavas [30], 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 [21] 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 finitely failing executions (er). 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 E-conditions are not

expressive enough to specify parameterised graph properties, for example, the existence of a cycle of length n.

Raad et al. [28] 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. [6] 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 finite 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 [9]. 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 [15].

7 Conclusion and Future Work

We proposed an incorrectness logic for under-approximate reasoning about graph programs, 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 finitely 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 a denotational semantics of graph programs, and demonstrated their use to prove the presence of various bugs in a faulty node colouring program.

This paper was principally a theoretical exposition, but was motivated by some potentially interesting applications. One idea (suggested by O'Hearn [21]) 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. [1, 3, 20]). 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 [26], switch to under-approximate proofs instead and reason about the circumstances that could cause some undesirable result to be reachable.

Beyond exploring these potential applications, future work should also extend our logic to a full-fledged graph programming language (e.g. GP 2 [23], or the recipes of GROOVE [8, 9]). 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 (reasoning over a whole graph can be counter-intuitive, as Examples 4 and 5 demonstrate), or some derived proof rules for simplifying reasoning about common patterns.

Acknowledgements. I am grateful to the ICGT'21 referees for their detailed reviews and suggestions, which have helped to improve the quality of this paper.

References

1. Al-Sibahi, A.S., Dimovski, A.S., Wasowski, A.: Symbolic execution of high-level transformations. In: SLE 2016. pp. 207–220. ACM (2016)
2. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009)
3. Azizi, B., Zamani, B., Rahimi, S.K.: SEET: symbolic execution of ETL transformations. Journal of Systems and Software **168**, 110675 (2020)
4. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Information and Computation **206**(7), 869–907 (2008)
5. Brenas, J.H., Echahed, R., Strecker, M.: Verifying graph transformation systems with description logics. In: ICGT 2018. LNCS, vol. 10887, pp. 155–170. Springer (2018)
6. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: LICS 2021. IEEE (2021), to appear
7. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journal of Computing **7**(1), 70–90 (1978)
8. Corrodi, C., Heußner, A., Poskitt, C.M.: A semantics comparison workbench for a concurrent, asynchronous, distributed programming language. Formal Aspects of Computing **30**(1), 163–192 (2018)
9. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. International Journal on Software Tools for Technology Transfer **14**(1), 15–40 (2012)
10. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19**(2), 245–296 (2009)
11. Habel, A., Pennemann, K., Rensink, A.: Weakest preconditions for high-level programs. In: ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer (2006)
12. Habel, A., Plump, D.: Relabelling in graph transformation. In: ICGT 2002. LNCS, vol. 2505, pp. 135–147. Springer (2002)
13. Heckel, R., Taentzer, G.: Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering. Springer (2020)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM (CACM) **12**(10), 576–580 (1969)

15. Isenberg, T., Steenken, D., Wehrheim, H.: Bounded model checking of graph transformation systems via SMT solving. In: FMOODS/FORTE 2013. LNCS, vol. 7892, pp. 178–192. Springer (2013)
16. König, B., Esparza, J.: Verification of graph transformation systems with context-free specifications. In: ICGT 2010. LNCS, vol. 6372, pp. 107–122. Springer (2010)
17. Makhlouf, A., Percebois, C., Tran, H.N.: Two-level reasoning about graph transformation programs. In: ICGT 2019. LNCS, vol. 11629, pp. 111–127. Springer (2019)
18. Murray, T.: An under-approximate relational logic: Heralding logics of insecurity, incorrect implementation & more. CoRR **abs/2003.04791** (2020), https://arxiv.org/abs/2003.04791
19. Navarro, M., Orejas, F., Pino, E., Lambers, L.: A navigational logic for reasoning about graph properties. Journal of Logical and Algebraic Methods in Programming **118**, 100616 (2021)
20. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Full contract verification for ATL using symbolic execution. Software and Systems Modeling **17**(3), 815–849 (2018)
21. O'Hearn, P.W.: Incorrectness logic. Proceedings of the ACM on Programming Languages **4**(POPL), 10:1–10:32 (2020)
22. Orejas, F., Pino, E., Navarro, M., Lambers, L.: Institutions for navigational logics for graphical structures. Theoretical Computer Science **741**, 19–24 (2018)
23. Plump, D.: The design of GP 2. In: WRS 2011. EPTCS, vol. 82, pp. 1–16 (2011)
24. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, U. of York (2013)
25. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae **118**(1-2), 135–175 (2012)
26. Poskitt, C.M., Plump, D.: Verifying total correctness of graph programs. ECEASST **61** (2013)
27. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: ICGT 2014. LNCS, vol. 8571, pp. 33–48. Springer (2014)
28. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: CAV 2020. LNCS, vol. 12225, pp. 225–252. Springer (2020)
29. Schneider, S., Dyck, J., Giese, H.: Formal verification of invariants for attributed graph transformation systems based on nested attributed graph conditions. In: ICGT 2020. LNCS, vol. 12150, pp. 257–275. Springer (2020)
30. de Vries, E., Koutavas, V.: Reverse Hoare logic. In: SEFM 2011. LNCS, vol. 7041, pp. 155–171. Springer (2011)
31. Wulandari, G.S., Plump, D.: Verifying graph programs with first-order logic. In: GCM 2020. EPTCS, vol. 330, pp. 181–200 (2020)

Appendix

Proof (Proposition 1; Lemmata 1–2). By induction over the form of E-conditions, following the proof structure for transformations 'App', 'A', and 'L' for the similar assertion language in [24]. □

Proof (Proposition 2). $\Longrightarrow$. Assume that $H \models \mathrm{WPost}(\mathcal{R}, c)$. There exists some $r \in \mathcal{R}$ such that:

$$H \models \mathrm{wpost}(r, c) = \exists \mathtt{x}_1, \cdots \mathtt{x}_n . \exists \emptyset \hookrightarrow R . \mathrm{Dang}(r^{-1}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r, c)).$$

There exists an $h : R^I \hookrightarrow G$ such that $h \models^I \mathrm{Dang}(r^{-1}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r, c))$. Using Proposition 1, there exists a direct derivation from some graph G to H via $r = \langle L \Rightarrow R \rangle$, and by Lemma 2, there exists some $g : L^I \hookrightarrow G$ such that $g \models^I \mathrm{Shift}(r, c)$. By Lemma 1, $G \models c$.

$\Longleftarrow$. Assume that there exists a graph G such that $G \models c$ and $G \Rightarrow_{\mathcal{R}} H$. There exists some $r = \langle L \Rightarrow R \rangle \in \mathcal{R}$ such that $G \Rightarrow_r H$. By the definition of $\models$, Lemma 1, and Lemma 2, there exists some $h : R^I \hookrightarrow G \models^I \mathrm{Right}(r, \mathrm{Shift}(r, c))$. By the definition of direct derivations and Proposition 1, $h \models^I \mathrm{Dang}(r^{-1})$, and thus $h \models^I \mathrm{Dang}(r^{-1}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r, c))$. By the definition of $\models$, $H \models \exists \mathtt{x}_1, \cdots \mathtt{x}_n . \exists R . \mathrm{Dang}(r^{-1}) \wedge \mathrm{Right}(r, \mathrm{Shift}(r, c))$, that is, $H \models \mathrm{wpost}(r, c)$. Being a disjunct of $\mathrm{WPost}(r, c)$, we derive the result $H \models \mathrm{WPost}(r, c)$. □

Proof (Theorem 1). 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, RuleSetFail. The validity of these axioms follows immediately from the definitions of $[\![\mathcal{R}]\!]ok$, $[\![\mathcal{R}]\!]er$, Proposition 1, and Proposition 2.

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 $H.H \models d$, there exists a $G'.G' \models e$ with $(G', G) \in [\![Q]\!]ok$, and for all $G'.G' \models e$, there exists a $G.G \models c$ with $(G, G') \in [\![P]\!]ok$. From the definition of $\models$ and $[\![P; Q]\!]ok$, it then follows that $\models [c]P; Q[ok : d]$. Analogous for case $\vdash [c]P; Q[er : d]$.

SeqFail. Suppose that $\vdash [c]P; Q[er : d]$. By induction, we have $\models [c]P[er : d]$. By definition of $\models$, for all $H.H \models d$, there exists a $G.G \models c$ with $(G, H) \in [\![P]\!]er$. By the definition of $[\![P; Q]\!]er$ and $\models$, it follows that $\models [c]P; Q[er : d]$.

IfElse. Suppose that $\vdash [c]\mathtt{if}\ \mathcal{R}\ \mathtt{then}\ P\ \mathtt{else}\ Q[\epsilon : d]$. By induction, $\models [c \wedge \mathrm{App}(\mathcal{R})]P[\epsilon : d]$ and $\models [c \wedge \neg\mathrm{App}(\mathcal{R})]Q[\epsilon : d]$. From the definition of $\models$, $[\![\mathtt{if}\ \mathcal{R}\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!]\epsilon$, and Proposition 1, we obtain the result that $\models [c]\mathtt{if}\ \mathcal{R}\ \mathtt{then}\ P\ \mathtt{else}\ Q[\epsilon : d]$.

Cons. Suppose that $\vdash [c]P[\epsilon : d]$. By induction, we have $\models [c']P[\epsilon : d']$, $\models d \Longrightarrow d'$, and $\models c' \Longrightarrow c$. It immediately follows that $\models [c]P[\epsilon : d]$.

IterZero. For every graph $G.G \models c \wedge \neg\mathrm{App}(\mathcal{R})$, by Proposition 1, $G \not\Rightarrow_{\mathcal{R}}$, $(G, G) \in [\![\mathcal{R}]\!]er$, and thus $(G, G) \in [\![\mathcal{R}!]\!]ok$. It immediately follows that $\models [c \wedge \neg\mathrm{App}(\mathcal{R})]\mathcal{R}![ok : c \wedge \neg\mathrm{App}(\mathcal{R})]$.

Iter. Suppose that $\vdash [c \wedge \mathrm{App}(\mathcal{R})]\mathcal{R}![ok : d \wedge \neg\mathrm{App}(\mathcal{R})]$. By induction, $\models [c \wedge \mathrm{App}(\mathcal{R})]\mathcal{R}; \mathcal{R}![ok : d \wedge \neg\mathrm{App}(\mathcal{R})$. By definition of $\models$, for all $H.H \models d \wedge \neg\mathrm{App}(\mathcal{R})$, there exists some $G.G \models c \wedge \mathrm{App}(\mathcal{R})$ and $(G, H) \in [\![\mathcal{R}; \mathcal{R}!]\!]ok$. By the definition of $[\![\mathcal{R}!]\!]ok$ and $\models$, we obtain $\models [c \wedge \mathrm{App}(\mathcal{R})]\mathcal{R}![ok : d \wedge \neg\mathrm{App}(\mathcal{R})]$.

IterVar. Suppose that $\vdash [c_0]\mathcal{R}![ok : c_n]$. By induction, $\models [c_{i-1}]\mathcal{R}[ok : c_i]$ for every $0 < i \leq n$ and $\models c_n \Longrightarrow \neg\mathrm{App}(\mathcal{R})$. By the definition of $\models$ and $[\![\mathcal{R}]\!]ok$, for every $G_i.G_i \models c_i$, there exists some $G_{i-1}.G_{i-1} \models c_{i-1}$ and $G_{i-1} \Rightarrow_{\mathcal{R}} G_i$. It follow that there is a sequence of derivations $G_0 \Rightarrow_{\mathcal{R}} \cdots \Rightarrow_{\mathcal{R}} G_n$ with $G_0 \models c_0$ and $G_n \models c_n$. By $\models c_n \Longrightarrow \neg\mathrm{App}(\mathcal{R})$ and Proposition 1, we have $G_n \not\Rightarrow_{\mathcal{R}}$, i.e. $(G_n, G_n) \in [\![\mathcal{R}]\!]er$. Together with the definition of $[\![\mathcal{R}!]\!]ok$, it follows that $\models [c_0]\mathcal{R}![ok : c_n]$. □

Proof (Theorem 2). We prove relative completeness extensionally by showing that for every program P, extensional assertion c, and exit condition $\epsilon \in \{ok, er\}$,

$\vdash [c]P[\epsilon : \text{WPOST}[P, c]]$, where $\text{WPOST}[P, c]$ is an extensional assertion expressing the weakest postcondition relative to P and c, i.e. if $\models [c]P[\epsilon : d]$ for any d, then $d \implies \text{WPOST}[P, c]$ is valid. Relative completeness is obtained by applying the rule of consequence to $\vdash [c]P[\epsilon : \text{WPOST}[P, c]]$.

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

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

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

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

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

Sequential Application ($\epsilon = er$). If the program $P; Q$ fails and the error occurs in Q, then the proof is analogous to the ok case. If the error occurs in P:

$$\begin{aligned} & H \models \text{WPOST}[P; Q, c] \\ \text{iff } & \exists G.G \models c \text{ and } (G, H) \in [\![P; Q]\!]er \\ \text{iff } & \exists G.G \models c \text{ and } (G, H) \in [\![P]\!]er \\ \text{iff } & H \models \text{WPOST}[P, c] \end{aligned}$$

By induction we have $\vdash [c]P[er : \text{WPOST}[P, c]]$, and by SeqFail derive $\vdash [c]P; Q[er : \text{WPOST}[P, c]]$. With Cons we get $\vdash [c]P; Q[er : \text{WPOST}[P; Q, c]]$.

If-then-else. The proof for this case follows a similar structure to sequential composition but treating the two branches separately.

Iteration. Define c_i as $\text{WPOST}[\mathcal{R}, c_{i-1}]$ for every $0 < i \leq n$. We have:

$$\begin{aligned} & G_n \models \text{WPOST}[\mathcal{R}!, c_0] \\ \text{iff } & \exists G_0.G_0 \models c_0 \text{ and } (G_0, G_n) \in [\![\mathcal{R}!]\!]ok \\ \text{iff } & \exists G_0, \cdots G_{n-1}.(G_{i-1}, G_i) \in [\![\mathcal{R}]\!]ok \text{ for all } 0 < i \leq n, \text{ and } (G_n, G_n) \in [\![\mathcal{R}]\!]er \\ \text{iff } & \exists G_1, \cdots G_{n-1}.G_1 \models \text{WPOST}[\mathcal{R}, c_0], \ (G_{i-1}, G_i) \in [\![\mathcal{R}]\!]ok \text{ for all } 1 < i \leq n \\ & \quad \text{and } (G_n, G_n) \in [\![\mathcal{R}]\!]er \\ \text{iff } & G_n \models c_n \text{ and } c_n \implies \neg\text{App}(\mathcal{R}) \end{aligned}$$

By induction, $\vdash [c_{i-1}]\mathcal{R}[ok : \text{WPOST}[\mathcal{R}, c_{i-1}]]$ and thus $\vdash [c_{i-1}]\mathcal{R}[ok : c_i]$. By IterVar and Cons derive the result, $\vdash [c_0]\mathcal{R}![ok : \text{WPOST}[\mathcal{R}!, c_0]]$. □