*Fundamenta Informaticae 118 (2012) 135–175*
*DOI 10.3233/FI-2012-708*
*IOS Press*



# Hoare-Style Verification of Graph Programs

**Christopher M. Poskitt**[*†], **Detlef Plump**
*Department of Computer Science*
*The University of York*
*Deramore Lane, York, YO10 5GH, United Kingdom*
*cposkitt@cs.york.ac.uk; det@cs.york.ac.uk*

**Abstract.** GP (for Graph Programs) is an experimental nondeterministic programming language for solving problems on graphs and graph-like structures. The language is based on graph transformation rules, allowing visual programming at a high level of abstraction. In particular, GP frees programmers from dealing with low-level data structures. In this paper, we present a Hoare-style proof system for verifying the partial correctness of (a subset of) graph programs. The pre- and post-conditions of the calculus are nested graph conditions with expressions, a formalism for specifying both structural graph properties and properties of labels. We show that our proof system is sound with respect to GP's operational semantics and give examples of its use.

## 1. Introduction

Rule-based transformations of graph-like structures are ubiquitous in computer science. Applications of graph transformation to programming languages and software engineering include the semantics and implementation of functional programming languages [26, 27], the specification and analysis of pointer structures [3, 2, 33], the semantics of the Unified Modelling Language [17, 21] and the semantics and analysis of model transformations [36, 9, 5, 15].

Applications to the semantics of languages and the analysis of systems naturally raise the question of how to formally verify properties of graph transformation systems. In recent years, a number of verification approaches have emerged which typically focus on sets of graph transformation rules or graph grammars [32, 4, 20, 6, 10, 19].

[*]This author is grateful to be supported by a scholarship of the Engineering and Physical Sciences Research Council.
[†]Address for correspondence: Department of Computer Science, The University of York, Deramore Lane, York, YO10 5GH, United Kingdom

Graph transformation languages such as PROGRES [34], AGG [35], Fujaba [23] and GrGen [8], however, provide control constructs on top of graph transformation rules for practical problem solving. To give a simple example, consider the problem of reversing the direction of the edges of an input graph. This requires two loops in sequence: the first applies as long as possible a rule which reverses an edge and marks it as reversed (to ensure termination), the second applies as long as possible a rule which removes the auxiliary edge mark.

The challenge to verify programs in practical graph transformation languages has, to the best of our knowledge, not yet been addressed. A first step beyond the verification of plain sets of rules has been made by Habel, Pennemann and Rensink [11] by constructing the weakest preconditions of so-called high-level programs. These programs provide constructs such as sequential composition and as-long-as-possible iteration over sets of conditional graph transformation rules. The authors adopt Dijkstra's approach to program verification: one calculates the weakest precondition for a program and its post-condition, and then needs to prove that the program's precondition implies the weakest precondition. High-level programs fall short of practical graph transformation languages though, in that they cannot calculate with labels (or attributes), a capability which is indispensable for many graph algorithms. For example, computing the shortest path between two nodes requires one to compare and add distances (edge labels). Another drawback of the approach of [11] is that for programs with loops, the generated weakest precondition is infinite.

In this paper we present an approach for verifying programs in the graph programming language GP [28, 22], an experimental nondeterministic language for high-level problem solving in the domain of graphs. GP is based on graph transformation rules and has a simple syntax and semantics, to facilitate formal reasoning about programs. The core of GP consists of just four constructs: single-step application of a set of rules, sequential composition, branching and looping. Our Hoare calculus assumes that the conditions of branching statements and the bodies of loops are sets of conditional rule schemata rather than arbitrary programs.

Instead of adopting the weakest-precondition approach to verification, we follow Hoare's seminal paper [16] and devise a calculus of syntax-directed proof rules. Our proof system aims at human-guided verification and the compositional construction of proofs, assisted by a mechanical theorem prover. This is in line with work on program verification for languages such as Java [30, 18, 37, 25].

The pre- and postconditions of our calculus are *E-conditions*: nested graph conditions in the sense of Habel and Pennemann [10], extended with expressions as labels and assignment constraints for specifying properties of labels. For example, the E-condition $\exists(\ \textcircled{\texttt{x}}\ \ \textcircled{\texttt{y}}\ \mid \texttt{x} * \texttt{x} = \texttt{y})$ expresses that there exists two nodes labelled with some integers $x$ and $y$ such that $x^2 = y$. Such an assertion cannot be finitely expressed with the conditions of [10]. To demonstrate the problem with an even simpler property, consider the E-condition $\exists(\ \textcircled{\texttt{x}}\ \mid \texttt{type}(\texttt{x}) = \texttt{int})$ which requires the existence of a node labelled with some integer. To specify this with a condition in the sense of [10], we would need to include all integers in the label alphabet (violating that paper's requirement that label alphabets are finite) and then resort to the infinite condition

$$\exists(\ \textcircled{0}\ ) \vee \exists(\ \textcircled{1}\ ) \vee \exists(\ \textcircled{-1}\ ) \vee \exists(\ \textcircled{2}\ ) \vee \exists(\ \textcircled{-2}\ ) \vee \ldots$$

The rest of this paper is organised as follows. We briefly review some preliminaries in Section 2, graph transformation in Section 3, and graph programs in Section 4. Following this, we present E-conditions in Section 5, and then use them to define a proof system for GP in Section 6, where its use will be demonstrated by proving properties of graph colouring programs. In Section 7, we formally define

two transformations of E-conditions used in the proof system, then in Section 8 we prove the axiom schemata and inference rules sound in the sense of partial correctness, with respect to GP's operational semantics. Finally, we conclude in Section 9.

This paper is an extended version of the conference paper [31], adding full proofs, and further examples.

## 2. Graphs, Assignments, and Substitutions

Graph transformation in GP is based on the double-pushout approach with relabelling [13]. This framework deals with partially labelled graphs, whose definition we recall below. We consider two classes of graphs, "syntactic" graphs labelled with expressions and "semantic" graphs labelled with (sequences of) integers and strings. We also introduce assignments which translate syntactic graphs into semantic graphs, and substitutions which operate on syntactic graphs.

A *graph* over a label alphabet $\mathcal{C}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where $V_G$ and $E_G$ are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to V_G$ are the *source* and *target* functions for edges, $l_G \colon V_G \to \mathcal{C}$ is the partial node labelling function and $m_G \colon E_G \to \mathcal{C}$ is the (total) edge labelling function. Given a node $v$, we write $l_G(v) = \bot$ to express that $l_G(v)$ is undefined. Graph $G$ is *totally labelled* if $l_G$ is a total function. We write $\mathcal{G}(\mathcal{C})$ for the set of all totally labelled graphs over $\mathcal{C}$, and $\mathcal{G}(\mathcal{C}_\bot)$ for the set of all graphs over $\mathcal{C}$.

Unlabelled nodes will occur only in the interfaces of rules and are necessary in the double-pushout approach to relabel nodes. There is no need to relabel edges as they can always be deleted and reinserted with different labels.

A *graph morphism* $g \colon G \to H$ between graphs $G$ and $H$ 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 $l_H(g(v)) = l_G(v)$ for all $v$ such that $l_G(v) \neq \bot$. Morphism $g$ is an *inclusion* if $g(x) = x$ for all nodes and edges $x$. It is *injective* (*surjective*) if $g_V$ and $g_E$ are injective (surjective). It is an *isomorphism* if it is injective, surjective and satisfies $l_H(g_V(v)) = \bot$ for all nodes $v$ with $l_G(v) = \bot$. In this case $G$ and $H$ are *isomorphic*, which is denoted by $G \cong H$.

We consider graphs over two distinct label alphabets. Graph programs and E-conditions contain graphs labelled with expressions, while the graphs on which programs operate are labelled with (sequences of) integers and character strings. We consider graphs of the first type as syntactic objects and graphs of the second type as semantic objects, and aim to clearly separate these levels of syntax and semantics.

Let $\mathbb{Z}$ be the set of integers and Char be a finite set of characters. We fix the label alphabet $\mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^+$ of all non-empty sequences over integers and character strings.

The other label alphabet we are using consists of expressions according to the EBNF grammar of Figure 1[1], where VarId is a syntactic class[2] of variable identifiers. We write $\mathcal{G}(\text{Exp})$ for the set of all graphs over the syntactic class Exp.

Each graph in $\mathcal{G}(\text{Exp})$ represents a possibly infinite set of graphs in $\mathcal{G}(\mathcal{L})$. The latter are obtained by instantiating variables with values from $\mathcal{L}$ and evaluating expressions. An *assignment* is a partial

[1] This grammar and those in the following sections are ambiguous, as we are not concerned with concrete syntax in this paper. If necessary we use parentheses to disambiguate expressions or programs.

[2] We use the non-terminals of our grammars to denote the syntactic classes of strings that can be derived from them.

| | | |
|---|---|---|
| Exp | ::= | (Term \| String) ['_' Exp] |
| Term | ::= | Num \| VarId \| Term ArithOp Term |
| ArithOp | ::= | '+' \| '-' \| '$*$' \| '/' |
| Num | ::= | ['-'] Digit {Digit} |
| String | ::= | '"' {Char} '"' |

Figure 1. Syntax of expressions

function $\alpha\colon \text{VarId} \to \mathcal{L}$. Given an expression $e$, an assignment $\alpha$ is *well-typed* for $e$ if it is defined for all variables occurring in $e$ and if for each term $t1 \oplus t2$ in $e$, with $\oplus$ in ArithOp, we have $\alpha(\mathtt{x}) \in \mathbb{Z}$ for all variables x occurring in $t1 \oplus t2$. In this case we inductively define the value $e^\alpha \in \mathcal{L}$ as follows. If $e$ is a numeral or a sequence of characters, then $e^\alpha$ is the integer or character string represented by $e$. If $e$ is a variable identifier, then $e^\alpha = \alpha(e)$. Otherwise, if $e$ has the form $t_1 \oplus t_2$ with $\oplus \in$ ArithOp and $t_1, t_2 \in$ Term, then $e^\alpha = t_1^\alpha \oplus_{\mathbb{Z}} t_2^\alpha$ where $\oplus_{\mathbb{Z}}$ is the integer operation represented by $\oplus$. Finally, if $e$ has the form $t\_e_1$ with $t \in \text{Term} \cup \text{String}$ and $e_1 \in$ Exp, then $e^\alpha = t^\alpha e_1^\alpha$ (the concatenation of the sequences $t^\alpha$ and $e_1^\alpha$).

Given a graph $G$ in $\mathcal{G}(\text{Exp})$ and an assignment $\alpha$ that is well-typed for all expressions in $G$, we write $G^\alpha$ for the graph in $\mathcal{G}(\mathcal{L})$ that is obtained from $G$ by replacing each label $e$ with $e^\alpha$ (note that $G^\alpha$ has the same nodes, edges, source and target functions as $G$). If $g\colon G \to H$ is a graph morphism with $G, H \in \mathcal{G}(\text{Exp})$, then $g^\alpha$ denotes the morphism $\langle g_V^\alpha, g_E^\alpha \rangle\colon G^\alpha \to H^\alpha$.

A *substitution* is a partial function $\sigma\colon \text{VarId} \to \text{Exp}$. Given an expression $e$, $\sigma$ is *well-typed* for $e$ if for each term $t_1 \oplus t_2$ in $e$, with $\oplus \in$ ArithOp, we have $\sigma(\mathtt{x}) \in$ Term for all variable identifiers x in $t_1 \oplus t_2$ for which $\sigma$ is defined. In this case, the expression $e^\sigma$ is obtained from $e$ by replacing every variable x for which $\sigma$ is defined with $\sigma(\mathtt{x})$ (if $\sigma$ is not defined for a variable x, then $\mathtt{x}^\sigma = \mathtt{x}$). Given a graph $G$ in $\mathcal{G}(\text{Exp})$ such that $\sigma$ is well-typed for all labels in $G$, we write $G^\sigma$ for the graph in $\mathcal{G}(\text{Exp})$ that is obtained by replacing each label $e$ with $e^\sigma$. If $g\colon G \to H$ is a graph morphism between graphs in $\mathcal{G}(\text{Exp})$, then $g^\sigma$ denotes the morphism $\langle g_V^\sigma, g_E^\sigma \rangle\colon G^\sigma \to H^\sigma$.

Given an assignment $\alpha$ : VarId $\to \mathcal{L}$, the substitution $\sigma_\alpha$ : VarId $\to$ Exp *induced* by $\alpha$ maps every variable x to the expression that is obtained from $\alpha(\mathtt{x})$ by replacing integers and strings with their syntactic counterparts. For example, if $\alpha(\mathtt{x})$ is the integer 23, then $\sigma_\alpha(\mathtt{x})$ is 23 from the syntactic class Num. Consider another example: if $\alpha(\mathtt{x})$ is the sequence $56, a, bc$ , where 56 is an integer and $a$ and $bc$ are strings, then $\sigma_\alpha(\mathtt{x}) = \mathtt{56\_"a"\_"bc"}$. Note that for any variable x, and any two well-typed assignments $\alpha, \alpha'$ for x, $\sigma_\alpha(\mathtt{x})^{\alpha'} = \alpha(\mathtt{x})$.

## 3. Graph Transformation

We briefly review the model of graph transformation underlying GP, the double-pushout approach with relabelling [13]. Our presentation is tailored to GP in that we consider graphs in $\mathcal{G}(\mathcal{L})$, and rules in which the interface consists of unlabelled nodes only.

A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ is a pair of inclusions $K \to L$ and $K \to R$, where $K$ consists of unlabelled nodes only, and $L$ and $R$ are totally labelled graphs over $\mathcal{L}$. Graph $K$ is the *interface* of $r$. Intuitively, an application of $r$ to a graph will remove the items in $L - K$, preserve $K$, add the items in

$R - K$, and relabel the unlabelled nodes in $K$. Given a graph $G$ in $\mathcal{G}(\mathcal{L})$, an injective graph morphism $g\colon L \to G$ is a *match* for $r$ if it satisfies the *dangling condition*: no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case $G$ *directly derives* the graph $H$ in $\mathcal{G}(\mathcal{L})$ that is constructed from $G$ as follows[3]:

1. Remove all nodes and edges in $g(L) - g(K)$.
2. Add disjointly 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 are defined analogously.
3. For each node $v$ in $K$, $l_H(g_V(v))$ becomes $l_R(v)$.

We write $G \Rightarrow_{r,g} H$ (or just $G \Rightarrow_r H$) if $G$ directly derives $H$ as above.

Figure 2 shows an example of a direct derivation. The rule in the upper row is applied to the left graph of the lower row, resulting in the right graph of the lower row. For simplicity, we do not depict edge labels and assume that they are all the same. The node identifiers 1 and 2 in the rule specify the inclusions of the interface. The middle graph of the lower row is an intermediate result (omitted in the above construction). This diagram represents a double-pushout in the category of partially labelled graphs over $\mathcal{L}$.

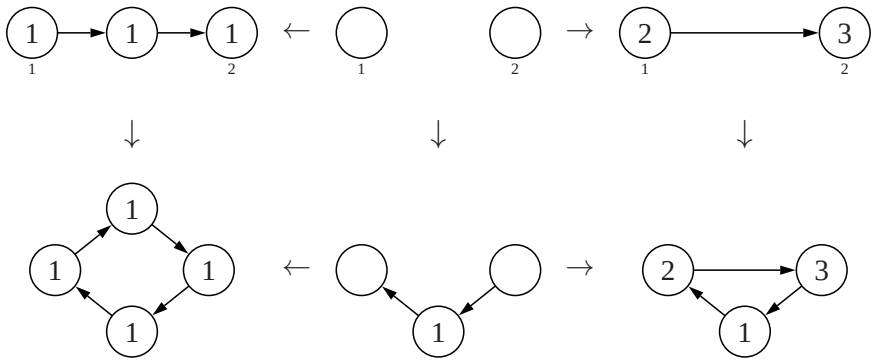



Figure 2. A direct derivation

To define conditional rules, we equip rules with predicates that restrict sets of matches. A *conditional rule* $q = (r, P)$ consists of a rule $r$ and a predicate $P$ on graph morphisms. Given totally labelled graphs $G$, $H$ and a match $g\colon L \to G$ for $q$, we write $G \Rightarrow_{q,g} H$ (or just $G \Rightarrow_q H$) if $P(g)$ holds and $G \Rightarrow_{r,g} H$. For a set of conditional rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if there is some $q$ in $\mathcal{R}$ such that $G \Rightarrow_q H$.

## 4. Graph Programs

We briefly review GP's conditional rule schemata, program syntax, and structural operational semantics. We also give an example program that computes a graph colouring, in order to make clear how the programming language and its features work. Further technical details and examples can be found in [28, 29].

[3] See [13] for an equivalent definition by graph pushouts.

### 4.1. Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of graph programs: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling their application. Rule schemata generalise graph transformation rules as introduced in the previous section, in that labels can contain (sequences of) expressions over parameters of type integer or string. Figure 3 shows a conditional rule schema consisting of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema which are graphs in $\mathcal{G}(\text{Exp})$, the node identifiers 1, 2, 3 specifying which nodes are preserved, and the keyword `where` followed by a rule schema condition.

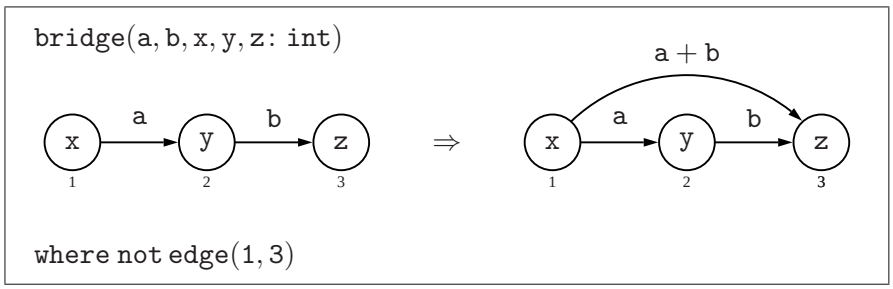



Figure 3. A conditional rule schema

In the GP programming system [22], rule schemata are constructed with a graphical editor. Labels in the left graph comprise only variables and constants (no composite expressions) because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`, where all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the existence of an edge between two nodes in the graph to which the rule schema is applied (and is typically used in negated form). For example, the expression `not edge(1, 3)` in the condition of Figure 3 forbids an edge from node 1 to node 3 when the left graph is matched. The grammar of Figure 4 defines the syntax of rule schema conditions, where Term is the syntactic class defined in Figure 1.

| | | |
|---|---|---|
| BoolExp | ::= | `edge` '(' Node ',' Node ')' \| Term RelOp Term |
| | | \| `not` BoolExp \| BoolExp BoolOp BoolExp |
| Node | ::= | Digit {Digit} |
| RelOp | ::= | '=' \| '\=' \| '>' \| '<' \| '>=' \| '<=' |
| BoolOp | ::= | `and` \| `or` |

Figure 4. Syntax of rule schema conditions

Conditional rule schemata represent possibly infinite sets of conditional graph transformation rules in the sense of the previous section. A rule schema $L \Rightarrow R$ with condition $\Gamma$ represents conditional rules $\langle\langle L^\alpha \leftarrow K \rightarrow R^\alpha\rangle, \Gamma^{\alpha,g}\rangle$, where $K$ consists of the preserved nodes (which in $K$ are unlabelled) and $\Gamma^{\alpha,g}$ is a predicate on graph morphisms $g\colon L^\alpha \rightarrow G$ (see [28, 29]). Thus, applying the rule schema

$\langle L \Rightarrow R, \Gamma \rangle$ to a graph $G$ in $\mathcal{G}(\mathcal{L})$ amounts to:

1. choosing an assignment $\alpha$: VarId $\to \mathcal{L}$,
2. choosing a graph morphism $g : L^\alpha \to G$ that satisfies the dangling condition with respect to $\langle L^\alpha \leftarrow K \to R^\alpha \rangle$,
3. checking the condition $\Gamma^{\alpha,g}$, and
4. applying $\langle L^\alpha \leftarrow K \to R^\alpha \rangle$ with match $g$, in the sense of Section 3.

For example, the upper rows of Figure 5 show the rule schema `bridge` of Figure 3 (without condition) and its instance $\texttt{bridge}^\alpha$, where $\alpha(\texttt{x}) = 0$, $\alpha(\texttt{y}) = \alpha(\texttt{z}) = 1$, $\alpha(\texttt{a}) = 3$ and $\alpha(\texttt{b}) = 2$. The condition of `bridge` evaluates under $\alpha$ to a predicate which is true for a match $g$ of the left-hand graph if and only if there is no edge from $g(1)$ to $g(3)$. The lower rows of Figure 5 show an application of $\texttt{bridge}^\alpha$ by a graph morphism satisfying the predicate.

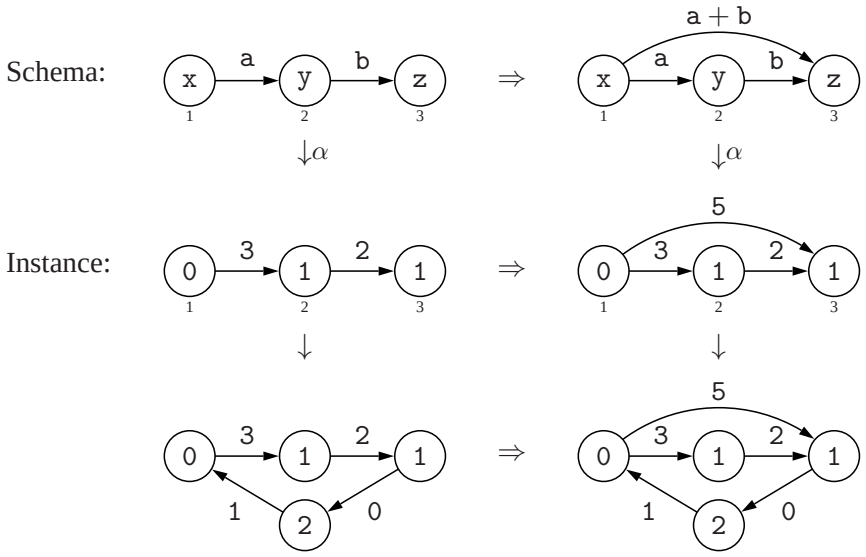



Figure 5. Application of the rule schema `bridge` using instantiation

## 4.2. Abstract Syntax

Figure 6 gives the abstract syntax of graph programs. A program consists of a number of declarations of conditional rule schemata and macros, and exactly one declaration of a main command sequence. The rule schema identifiers (category RuleId) occurring in a call of category RuleSetCall refer to declarations of conditional rule schemata in category RuleDecl (see Section 4.1). The latter category is not defined in the textual syntax because rule schemata are declared graphically in the GP programming system [22].

Macros are a simple means to structure programs and thereby make them more readable. Every program can be transformed into an equivalent macro-free program by replacing macro calls with their associated command sequences (recursive macros are not allowed). This allows us, when defining the semantics of GP, to consider programs as command sequences.

| | | |
|---|---|---|
| Prog | ::= | Decl {Decl} |
| Decl | ::= | RuleDecl \| MacroDecl \| MainDecl |
| MacroDecl | ::= | MacroId '=' ComSeq |
| MainDecl | ::= | `main` '=' ComSeq |
| ComSeq | ::= | Com {';' Com} |
| Com | ::= | RuleSetCall \| MacroCall |
| | | \| `if` ComSeq `then` ComSeq [`else` ComSeq] |
| | | \| ComSeq '!' |
| | | \| `skip` \| `fail` |
| RuleSetCall | ::= | RuleId \| '{' [RuleId {',' RuleId}] '}' |
| MacroCall | ::= | MacroId |

Figure 6. Abstract syntax of GP

A branching command `if` $C$ `then` $P$ `else` $Q$ is executed on a graph $G$ by first executing the program $C$ on $G$. If $C$ can produce a graph, then the program $P$ is executed *on the input graph* $G$. On the other hand, if all executions of $C$ on $G$ end in failure, then the program $Q$ is executed, again, on the input graph $G$.

The commands `skip` and `fail` can be expressed through the other commands (see Section 4.3), hence the core of GP includes only the call of a set of conditional rule schemata (RuleSetCall), sequential composition (';'), the if-then-else statement and as-long-as-possible iteration ('!').

## 4.3. Structural Operational Semantics

GP's formal semantics [29] is given in the style of structural operational semantics (see for example [24]). Inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In our setting, a configuration is either a command sequence together with a graph, just a graph, or the special element fail:

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

Configurations in $\text{ComSeq} \times \mathcal{G}(\mathcal{L})$ represent unfinished computations, given by a command sequence that remains to be executed and a state (a graph), while graphs in $\mathcal{G}(\mathcal{L})$ are proper results of computations. In addition, the element fail represents a failure state.

Each inference rule in Figure 7 consists of a premise and a conclusion separated by a horizontal bar. Both parts contain meta-variables for command sequences and graphs, where $\mathcal{R}$ stands for a call in category RuleSetCall, $C, P, P', Q$ stand for command sequences in category ComSeq, and $G, H$ stand for graphs in $\mathcal{G}(\mathcal{L})$. Given a rule set call $\mathcal{R}$, we write $G \not\Rightarrow_{\mathcal{R}}$ if there is no graph $H$ such that $G \Rightarrow_{\mathcal{R}} H$. Meta-variables are considered to be universally quantified. For example, the rule $[\text{Call}_1]_{\text{SOS}}$ should be read as: "For all $\mathcal{R}$ in RuleSetCall and all $G, H$ in $\mathcal{G}(\mathcal{L})$, $G \Rightarrow_{\mathcal{R}} H$ implies $\langle \mathcal{R}, G \rangle \rightarrow H$."

Figure 7 shows the inference rules for the core constructs of GP. We write $\rightarrow^+$ and $\rightarrow^*$ for the transitive and reflexive-transitive closures of $\rightarrow$. A command sequence $C$ *finitely fails* on a graph $G \in \mathcal{G}(\mathcal{L})$ if (1) there does not exist an infinite sequence $\langle C, G \rangle \rightarrow \langle C_1, G_1 \rangle \rightarrow \ldots$ and (2) for each terminal configuration $\gamma$ such that $\langle C, G \rangle \rightarrow^* \gamma$, $\gamma = \text{fail}$. (A configuration $\gamma$ is *terminal* if there is no

configuration $\delta$ such that $\gamma \to \delta$.) In other words, $C$ finitely fails on $G$ if all computations starting from $\langle C, G\rangle$ eventually end in the configuration fail.

$$[\text{Call}_1]_{\text{SOS}}\ \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R}, G\rangle \to H} \qquad [\text{Call}_2]_{\text{SOS}}\ \frac{G \not\Rightarrow_{\mathcal{R}}}{\langle \mathcal{R}, G\rangle \to \text{fail}}$$

$$[\text{Seq}_1]_{\text{SOS}}\ \frac{\langle P, G\rangle \to \langle P', H\rangle}{\langle P; Q, G\rangle \to \langle P'; Q, H\rangle} \qquad [\text{Seq}_2]_{\text{SOS}}\ \frac{\langle P, G\rangle \to H}{\langle P; Q, G\rangle \to \langle Q, H\rangle}$$

$$[\text{Seq}_3]_{\text{SOS}}\ \frac{\langle P, G\rangle \to \text{fail}}{\langle P; Q, G\rangle \to \text{fail}}$$

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

$$[\text{If}_2]_{\text{SOS}}\ \frac{C \text{ finitely fails on } G}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{Alap}_1]_{\text{SOS}}\ \frac{\langle P, G\rangle \to^{+} H}{\langle P!, G\rangle \to \langle P!, H\rangle} \qquad [\text{Alap}_2]_{\text{SOS}}\ \frac{P \text{ finitely fails on } G}{\langle P!, G\rangle \to G}$$

Figure 7. Inference rules for core commands

The meaning of the remaining GP commands is defined in terms of the meaning of the core commands, see Figure 8. We refer to these commands as *derived* commands.

$[\text{Skip}]_{\text{SOS}}$ $\langle \texttt{skip}, G\rangle \to \langle \texttt{null}, G\rangle$
where `null` is the rule schema $\emptyset \Rightarrow \emptyset$

$[\text{Fail}]_{\text{SOS}}$ $\langle \texttt{fail}, G\rangle \to \langle \{\}, G\rangle$

$[\text{If}_3]_{\text{SOS}}$ $\langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$

Figure 8. Inference rules for derived commands

The meaning of graph programs is summarised by a semantic function $[\![\_]\!]$, which assigns to every program $P$ the function $[\![P]\!]$ mapping an input graph $G$ to the set of all possible results of running $P$ on $G$. The result set may contain, besides proper results in the form of graphs, the special value $\perp$ which indicates a non-terminating or stuck computation. The *semantic function* $[\![\_]\!]\colon \text{ComSeq} \to (\mathcal{G}(\mathcal{L}) \to 2^{\mathcal{G}(\mathcal{L})\cup\{\perp\}})$ is defined by[4]:

$$[\![P]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle P, G\rangle \xrightarrow{+} H\} \cup \{\perp \mid P \text{ can diverge or get stuck from } G\}$$

[4]We write $[\![P]\!]G$ for the application of $[\![P]\!]$ to a graph $G$.

where $P$ *can diverge from* $G$ if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$, and $P$ *can get stuck from* $G$ if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$ (where the rest program $Q$ cannot be executed because no inference rule is applicable).

A program can get stuck only in two situations: either it contains a subprogram `if` $C$ `then` $P$ `else` $Q$ where $C$ both can diverge from some graph and cannot produce a proper result from that graph, or it contains a subprogram $B!$ where the loop's body $B$ possesses the said property of $C$.

## 4.4. Example Program: Node Colouring

We discuss an example program to familiarise the reader with GP's features. This program will be a running example throughout the remainder of the paper.

A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each non-looping edge have different colours. The program `colouring` in Figure 9 produces a colouring for every integer-labelled input graph, recording colours as so-called tags. In general, a tagged label is a sequence of expressions separated by underscores.

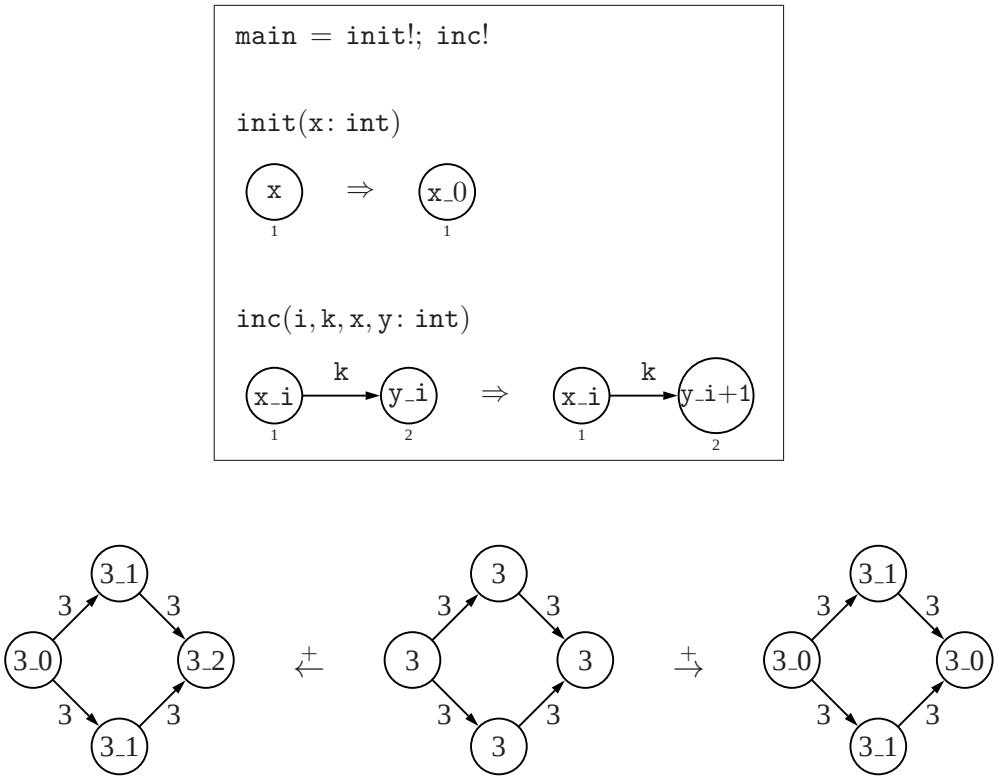



Figure 9. The program `colouring` and two of its executions

The program initially colours each node with 0 by applying the rule schema `init` as long as possible, using the iteration operator '`!`'. It then repeatedly increments the target colour of edges with the same colour at both ends. Note that this process is nondeterministic: Figure 9 shows two executions, one

producing a colouring with two colours, and one producing a colouring with three colours.

It is easy to see that whenever `colouring` terminates, the resulting graph is a correctly coloured version of the input graph. This is because the output cannot contain an edge with the same colour at both of its incident nodes, as then `inc` would have been applied at least one more time. Also, it can be shown that every execution of the program terminates after at most a quadratic number of rule schema applications [28].

## 5. Nested Graph Conditions with Expressions

We introduce nested graph conditions with expressions (or E-conditions) to specify graph properties in the pre- and postconditions of graph programs. E-conditions extend the nested conditions of [10] with expressions for labels, and assignment constraints that restrict the values that can be assigned to variables. E-conditions can be considered as finite representations of (possibly infinite) sets of nested conditions.

**Definition 5.1. (Assignment constraint)**
An *assignment constraint* is a Boolean expression conforming to the grammar in Figure 10. We require that the arguments of the operators `>`, `<`, `>=` and `<=` belong to the syntactic class Term and that the arguments of `=` and `\=` belong to either Term, String, or Exp $-$ (Term $\cup$ String). (See Figure 1 for the definition of Term, String and Exp.) □

| | | |
|---|---|---|
| ACBoolExp | ::= | Exp RelOp Exp \| `not` ACBoolExp |
| | | \| ACBoolExp BoolOp ACBoolExp |
| | | \| `type` '(' Exp ')' '=' Type \| `true` |
| RelOp | ::= | '=' \| '\=' \| '>' \| '<' \| '>=' \| '<=' |
| BoolOp | ::= | `and` \| `or` |
| Type | ::= | `int` \| `string` \| `tagged` |

Figure 10. Syntax of assignment constraints

Given an assignment constraint $\gamma$ and an assignment $\alpha$ well-typed for all expressions in $\gamma$, the value $\gamma^\alpha$ in $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$ is inductively defined as follows. If $\gamma = \texttt{true}$, then $\gamma^\alpha = \texttt{tt}$. Let now $\gamma$ have the form $e_1 \bowtie e_2$ with $\bowtie \in$ RelOp and $e_1, e_2 \in$ Exp. If $\bowtie$ is `=` or `\=`, then $(e_1 \bowtie e_2)^\alpha = \texttt{tt}$ (resp. `ff`) if $e_1^\alpha = e_2^\alpha$, otherwise $(e_1 \bowtie e_2)^\alpha = \texttt{ff}$ (resp. `tt`). If $\bowtie$ is `>`, and $e_1, e_2$ are in Term, then the value of $(e_1 \bowtie e_2)^\alpha$ is the truth value of $e_1^\alpha > e_2^\alpha$. (The cases for when $\bowtie$ is `<`, `>=`, and `<=` are analogous.)

If $\gamma = \texttt{not}\ \gamma_1$ with $\gamma_1 \in$ ACBoolExp, then $\gamma^\alpha = \texttt{tt}$ (resp. `ff`) if $\gamma_1^\alpha = \texttt{ff}$ (resp. `tt`). If $\gamma = \gamma_1 \oplus \gamma_2$ with $\gamma_1, \gamma_2 \in$ ACBoolExp and $\oplus \in$ BoolOp, then $\gamma^\alpha = \gamma_1^\alpha \oplus_{\mathbb{B}} \gamma_2^\alpha$ where $\oplus_{\mathbb{B}}$ is the Boolean operation on $\mathbb{B}$ represented by $\oplus$.

Finally, if $\gamma$ has the form $\texttt{type}(e) = t$ with $e \in$ Exp and $t \in$ Type, then $\gamma^\alpha = \texttt{tt}$ if $\mathrm{t}(e^\alpha) = t$, where the function $\mathrm{t}\colon \mathcal{L} \to$ Type is defined by:

$$\mathrm{t}(l) = \begin{cases} \texttt{int} & \text{if } l \in \mathbb{Z}, \\ \texttt{string} & \text{if } l \in \mathrm{Char}^*, \\ \texttt{tagged} & \text{otherwise.} \end{cases}$$

**Example 5.1. (Assignment constraint)**
Consider the assignment constraint $\gamma = \texttt{a > b and b \= 0 and type(a) = int}$. Let $\alpha_1 = (\texttt{a} \mapsto 5, \texttt{b} \mapsto 1)$ and $\alpha_2 = (\texttt{a} \mapsto 3, \texttt{b} \mapsto 0)$. Then $\gamma^{\alpha_1} = \texttt{tt}$ and $\gamma^{\alpha_2} = \texttt{ff}$. □

Note that variables in assignment constraints do not have a type per se, unlike the variables in GP rule schemata. Rather, `type` can be used to restrict the type(s) that a variable may be instantiated to.

A substitution $\sigma\colon \text{VarId} \to \text{Exp}$ is well-typed for an assignment constraint $\gamma$ if it is well-typed for all expressions in $\gamma$, and if for each $t_1 \bowtie t_2$ with $t_1, t_2 \in \text{Term}$ and $\bowtie \in \text{RelOp} - \{\texttt{=}, \texttt{\=}\}$, we have $\sigma(\texttt{x}) \in \text{Term}$ for all variable identifiers `x` in $t_1, t_2$ for which $\sigma$ is defined. In this case, the assignment constraint $\gamma^\sigma$ is obtained from $\gamma$ by replacing every variable `x` for which $\sigma$ is defined with $\sigma(\texttt{x})$.

**Notation 5.1. (`type`)**
We allow $\texttt{type}(\texttt{x}_1, \ldots, \texttt{x}_n) = \texttt{int}$ to be short for $\texttt{type}(\texttt{x}_1) = \texttt{int and } \ldots \texttt{ and type}(\texttt{x}_n) = \texttt{int}$. □

**Definition 5.2. (E-condition)**
An *E-condition* $c$ over a graph $P$ is of the form true or $\exists(a \mid \gamma, c')$, where $a\colon P \hookrightarrow C$ is an injective[5] graph morphism with $P, C \in \mathcal{G}(\text{Exp})$, $\gamma$ is an assignment constraint, and $c'$ is an E-condition over $C$. Boolean formulae over E-conditions over $P$ yield E-conditions over $P$, that is, $\neg c$ and $c_1 \wedge c_2$ are E-conditions over $P$ if $c, c_1, c_2$ are E-conditions over $P$. □

All substitutions $\sigma$ are well-typed for $c = \text{true}$. In this case we define $c^\sigma = \text{true}$. A substitution $\sigma$ is well-typed for $c = \exists(a \mid \gamma, c')$ if it is well-typed for the graphs in $a$, for $\gamma$, and for $c'$. In this case the application of $\sigma$ to $c$ is defined $c^\sigma = \exists(a^\sigma \mid \gamma^\sigma, (c')^\sigma)$.

The *satisfaction* of E-conditions by injective graph morphisms between graphs in $\mathcal{G}(\mathcal{L})$ is defined inductively. Every such morphism satisfies the E-condition true. An injective graph morphism $s\colon S \hookrightarrow G$ with $S, G \in \mathcal{G}(\mathcal{L})$ *satisfies* the E-condition $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$, denoted $s \models c$, if there exists an assignment $\alpha$ that is well-typed for all expressions in $P, C, \gamma$ and is undefined for variables present only in $c'$, such that $S = P^\alpha$, and such that there is an injective graph morphism $q\colon C^\alpha \hookrightarrow G$ with $q \circ a^\alpha = s$, $\gamma^\alpha = \texttt{tt}$, and $q \models (c')^{\sigma_\alpha}$. Here, $\sigma_\alpha$ is the substitution induced by $\alpha$, which we require to be well-typed for all expressions in $c'$. If such an assignment $\alpha$ and morphism $q$ exist, we say that $s$ *satisfies* $c$ *by* $\alpha$, and write $s \models_\alpha c$. Figure 11 summarises $s \models_\alpha c$ (assuming that $\gamma^\alpha = \texttt{tt}$).

$$\begin{array}{ccc} S = P^\alpha & \overset{a^\alpha}{\hookrightarrow} & C^\alpha \\ & {\scriptstyle s}\searrow \; = \; \swarrow{\scriptstyle q} & \models (c')^{\sigma_\alpha} \\ & G & \end{array}$$

Figure 11. Satisfaction of an E-condition

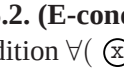
**Remark 5.1. (Induced substitutions)**
In the definition of satisfaction, we apply an induced substitution $\sigma_\alpha$ to the nested E-condition $c'$, before checking that the morphism $q$ satisfies it. This is necessary to enforce equal assignment of variables that appear only in the assignment constraint in different parts of the nesting.

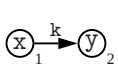
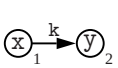
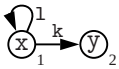
□

[5] We restrict to injective morphisms since GP is restricted to injective matching.

For brevity, we write false for $\neg$true, $\exists(a \mid \gamma)$ for $\exists(a \mid \gamma, \text{true})$, $\exists(a, c')$ for $\exists(a \mid \texttt{true}, c')$, and $\forall(a \mid \gamma, c')$ for $\neg\exists(a \mid \gamma, \neg c')$. In our examples, when the domain of morphism $a\colon P \hookrightarrow C$ can unambiguously be inferred, we write only the codomain $C$. For instance, an E-condition $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow C'))$ can be written as $\exists(C, \exists(C'))$, where the domain of the outermost morphism is the empty graph, and the domain of the nested morphism is the codomain of the encapsulating E-condition's morphism.

An E-condition over a graph morphism whose domain is the empty graph is referred to as an *E-constraint*. We later refer to E-conditions over left- and right-hand sides of rule schemata as *E-app-conditions*.

**Example 5.2. (E-condition)**
The E-condition $\forall($  $\mid$ x > y, $\exists($  $))$ (which is an E-constraint) expresses that every pair of adjacent integer-labelled nodes with the source label greater than the target label has a loop incident to the source node. The unabbreviated version of the condition is as follows:

$\neg\exists(\emptyset \hookrightarrow$  $\mid$ x > y, $\neg\exists($  $\hookrightarrow$  $\mid$ true, true$))$.

□

We write $\not\models$ and $\not\models_\alpha$ in place of $\models$ and $\models_\alpha$, respectively, when the satisfaction relation does not hold for a morphism and E-condition.

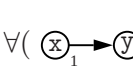
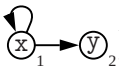
A graph $G$ in $\mathcal{G}(\mathcal{L})$ satisfies an E-condition $c$, denoted $G \models c$, if the morphism $i_G\colon \emptyset \hookrightarrow G$ satisfies $c$. (Note that graphs will only ever satisfy E-constraints.)

The satisfaction of Boolean formulae over E-conditions is defined inductively. We have $s \models \neg c$ if $s \not\models c$, and $s \models c \wedge d$ if $s \models c$ and $s \models d$. Given an assignment $\alpha$, we have $s \models_\alpha \neg c$ if $s \not\models_\alpha c$, and $s \models_\alpha c \wedge d$ if $s \models_\alpha c$ and $s \models_\alpha d$.

Given a substitution $\sigma$, we define $(\neg c)^\sigma = \neg c^\sigma$, and $(c \wedge d)^\sigma = c^\sigma \wedge d^\sigma$ if $\sigma$ is well-typed for $c$ and $c, d$ respectively.

**Notation 5.2. (Unconstrained variables)**
For simplicity, we omit labels of nodes and edges in E-conditions that are unconstrained variables. We leave it implicit that in each graph of the E-condition, each such variable occurs only once and does not occur in any assignment constraint. □

By this convention, we can simplify the E-condition in Example 5.2 to:

$\forall($  $\mid$ x > y, $\exists($  $))$.

Here, it is implicit that the non-looping edge in both graphs are labelled by the same variable (but not x or y), and the looping edge in the nested graph is labelled by another, distinct variable. In Example 5.2, k and l are used, respectively, but the choice of symbols is unimportant.

## 6. A Hoare Calculus for Graph Programs

We present and discuss a system of partial correctness proof rules for GP, in the style of Hoare [1], using E-constraints as the pre- and postconditions. We demonstrate the use of the proof system in two examples.

**Definition 6.1. (Partial correctness)**
A graph program $P$ is *partially correct* with respect to a precondition $c$ and a postcondition $d$ (both of which are E-constraints), if for every graph $G \in \mathcal{G}(\mathcal{L})$, $G \models c$ implies $H \models d$ for every graph $H$ in $[\![P]\!]G$. □

Recall that $[\![\_]\!]$ is GP's semantic function (see Section 4.3), and $[\![P]\!]G$ contains all graphs resulting from executing program $P$ on graph $G$. Note that partial correctness of a program $P$ does not entail that $P$ will terminate on graphs satisfying the precondition.

Given E-constraints $c, d$ and a program $P$, a triple of the form $\{c\}\ P\ \{d\}$ expresses the claim that $P$ is partially correct with respect to precondition $c$ and postcondition $d$. Our proof system in Figure 12 operates on such triples. As in classical Hoare logic [16, 1], we use the proof system to construct proof trees, deriving the desired triple by application of the axiom schemata and inference rules. We let $c, d, e, inv$ range over E-constraints, $P, Q$ over arbitrary command sequences, $r, r_i$ over conditional rule schemata, and $\mathcal{R}$ over sets of conditional rule schemata.

$$\text{[ruleapp]}\ \frac{}{\{\text{Pre}(r,c)\}\ r\ \{c\}} \qquad \text{[nonapp]}\ \frac{}{\{\neg\text{App}(\mathcal{R})\}\ \mathcal{R}\ \{\text{false}\}}$$

$$\text{[ruleset]}\ \frac{\{c\}\ r_1\ \{d\}\ \ldots\ \{c\}\ r_n\ \{d\}}{\{c\}\ \{r_1,\ldots,r_n\}\ \{d\}} \qquad \text{[!]}\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}}$$

$$\text{[comp]}\ \frac{\{c\}\ P\ \{e\} \qquad \{e\}\ Q\ \{d\}}{\{c\}\ P; Q\ \{d\}} \qquad \text{[cons]}\ c \Rightarrow c'\ \frac{\{c'\}\ P\ \{d'\}}{\{c\}\ P\ \{d\}}\ d' \Rightarrow d$$

$$\text{[if}_1\text{]}\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\} \qquad \{c \wedge \neg\text{App}(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

Figure 12. Partial correctness proof rules for GP's core commands

Two transformations — App and Pre — are required in some of the proof rules (formal constructions are given in Section 7). Intuitively, App takes as input a set $\mathcal{R}$ of conditional rule schemata, and transforms it into an E-condition specifying that at least one rule schema in $\mathcal{R}$ is applicable. Pre constructs the weakest precondition such that if $G \models \text{Pre}(r, c)$, and the application of $r$ to $G$ results in a graph $H$, then $H \models c$. The transformation Pre is informally described by the following steps: (1) form a disjunction of right E-app-conditions, accounting for the possible ways in which $c$ and the right-hand side of the rule schema $r$ might overlap, (2) convert the right E-app-condition into a left E-app-condition (i.e. over the left-hand side of $r$), (3) nest this within an E-condition that is quantified over every possible match for $r$ (accounting also for its applicability).

The proof rules share a number of similarities with their counterparts for imperative programming languages, but there are also a number of important differences. The axiom [ruleapp] is as basic to our

proof system as the assignment axiom is to the proof systems of [16, 1]. The [ruleset] rule requires each rule schema to be considered in turn, since one is nondeterministically chosen during program execution. Our [if$_1$] rule considers the applicability of the guard, $\mathcal{R}$, a set of rule schemata, rather than the evaluation of some Boolean expression. We also have the axiom [nonapp], which allows one to infer postconditions of failing programs. The [comp] rule should be familiar from conventional Hoare calculi. The iteration rule [!] is our analogue to classical loop rules; it requires one to prove that the E-condition $inv$ is an invariant of the loop body. Like other proof systems, we have a rule of consequence [cons], which can strengthen preconditions and weaken postconditions. This rule requires one to prove that the E-conditions $c \Rightarrow c'$ and $d' \Rightarrow d$ are valid, which, as in conventional Hoare logic, has to happen outside of the proof system.

Two of the proof rules deal with programs that are restricted in a particular way: both the condition $C$ of a branching command `if` $C$ `then` $P$ `else` $Q$ and the body $P$ of a loop $P!$ must be sets of conditional rule schemata (whereas GP allows arbitrary programs). This restricted language is complete though in that every computable function on graphs (with untagged labels) is computed by some program. This is proved in [12] for a similar language.

When constructing a proof tree for a program containing derived commands, one can simply replace each derived command with the corresponding core command (see Figure 8) and use the proof rules of Figure 12. However, it is more convenient to have proof rules dealing directly with the derived commands, and we give these in Figure 13.

$$[\text{skip}]\ \frac{}{\{c\}\ \texttt{skip}\ \{c\}} \qquad [\text{fail}]\ \frac{}{\{\text{true}\}\ \texttt{fail}\ \{\text{false}\}}$$

$$[\text{if}_2]\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \{d\}}\ c \wedge \neg\text{App}(\mathcal{R}) \Rightarrow d$$

Figure 13. Partial correctness proof rules for GP's derived commands

**Example 6.1. (Colouring)**
Our first example proves a property of the `colouring` program of Figure 9. We prove that if `colouring` is executed on a graph which satisfies the following precondition, then any graph resulting from that execution will satisfy the postcondition:

**Precondition** $\neg\exists(\ \textcircled{a}\ \mid \texttt{not type(a) = int})$
or "every node is integer-labelled"

**Postcondition** $\forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a = b\_c and type(b, c) = int})) \wedge \neg\exists(\ \textcircled{x\_i} \xrightarrow{k} \textcircled{y\_i} \mid \texttt{type(i, k, x, y) = int})$
or "every node label is an integer with a colour attached to it, and nodes linked by integer labelled edges have distinct colours"

Note that the property we are proving does not guarantee that nodes linked by string-labelled edges will have distinct colours. Indeed, they might not, since the rule schemata of `colouring` operate only

on integer labelled nodes and edges. If we strengthened the precondition to require that input graphs contain only integer-labelled edges, then it would be possible to have a more general postcondition (we do not show this here to keep the example simple).

A proof tree proving the above for our `colouring` program is given in Figure 14. The precondition, program, and postcondition form the triple at the root of the tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\mathtt{init}, e)\}\ \mathtt{init}\ \{e\}}[\text{ruleapp}]}{\{e\}\ \mathtt{init}\ \{e\}}[\text{cons}]}{\{e\}\ \mathtt{init!}\ \{e \wedge \neg\mathrm{App}(\{\mathtt{init}\})\}}[!]}{\{c\}\ \mathtt{init!}\ \{d\}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\mathtt{inc}, d)\}\ \mathtt{inc}\ \{d\}}[\text{ruleapp}]}{\{d\}\ \mathtt{inc}\ \{d\}}[\text{cons}]}{\{d\}\ \mathtt{inc!}\ \{d \wedge \neg\mathrm{App}(\{\mathtt{inc}\})\}}[!]}{\{c\}\ \mathtt{init!};\ \mathtt{inc!}\ \{d \wedge \neg\mathrm{App}(\{\mathtt{inc}\})\}}[\text{comp}]$$

$$\begin{aligned}
c &= \neg\exists(\ \textcircled{a}\ \mid \mathtt{not\ type(a) = int})\\
d &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \mathtt{a = b_c\ and\ type(b, c) = int}))\\
e &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \mathtt{type(a) = int}) \vee \exists(\ \textcircled{a}_1 \mid \mathtt{a = b_c\ and\ type(b, c) = int}))\\
\neg\mathrm{App}(\{\mathtt{init}\}) &= \neg\exists(\ \textcircled{x}\ \mid \mathtt{type(x) = int})\\
\neg\mathrm{App}(\{\mathtt{inc}\}) &= \neg\exists(\ \textcircled{x_i} \xrightarrow{k} \textcircled{y_i}\ \mid \mathtt{type(i, k, x, y) = int})\\
\mathrm{Pre}(\mathtt{init}, e) &= \forall(\ \textcircled{x}_1 \mid \mathtt{type(x) = int},\\
&\qquad \forall(\ \textcircled{x}_1\ \textcircled{a}_2, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \mathtt{type(a) = int})\\
&\qquad\quad \vee \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \mathtt{a = b_c\ and\ type(b, c) = int}))\\
&\qquad \wedge\forall(\ \textcircled{x}_1, \exists(\ \textcircled{x}_1 \mid \mathtt{type(x_0) = int})\\
&\qquad\quad \vee\exists(\ \textcircled{x}_1 \mid \mathtt{x_0 = b_c\ and\ type(b, c) = int})))\\
\mathrm{Pre}(\mathtt{inc}, d) &= \forall(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2 \mid \mathtt{type(i, k, x, y) = int},\\
&\qquad \forall(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2\ \textcircled{a}_3, \exists(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2\ \textcircled{a}_3 \mid \mathtt{a = b_c\ and\ type(b, c) = int}))\\
&\qquad \wedge\forall(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2, \exists(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2 \mid \mathtt{x_i = b_c\ and\ type(b, c) = int}))\\
&\qquad \wedge\forall(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2, \exists(\ \textcircled{x_i}_1 \xrightarrow{k} \textcircled{y_i}_2 \mid \mathtt{y_i+1 = b_c\ and\ type(b, c) = int})))
\end{aligned}$$

Figure 14. A proof tree for the program `colouring` of Figure 9

The side conditions arising from applications of [cons] are satisfied as follows (we omit the trivial cases):

$e \Rightarrow \mathrm{Pre}(\mathtt{init}, e)$. For $\mathrm{Pre}(\mathtt{init}, e)$ to be satisfied, for every integer labelled node in the graph, it must be the case that every other node is labelled with either a single integer or an integer and a colour (the second conjunct of the nested E-condition can be disregarded since the node will always be integer labelled). The E-condition $e$ guarantees that every node is integer labelled, so the whole implication must be valid.

$d \Rightarrow \text{Pre}(\texttt{inc}, d)$. For $\text{Pre}(\texttt{inc}, d)$ to be satisfied, for every pair of integer-labelled coloured nodes linked by an integer-labelled edge, it must be the case that any node outside of this pair must be labelled with an integer and a colour (the second and third conjuncts of the nested part can be disregarded since `x`, `y`, and `i` can only be integers). The E-condition $d$ guarantees that every node will be labelled with an integer and a colour, so the whole implication must be valid.

$c \Rightarrow e$. For $e$ to be satisfied, every node must either be labelled with a single integer, or an integer and a colour. The E-condition $c$ guarantees that every node is integer labelled, so the whole implication is valid.

$e \wedge \neg\text{App}(\{\texttt{init}\}) \Rightarrow d$. For $d$ to be satisfied, every node must be labelled with an integer and a colour. The E-condition $e$ guarantees that every node is labelled with a single integer, or an integer and a colour; but $\neg\text{App}(\{\texttt{init}\})$ guarantees that no node is labelled with a single integer. Hence, every node is labelled with an integer and a colour, and the whole implication is valid. □

**Example 6.2. (2-Colouring)**
We now consider the program `2-colouring`, given in Figure 15. The program checks whether a non-empty and connected input graph is 2-colourable and, if this is the case, correctly colours its nodes with 0 or 1. If the graph is not 2-colourable, then the program returns the input graph unmodified. The program first picks an arbitrary integer-labelled node and colours it with 0, before repeatedly colouring uncoloured nodes adjacent to coloured nodes with either 0 or 1, as appropriate. Next, the program attempts to find two adjacent nodes with the same colour (an illegal colouring); if it can find such nodes, every colour is removed.

Note that on an empty input graph, the rule schema `choose` and hence the whole program will fail. Also, if the input graph is disconnected, the program will check 2-colourability only for one of the graph's connected components. These restrictions could be lifted by using the program as the body of an as-long-as-possible loop, but we prefer to keep matters simple in this example.

We prove that if `2-colouring` is executed on a graph which satisfies the following precondition, then any graph resulting from that execution will satisfy the postcondition:

**Precondition** $\neg\exists(\,\textcircled{\texttt{x\_i}} \mid \texttt{type(i, x) = int})$
or "no integer-labelled node is coloured"

**Postcondition** $\neg\exists(\,\textcircled{\texttt{x\_i}} \mid \texttt{type(i, x) = int}) \vee (\forall(\,\textcircled{\texttt{x\_i}}_1 \mid \texttt{type(i, x) = int}, \exists(\,\textcircled{\texttt{x\_i}}_1 \mid \texttt{i = 0 or i = 1})) \wedge \neg\exists(\,\textcircled{\texttt{x\_i}} \xrightarrow{\texttt{a}} \textcircled{\texttt{y\_i}} \mid \texttt{type(a, i, x, y) = int}))$
or "either the precondition holds, or every integer-labelled node with a colour has colour 0 or 1 and no two nodes linked by an integer-labelled edge have the same colour"

A proof tree proving the above for our `2-colouring` program is given in Figure 6. The E-constraints used as the assertions are given in full in Figure 17.

The side conditions arising from applications of [cons] and [if$_2$] are satisfied as follows (we omit the trivial cases):

$c \Rightarrow \text{Pre}(\texttt{choose}, f)$. The first conjunct of the nested part of $\text{Pre}(\texttt{choose}, f)$ is clearly satisfied by any graph. The second conjunct demands that there is not a distinct node from node 1 that is integer-labelled

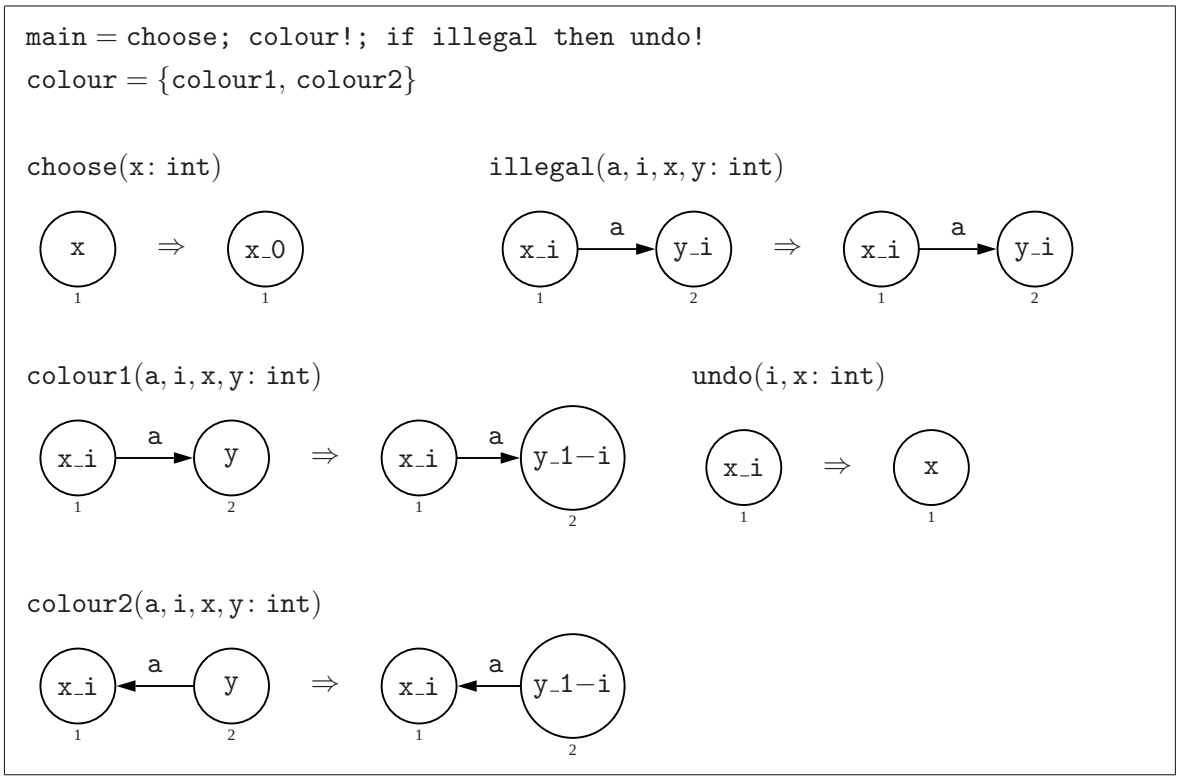



Figure 15. The program 2-colouring

and tagged with a colour. E-condition $c$ expresses that no integer-labelled node is coloured, hence the whole implication is valid.

$e \Rightarrow \text{Pre}(\texttt{colour1}, e)$. For $\text{Pre}(\texttt{colour1}, e)$ to be satisfied by a graph, for every possible match of `colour1`, node 1 must be coloured 0 or 1, and every coloured node outside of the match must be coloured 0 or 1. Additionally, the colour that node 2 will be assigned after the application of `colour1` must also be 0 or 1 (which it will be if `i` is assigned to 0 or 1, by `1-i` in the assignment constraint). The E-condition $e$ is satisfied if and only if every coloured integer-labelled node has colour 0 or 1, so the whole implication must be valid.

$e \Rightarrow \text{Pre}(\texttt{colour2}, e)$. Analogous to the above.

$f \Rightarrow e$. For $e$ to be satisfied, every coloured integer-labelled node in the graph must be coloured with 0 or 1. If $f$ is satisfied, then one such node is coloured 0, but there are not two coloured integer-labelled nodes, i.e. only one node is coloured and it has colour 0. Hence, every coloured node is correctly coloured, and the implication is valid.

$\neg\text{App}(\{\texttt{undo}\}) \Rightarrow c \vee d$. Valid since $\neg\text{App}(\{\texttt{undo}\})$ and $c$ are the same E-conditions.

$e \wedge \neg\text{App}(\{\texttt{illegal}\}) \Rightarrow c \vee d$. Valid since $e \wedge \neg\text{App}(\{\texttt{illegal}\})$ forms the same E-condition as $d$.

□

$$\text{[comp]}\ \frac{\text{Subtree } A \qquad \text{Subtree } B}{\{c\}\ \texttt{choose};\ \{\texttt{colour1}, \texttt{colour2}\}!;\ \texttt{if illegal then undo}!\ \{c \vee d\}}$$

where Subtree $A$ is:

$$\dfrac{\dfrac{\dfrac{}{\{\text{Pre}(\texttt{choose}, f)\}\ \texttt{choose}\ \{f\}}\,\text{[ruleapp]}}{\{c\}\ \texttt{choose}\ \{f\}}\,\text{[cons]} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{Pre}(\texttt{colour1}, e)\}\ \texttt{colour1}\ \{e\}}\,\text{[ruleapp]}}{\{e\}\ \texttt{colour1}\ \{e\}}\,\text{[cons]} \qquad \dfrac{}{\{\text{Pre}(\texttt{colour2}, e)\}\ \texttt{colour2}\ \{e\}}\,\text{[ruleapp]}}{\{e\}\ \{\texttt{colour1}, \texttt{colour2}\}\ \{e\}}\,\text{[ruleset]}}{\{e\}\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e \wedge \neg\text{App}(\{\texttt{colour1}, \texttt{colour2}\})\}}\,\text{[!]}}{\{f\}\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e\}}\,\text{[cons]}}{\{c\}\ \texttt{choose};\ \{\texttt{colour1}, \texttt{colour2}\}!\ \{e\}}\,\text{[comp]}$$

and Subtree $B$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{true}\}\ \texttt{undo}\ \{\text{true}\}}\,\text{[ruleapp]}}{\{\text{true}\}\ \texttt{undo}!\ \{\neg\text{App}(\{\texttt{undo}\})\}}\,\text{[!]}}{\{e \wedge \text{App}(\{\texttt{illegal}\})\}\ \texttt{undo}!\ \{c \vee d\}}\,\text{[cons]}}{\{e\}\ \texttt{if illegal then undo}!\ \{c \vee d\}}\,\text{[if}_2\text{]}$$

Figure 16. A proof tree for the program 2-colouring of Figure 15

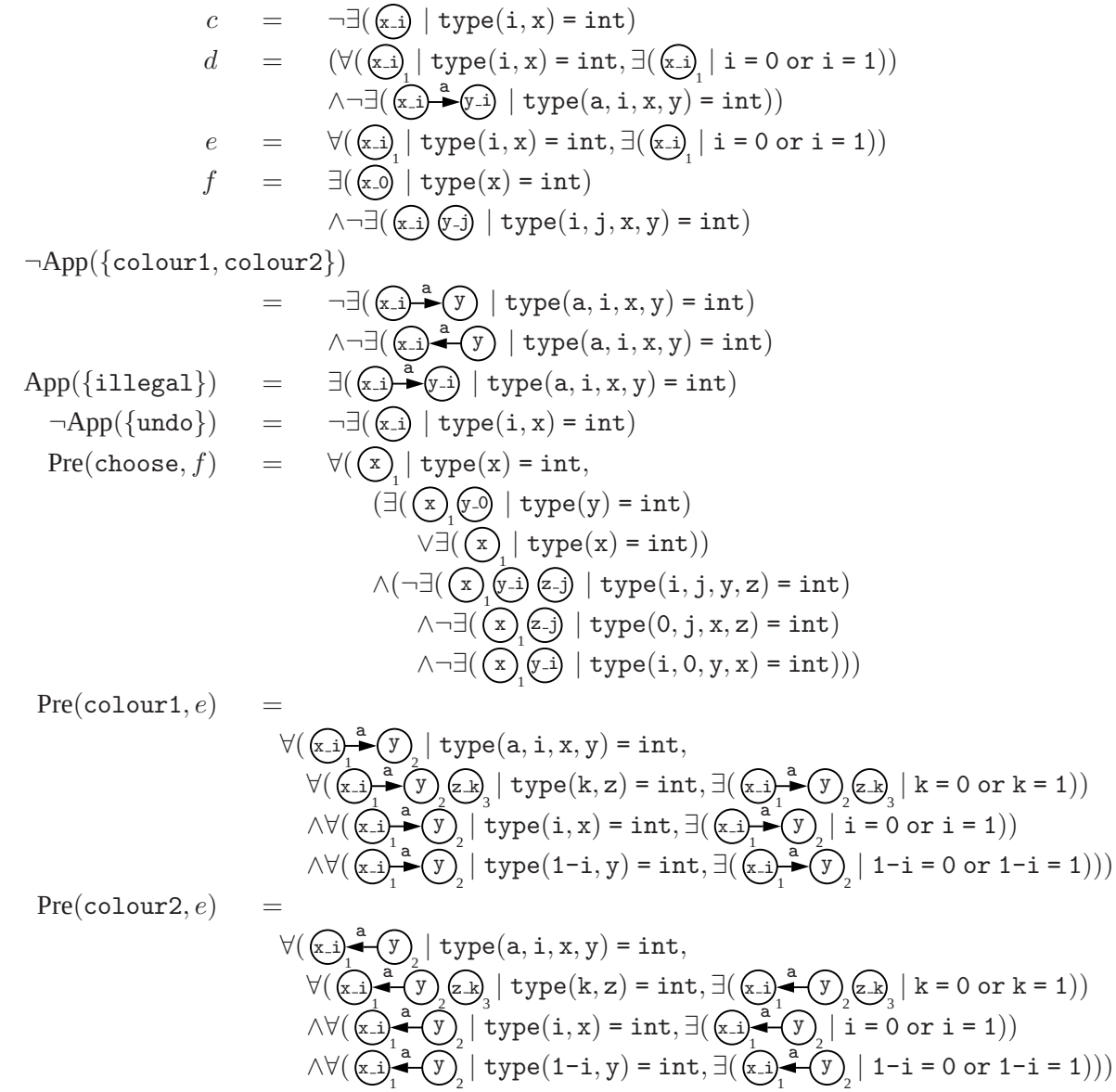



Figure 17. The E-conditions used in the proof tree of Figure 6

# 7. Transformations of E-Conditions

In this section we give formal definitions of the transformations App and Pre, and prove that they are correct. They are adapted from the basic transformations of nested conditions in [10].

We begin in Section 7.1 by stating and proving basic lemmata about the satisfiability of E-conditions to which substitutions have been applied (these lemmata are used in later proofs). In Section 7.2, we define the transformation App and prove that it is correct. In Section 7.3, we build up to a definition and correctness proof of transformation Pre, breaking the transformation steps into the intermediate transformations A and L.

## 7.1. Substitution and Satisfiability Lemmata

In this subsection, we state and prove two lemmata about the satisfiability of E-conditions to which substitutions have been applied. These lemmata are later applied in the correctness proofs of App and Pre.

Lemma 7.1 states that if a morphism satisfies an E-condition by a particular assignment, then it will also satisfy that E-condition after a substitution induced[6] by some (or all) of the assignment's mappings is applied (and vice versa). Intuitively, this is because the induced substitution replaces variables with syntactic representations of the labels in $\mathcal{L}$ that the assignment would have mapped them to.

Lemma 7.2 states that if a morphism satisfies an E-condition to which a substitution has been applied, then it also satisfies the E-condition before the application of that substitution. Intuitively, this is true since one can define a new assignment that incorporates the effect of that substitution.

**Lemma 7.1. (Induced substitutions that preserve satisfiability)**
Let $s\colon P^{\alpha} \hookrightarrow G$ be an injective morphism, where $\alpha$ is a well-typed assignment and $G \in \mathcal{G}(\mathcal{L})$. Let $c$ be an E-condition, and $\alpha'$ be an assignment such that if $\alpha$ is defined for a variable $\mathtt{x}$ then $\alpha'(\mathtt{x}) = \alpha(\mathtt{x})$. Then,

$$s \models_{\alpha'} c \text{ if and only if } s \models_{\alpha'} c^{\sigma_\alpha}.$$

□

**Proof:**
*Case one.* $c = \text{true}$. We have that $c^{\sigma_\alpha} = \text{true}^{\sigma_\alpha} = \text{true}$. All morphisms satisfy true.

*Case two.* $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$. In both the "only if" and "if" directions, the argument follows from the definition of $\sigma_\alpha$, and the fact that for every variable $\mathtt{x}$ that $\alpha$ is defined on, $\alpha'(\mathtt{x}) = \alpha(\mathtt{x})$. Together, we get that $\sigma_\alpha(\mathtt{x})^{\alpha'} = \alpha(\mathtt{x}) = \alpha'(\mathtt{x})$. That is, the substitution ultimately does not change the label in $\mathcal{L}$ obtained by the application of assignment $\alpha'$ to a label. □

**Corollary 7.1.** Let $s\colon P^{\alpha} \hookrightarrow G$ be an injective morphism, where $\alpha$ is a well-typed assignment defined only for variables in $P$, and $G \in \mathcal{G}(\mathcal{L})$. Let $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$ be an E-condition over $P$. Then,

$$s \models c \text{ implies } s \models c^{\sigma_\alpha}.$$

□

[6] Substitutions induced by assignments are defined in Section 2.

**Lemma 7.2. (Discarding a substitution preserves satisfiability)**
Given an injective morphism $s\colon S \hookrightarrow G$ with $S, G \in \mathcal{G}(\mathcal{L})$, an E-condition $c$, sets of variable identifiers $X, Y$, an assignment $\alpha\colon X \to \mathcal{L}$, and a substitution $\sigma\colon Y \to \text{Exp}$,

$$s \models_\alpha c^\sigma \text{ implies } s \models_{\alpha_\sigma} c$$

where $\alpha_\sigma\colon X \to \mathcal{L}$ is defined for all variables $\mathtt{x}$ in $X$ as follows:

$$\alpha_\sigma(\mathtt{x}) = \begin{cases} \alpha(\mathtt{x}) & \text{if } \sigma(\mathtt{x}) \text{ is undefined,} \\ \sigma(\mathtt{x})^\alpha & \text{if } \sigma(\mathtt{x}) \text{ is defined.} \end{cases}$$

□

**Proof:**
By structural induction.

*Induction basis.* Let $c = \text{true}$. Then we have $s \models_\alpha \text{true}^\sigma$ and $s \models_{\alpha_\sigma} \text{true}$. All morphisms satisfy true.

*Induction hypothesis.* The statement holds for $c'$.

*Induction step.* Let $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$. Assume that $s \models_\alpha c^\sigma$. Then we have $(\gamma^\sigma)^\alpha = \mathtt{tt}$ and an injective graph morphism $q\colon (C^\sigma)^\alpha \hookrightarrow G$ with $q \circ (a^\sigma)^\alpha = s$. Now consider $\alpha_\sigma$ from the statement, an assignment which has as its domain all the variables occurring in $P, C$, and $\gamma$. For all variables $\mathtt{x}$ where $\sigma(\mathtt{x})$ is undefined (i.e. variables which are not substituted and thus remain present in $c^\sigma$), we have $\alpha_\sigma(\mathtt{x}) = \alpha(\mathtt{x})$. For all variables $\mathtt{x}$ where $\sigma(\mathtt{x})$ is defined (i.e. variables which are substituted), we have $\alpha_\sigma(\mathtt{x}) = \sigma(\mathtt{x})^\alpha$. Intuitively, $\alpha_\sigma$ has the net effect of applying the substitution $\sigma$ (where defined) before applying the original assignment $\alpha$. This assignment gives us $P^{\alpha_\sigma} = (P^\sigma)^\alpha$, $C^{\alpha_\sigma} = (C^\sigma)^\alpha$, $\gamma^{\alpha_\sigma} = (\gamma^\sigma)^\alpha = \mathtt{tt}$, and thus an injective graph morphism $q'\colon C^{\alpha_\sigma} \hookrightarrow G$ with $q' \circ a^{\alpha_\sigma} = s$ and $q' = q$. By assumption, $q' \models ((c')^\sigma)^{\sigma_\alpha}$, and so there is an assignment $\alpha'$ such that $q' \models_{\alpha'} ((c')^\sigma)^{\sigma_\alpha}$. We assume without loss of generality that $\alpha'$ contains at least the mappings of $\alpha$. Lemma 7.1 and the induction hypothesis together yield $q' \models_{\alpha'_\sigma} c'$. Clearly, $\alpha'_\sigma$ has at least the mappings of $\alpha_\sigma$; using this and the definition of $\models$, we yield $q' \models (c')^{\sigma_{\alpha_\sigma}}$. Putting everything together we get the result that $s \models_{\alpha_\sigma} c$. □

## 7.2. Applicability of Sets of Rule Schemata

In this subsection, we define and prove correct the transformation App, which takes as input a set of rule schemata, and returns an E-condition expressing the weakest property that a graph must satisfy for at least one rule schema in the set to be applicable to it (i.e. at least one rule schema can be applied to the graph). For a rule schema to be applicable to a graph, there must be an opportunity to apply it without violating the dangling condition, and without violating any constraints the rule schema imposes over the instantiation of variables. The definition of App makes use of two intermediate transformations, Dang and $\tau$, which respectively address these requirements.

In Lemma 7.3, we define and prove correct the transformation Dang, which takes as input a rule schema, and returns as output an E-condition which is satisfied by morphisms (from the left-hand side of the rule) that violate the dangling condition.

In Lemma 7.4, we define and prove correct the transformation $\tau$, which takes as input the left-hand side and condition of a rule schema, and returns an E-condition which is satisfied by morphisms (from the left-hand side of the rule) that satisfy the rule schema condition.

**Lemma 7.3. (Dangling condition)**
There is a transformation Dang such that for all rule schemata $r$, and all injective graph morphisms $q\colon L^\alpha \hookrightarrow G$ with $\alpha$ a well-typed assignment,

$$q \models \neg\text{Dang}(r) \text{ if and only if } q \text{ satisfies the dangling condition.}$$

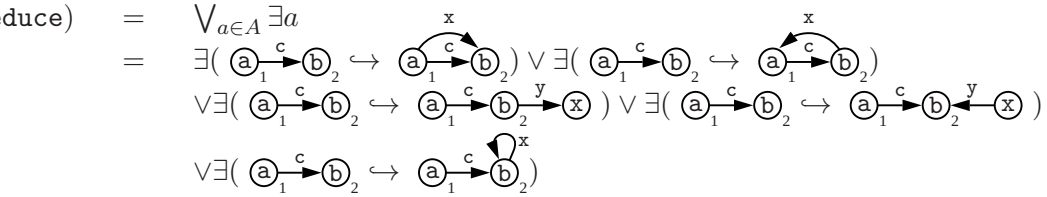
□

The idea of transformation Dang is to generate a disjunction of E-conditions, each one expressing some context (e.g. an edge incident to a node which would be deleted by $r$), which if present around the image of $L$ in $q$, would imply that the morphism is violating the dangling condition.

*Construction.* Define $\text{Dang}(r) = \bigvee_{a\in A} \exists a$, where the index set $A$ ranges over all[7] injective graph morphisms $a\colon L \hookrightarrow L^\oplus$ such that the pair $\langle K \hookrightarrow L, a\rangle$ has no natural pushout[8] complement, and each $L^\oplus$ is a graph that can be obtained from $L$ by adding either (1) a loop labelled by `x`, (2) a single edge between distinct nodes labelled by `x`, or (3) a single node and a non-looping edge incident to that node labelled by `x` and `y` respectively; in all cases, `x`, `y` are variables distinct from each other and all labels in $L$. If the index set $A$ is empty, then $\text{Dang}(r) = \text{false}$.

**Example 7.1.** Consider the rule schema $\texttt{reduce} = \langle$  $\rangle$. Applying Dang to `reduce` yields the following E-condition:

$\text{Dang}(\texttt{reduce}) = \bigvee_{a\in A} \exists a$

$=$ 

□

**Proof:**
*Only if.* Assume that $q \models \neg\text{Dang}(r)$. By definition of $\models$ and the construction of Dang, we have $q \not\models \text{Dang}(r) = \bigvee_{a\in A} \exists a$ where $A$ ranges over morphisms $a : L \hookrightarrow L^\oplus$ such that $\langle K \hookrightarrow L, a\rangle$ has no (natural) pushout complement. Each $L^\oplus$ is obtained from $L$ by adding either (1) a loop, (2) an edge between distinct nodes, or (3) a new node incident to a non-looping edge (i.e. the three possible ways a single edge can be added to $L$). It follows that there is no assignment $\alpha'$ and morphism $q'\colon (L^\oplus)^{\alpha'} \hookrightarrow G$ with $q' \circ a^{\alpha'} = q$. Hence $q$ satisfies the dangling condition, since no node in the image of $q$, that would be deleted by $r$, is incident to an edge in $G$ outside of the match, i.e. the image of some edge from $L^\oplus - L$ in $q'$.

[7] We equate morphisms with isomorphic codomains, so $A$ is finite.
[8] A pushout is *natural* if it is simultaneously a pullback [13].

*If.* Assume that $q : L^\alpha \hookrightarrow G$ is a match for $r$, i.e. it satisfies the dangling condition. Then the pair $\langle K^\alpha \hookrightarrow L^\alpha, q\rangle$ has a pushout complement $D \in \mathcal{G}(\mathcal{L})$. We assume that there is an $a \in A$ such that $\langle K \hookrightarrow L, a\rangle$ has no pushout complement, and some assignment $\alpha'$ such that $q \models_{\alpha'} \exists a$, then derive a contradiction. This assumption gives us a morphism $q' : (L^\oplus)^{\alpha'} \hookrightarrow G$ with $q' \circ a^{\alpha'} = q$. The assignment $\alpha'$ is the same as $\alpha$ other than for having mappings for the additional variables in $L^\oplus$ (i.e. a variable for the extra edge to those in $L$, and possibly a variable for an extra node). Construct $(2)$ (see Figure 18) as a pullback of $(L^\oplus)^{\alpha'} \hookrightarrow G \hookleftarrow D$. By the universal property of pullbacks, there is a morphism $K^\alpha \hookrightarrow (K')^{\alpha'}$ such that the resulting diagrams commute. By the pushout-pullback decomposition, $(1) + (2)$ has a decomposition into pushouts $(1)$ and $(2)$, and $\langle K^\alpha \hookrightarrow L^\alpha, a^{\alpha'}\rangle$ has a pushout complement. Clearly, before the application of assignments $\alpha$ and $\alpha'$, the pair of morphisms $\langle K \hookrightarrow L, a\rangle$ has a pushout complement in $\mathcal{G}(\text{Exp})$. A contradiction. There is no assignment $\alpha'$ such that $q \models_{\alpha'} \bigvee_{a\in A} \exists a = \text{Dang}(r)$, i.e. the result that $q \models \neg\text{Dang}(r)$.

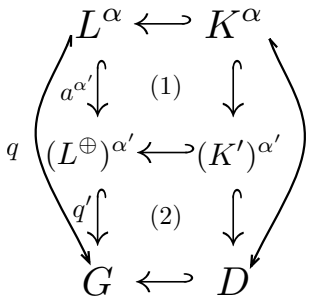



Figure 18. Diagram chasing for a contradiction

□

**Lemma 7.4. (Rule schema condition)**
There is a transformation $\tau$ such that for all rule schemata $r = \langle L \Rightarrow R\rangle$ with rule schema condition $\Gamma$, and all injective graph morphisms $q\colon L^\alpha \hookrightarrow G$ with $\alpha$ a well-typed assignment,

$$q \models_\alpha \tau(L, \Gamma) \text{ if and only if } q \text{ and } \alpha \text{ satisfy the rule schema condition } \Gamma.$$

□

The idea of $\tau$ is to encode the rule schema condition within both the assignment constraints of E-conditions (the morphisms of which are simply the identity morphism on $L$), and the Boolean connectives between them. The exception is the `edge` predicate, which is concerned with the context of $L$ in the graph; this is encoded by an E-condition, the morphism of which has $L$ as its domain, and $L$ as its codomain but with the extra edge demanded by the predicate.

*Construction.* We define $\tau(L, \Gamma)$ inductively (see Figure 4 for the syntax of rule schema conditions). If $\Gamma$ is empty, then $\tau(L, \Gamma) = \text{true}$. If $\Gamma$ has the form $t_1 \bowtie t_2$ with $t_1, t_2$ in Term and $\bowtie$ in RelOp, then $\tau(L, \Gamma) = \exists(L \hookrightarrow L \mid t_1 \bowtie t_2)$. If $\Gamma$ has the form `not` $b$ with $b$ in BoolExp, then $\tau(L, \Gamma) = \neg\tau(L, b)$. If $\Gamma$ has the form $b_1 \oplus b_2$ with $b_1, b_2$ in BoolExp and $\oplus$ in BoolOp, then $\tau(L, \Gamma) = \tau(L, b_1) \oplus_{\wedge,\vee} \tau(L, b_2)$ where $\oplus_{\wedge,\vee}$ is $\wedge$ for `and` and $\vee$ for `or`. Finally, if $\Gamma$ is of the form `edge`$(n_1,n_2)$ with $n_1, n_2$ in Node, then $\tau(L, \Gamma) = \exists(L \hookrightarrow L')$ where $L'$ is a graph equal to $L$, except for an additional edge whose source is the node with identifier $n_1$, whose target is the node with identifier $n_2$, and whose label is a variable distinct from all others in use.

**Example 7.2.** Consider the left-hand side of a rule schema $L =$  and the rule schema condition $\Gamma =$ `a < b and b < c`. Applying the transformation $\tau$ to $L$ and $\Gamma$ yields the following E-condition:

$$\begin{aligned}\tau(L,\Gamma) &= \tau(L, \texttt{a < b}) \wedge \tau(L, \texttt{b < c}) \\ &= \exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \ \mid \texttt{a < b}) \wedge \exists(\ \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \hookrightarrow \textcircled{a}_1 \xrightarrow{c} \textcircled{b}_2 \ \mid \texttt{b < c})\end{aligned}$$

□

**Proof:**
*Only If.* Assume that $q \models_\alpha \tau(L,\Gamma)$. We consider each of the forms that $\Gamma$ can take (using the grammar defined in Figure 4).

Suppose that $\Gamma$ is an empty rule schema condition. Trivially, we have that $q$ and $\alpha$ satisfy the rule schema condition $\Gamma$.

Suppose that $\Gamma$ has the form $t_1 \bowtie t_2$ with $t_1, t_2$ in Term and $\bowtie$ in RelOp. The assumption and construction together give us $q \models_\alpha \exists(L \hookrightarrow L \mid t_1 \bowtie t_2)$ and $(t_1 \bowtie t_2)^\alpha = \texttt{tt}$. Since the assignment constraint is identical to the rule schema condition, we have that $q$ and $\alpha$ satisfy the rule schema condition $\Gamma$.

Suppose that $\Gamma$ has the form `edge`$(n_1,n_2)$ with $n_1, n_2$ in Node. The assumption and construction together give us $q \models_\alpha \exists(L \hookrightarrow L')$ where $L'$ is obtained from $L$ by adding an edge from the node with identifier $n_1$ to the node with identifier $n_2$. There is a morphism $q' : (L')^\alpha \hookrightarrow G$ with $q' \circ (L \hookrightarrow L')^\alpha = q$. Hence the image of $q$ is such that it satisfies the rule schema condition $\Gamma$ that demands the existence of an edge from $n_1$ to $n_2$.

Suppose that $\Gamma$ has the form `not` $b$ with $b$ in BoolExp. The assumption and construction together give us $q \models_\alpha \neg\tau(L,b)$. By the definition of $\models_\alpha$, we have $q \not\models_\alpha \tau(L,b)$. By induction, $q$ and $\alpha$ do not satisfy the rule schema condition $b$. Hence the rule schema condition `not` $b$ is satisfied.

Suppose finally that $\Gamma$ has the form $b_1 \oplus b_2$ with $b_1, b_2$ in BoolExp and $\oplus$ in BoolOp. The assumption and construction together give us $q \models_\alpha \tau(L,b_1) \oplus_{\wedge,\vee} \tau(L,b_2)$. By the definition of $\models_\alpha$ and $\oplus_{\wedge,\vee}$, we have that $q \models_\alpha \tau(L,b_1)$ and (resp. or) $q \models_\alpha \tau(L,b_2)$. It is clear from induction that $q$ and $\alpha$ satisfy the rule schema condition $\Gamma$.

*If.* Assuming that $q$ and $\alpha$ together satisfy the rule schema condition $\Gamma$, one can construct a similar argument in the other direction yielding $q \models_\alpha \tau(L,\Gamma)$. □

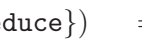
**Proposition 7.1. (Applicability of a set of rule schemata)**
For every set $\mathcal{R}$ of conditional rule schemata, there exists an E-constraint App($\mathcal{R}$) such that for every graph $G \in \mathcal{G}(\mathcal{L})$,

$$G \models \text{App}(\mathcal{R}) \text{ if and only if there is a graph } H \text{ such that } G \Rightarrow_{\mathcal{R}} H.$$

□

The transformation App generates an E-constraint that can only be satisfied by a graph $G$ if at least one of the rule schemata from $\mathcal{R}$ can directly derive a graph $H$ from $G$. The idea is to generate a disjunction of E-constraints from the left-hand sides of the rule schemata, using nesting to handle restrictions on the applicability of the rule schemata (i.e. the dangling condition when deleting nodes, and the rule schema condition restricting possible assignments).

*Construction.* Define $\text{App}(\{\}) = \text{false}$ and $\text{App}(\{r_1, \ldots, r_n\}) = \text{app}(r_1) \vee \ldots \vee \text{app}(r_n)$. For a rule schema $r_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ with rule schema condition $\Gamma_i$, define $\text{app}(r_i) = \exists(\emptyset \hookrightarrow L_i \mid \gamma_{r_i}, \neg\text{Dang}(r_i) \wedge \tau(L_i, \Gamma_i))$ where $\gamma_{r_i}$ is an assignment constraint restricting the types of variables in $r_i$ to the corresponding types in the declaration of $r_i$. For example, if $r_i$ corresponds to the declaration of `inc` (see Figure 9), then $\gamma_{r_i}$ would be the assignment constraint $\texttt{type}(\texttt{i}, \texttt{k}, \texttt{x}, \texttt{y}) = \texttt{int}$.

**Example 7.3.** Consider the rule schema $\texttt{reduce}(\texttt{a}, \texttt{b}, \texttt{c} : \texttt{int}) = \langle$  $\rangle$ with rule schema condition $\Gamma = \texttt{a} < \texttt{b}$ and $\texttt{b} < \texttt{c}$. Applying App to `reduce` yields the following E-condition:

$$
\begin{aligned}
\text{App}(\{\texttt{reduce}\}) &= \text{app}(\texttt{reduce}) \\
&= \exists(\emptyset \hookrightarrow \text{} \mid \texttt{type}(\texttt{a},\texttt{b},\texttt{c}) = \texttt{int}, \\
&\qquad \neg\text{Dang}(\texttt{reduce}) \wedge \tau(\text{}, \Gamma)) \\
&= \exists(\text{} \mid \texttt{type}(\texttt{a},\texttt{b},\texttt{c}) = \texttt{int}, \\
&\qquad (\neg\exists(\text{}) \wedge \neg\exists(\text{}) \wedge \neg\exists(\text{  }) \\
&\qquad \wedge\neg\exists(\text{}) \wedge \neg\exists(\text{})) \\
&\qquad \wedge(\exists(\text{} \mid \texttt{a} < \texttt{b}) \wedge \exists(\text{} \mid \texttt{b} < \texttt{c})))
\end{aligned}
$$

□

**Proof:**
Define $i_G \colon \emptyset \hookrightarrow G$.

*Only if.* Assume that $G \models \text{App}(\mathcal{R})$. By the definitions of $\models$ and App, we have that $i_G \models \text{App}(\mathcal{R}) = \text{app}(r_1) \vee \ldots \vee \text{app}(r_n)$ where $r_i \in \mathcal{R}$. By assumption, there is a rule schema $r_i : \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ in $\mathcal{R}$ with rule schema condition $\Gamma_i$, and a well-typed assignment $\alpha$ such that $i_G \models_\alpha \text{app}(r_i) = \exists(a : \emptyset \hookrightarrow L_i \mid \gamma_{r_i}, \neg\text{Dang}(r_i) \wedge \tau(L_i, \Gamma_i))$. There exists an injective graph morphism $q : L_i^\alpha \hookrightarrow G$ with $q \circ a^\alpha = i_G$, $q \models \neg\text{Dang}(r_i)^{\sigma_\alpha}$ and $q \models \tau(L_i, \Gamma_i)^{\sigma_\alpha}$. By Lemma 7.2, we have $q \models \neg\text{Dang}(r_i)$ and $q \models \tau(L_i, \Gamma_i)$. By Lemma 7.3, the dangling condition is satisfied by $q$, and by Lemma 7.4, $q$ satisfies the rule schema condition $\Gamma$. Putting everything together, and by the definition of rule schema application, $q$ is a match for $r_i$. Hence there is a direct derivation $G \Rightarrow_{r_i,q} H$ for some graph $H \in \mathcal{G}(\mathcal{L})$. Since $r_i \in \mathcal{R}$, we get the result that there exists a graph $H$ such that $G \Rightarrow_{\mathcal{R}} H$.

*If.* Assume that there exists a graph $H$ such that $G \Rightarrow_{\mathcal{R}} H$. Then there is a rule schema $r \in \mathcal{R}$ such that $G \Rightarrow_r H$. Hence there is some instantiation of the variables in $L$ and $\Gamma$ by an assignment $\alpha$ that gives a match $q : L^\alpha \hookrightarrow G$ for $r$ and $\Gamma^\alpha = \texttt{tt}$. By Lemma 7.4, we have $q \models_\alpha \tau(L, \Gamma)$, and then with Lemma 7.1 get $q \models \tau(L, \Gamma)^{\sigma_\alpha}$. The morphism $q$ is guaranteed to satisfy the dangling condition since direct derivations are constructed from two natural pushouts. With Lemma 7.3 this gives us that $q \models \neg\text{Dang}(r)$. Since $\alpha$ is defined only for variables in $L$ (variables appearing in $\Gamma$ must also appear in $L$, by the definition of rule schema conditions), we get from Corollary 7.1 that $q \models \neg\text{Dang}(r)^{\sigma_\alpha}$. By the definition of $\models$, we get $q \models \neg\text{Dang}(r)^{\sigma_\alpha} \wedge \tau(L, \Gamma)^{\sigma_\alpha}$. From the construction we have that $\gamma_r$ only restricts the instantiations of variables to the types that were declared in $r$, so clearly we have that $\gamma_r^\alpha = \texttt{tt}$. Bringing this all together, we have that $i_G \models_\alpha \text{app}(r) = \exists(\emptyset \hookrightarrow L \mid \gamma_r, \neg\text{Dang}(r) \wedge \tau(L, \Gamma))$ since $q \circ (\emptyset \hookrightarrow L^\alpha) = i_G$. As $\text{app}(r)$ is a disjunct of $\text{App}(\mathcal{R})$, we get $i_G \models_\alpha \text{App}(\mathcal{R})$, and by definition of $\models$, we get the result that $G \models \text{App}(\mathcal{R})$. □

## 7.3. Transformation of Postconditions into Preconditions

In this subsection, we define and prove correct the transformation Pre, which takes as input a rule schema and a postcondition (in the form of an E-condition), returning an E-condition that if satisfied by a graph, guarantees that any graph resulting from the application of the rule schema will satisfy the postcondition. The transformation Pre makes use of two intermediate transformations, A and L, which are adapted from the basic transformations of nested conditions described by Habel and Pennemann in [10].

In Proposition 7.2, we define and prove correct the transformation A, which transforms an E-constraint into an E-app-condition over the right-hand side of a rule schema.

In Proposition 7.3, we define and prove correct the transformation L, which transforms an E-app-condition over the right-hand side of a rule schema into an E-app-condition over the left-hand side of that same rule schema.

**Remark 7.1.** In the transformations that follow, there are statements of the form $s\colon P^{\alpha} \hookrightarrow G \models \exists(a\colon P \hookrightarrow C \mid \gamma, c')$ for $P, C \in \mathcal{G}(\text{Exp})$ and $G \in \mathcal{G}(\mathcal{L})$, i.e. the domain of $s$ is some instantiation of the graph in the domain of $a$. For the sake of simplicity, if such a morphism does satisfy such an E-condition, we often assume $\alpha$ to be the assignment by which the E-condition is satisfied, i.e. $s \models_{\alpha} \exists(a \mid \gamma, c')$. We can do this without loss of generality, since we can always "overload" $\alpha$ with mappings for variables not present in $P$ but present in $C, \gamma$, without affecting the graph resulting from the application of $\alpha$ to $P$. □

**Remark 7.2.** E-conditions, by definition, contain arbitrary expressions as the labels of their graphs. We can however restrict ourselves (without loss of generality) to considering E-conditions in which nodes and edges are labelled only by (sequences of) distinct variables, since the variables can be equated with the original expressions in the assignment constraint. For example, the E-condition $\exists($  $)$ can be rewritten as the equivalent E-condition $\exists($  $\mid$ `a = x*x`$)$. □

**Proposition 7.2. (From E-constraints to E-app-conditions)**
Let $c$ be an E-constraint, the graphs of which are labelled by (sequences of) distinct variables. There is a transformation A such that for all rule schemata $r = \langle L \Rightarrow R \rangle$ sharing no variables with $c$[9], and all injective graph morphisms $h\colon R^{\alpha} \hookrightarrow H$ with $H \in \mathcal{G}(\mathcal{L})$ and $\alpha$ a well-typed assignment,

$$h \models \text{A}(r, c) \text{ if and only if } H \models c.$$

□

The idea of A is to consider a disjunction of all possible "overlappings" of $R$ and the graphs of the E-constraint. Since distinct labels on the syntactic level (Exp) can be instantiated to equal labels on the semantic level ($\mathcal{L}$), the transformation applies substitutions to variables to facilitate overlappings of nodes and edges on the syntactic level. Intuitively, an E-condition resulting from A asserts that the property described by the E-constraint still holds, but makes this assertion within the context of $R$.

*Construction.* All graphs used in the construction of the transformation belong to the class $\mathcal{G}(\text{Exp})$. For E-constraints $c = \exists(a\colon \emptyset \hookrightarrow C \mid \gamma, c')$ and rule schemata $r$, define $\text{A}(r, c) = \text{A}'(i_R\colon \emptyset \hookrightarrow R, c)$. For

[9] It is always possible to replace the label variables in $c$ with new ones that are distinct from those in $r$.

injective graph morphisms $p\colon P \hookrightarrow P'$, and E-conditions over $P$,

$$\begin{aligned} \mathrm{A}'(p, \mathrm{true}) &= \mathrm{true}, \\ \mathrm{A}'(p, \exists(a\colon P \hookrightarrow C \mid \gamma, c')) &= \bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_\sigma} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (c')^\sigma)). \end{aligned}$$

The second line of the equations relies on the following. Construct the pushout $(1)$ of $p$ and $a$ (see Figure 19) leading to injective graph morphisms $a' : P' \hookrightarrow C'$ and $q : C \hookrightarrow C'$. The finite double disjunction $\bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_\sigma}$ ranges first over substitutions from $\Sigma$, which we define to contain (1) the empty substitution[10], and (2) all possible substitutions of the form $(\mathtt{a}_1 \mapsto \beta_1, \ldots, \mathtt{a}_k \mapsto \beta_k)$ where each $\mathtt{a}_i$ is a distinct label variable from $C$ that is not also in $P$ or $P'$, and each $\beta_i$ is some label from $P'$ (if a node or label in $P'$ is tagged, then $\beta_i$ may be a portion, or the entirety of, that sequence). For each $\sigma \in \Sigma$, the double disjunction then ranges over every surjective graph morphism $e\colon (C')^\sigma \to E$ such that $b = e \circ (a')^\sigma$ and $s = e \circ q^\sigma$ are injective graph morphisms. The set $\varepsilon_\sigma$ is the set of such surjective graph morphisms for a particular $\sigma$, the codomain of which we consider up to isomorphism. Given a surjective graph morphism $e_1 : (C')^{\sigma_1} \to E_1$, $E_1$ is considered redundant and is excluded from the disjunction if there exists a surjective graph morphism $e_2 : (C')^{\sigma_2} \to E_2$, such that $E_2 \ncong E_1$, and there exists some $\sigma \in \Sigma$ such that $E_2^\sigma \cong E_1$.

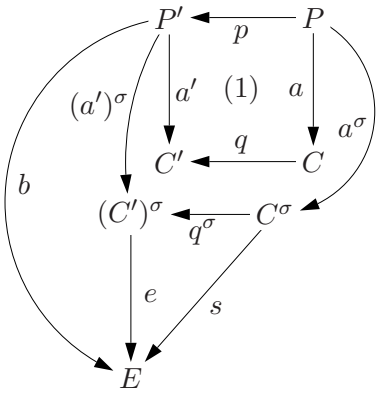



Figure 19. Construction of $\mathrm{A}'$

Note that the special form of the substitutions in $\Sigma$ means that for any $\sigma \in \Sigma$, $P^\sigma = P$, and $(P')^\sigma = P'$. Note also that $b$ and $s$ are jointly surjective; the idea is that each $E$ contains an image of both $P'$ and $C^\sigma$, with the substitutions equating labels on the syntactic level and thus facilitating $E$s in which nodes and edges are overlapping (needed for expressing how the rule schema interacts with the original E-constraint).

The transformations $\mathrm{A}, \mathrm{A}'$ are extended for Boolean formulae over E-conditions in the usual way, that is, $\mathrm{A}(r, \neg c) = \neg\mathrm{A}(r, c)$, and $\mathrm{A}(r, c_1 \wedge c_2) = \mathrm{A}(r, c_1) \wedge \mathrm{A}(r, c_2)$ (analogous for $\mathrm{A}'$).

**Example 7.4.** Let $r = \mathtt{init}$ (see Figure 9), and

$$c = \forall(\ \textcircled{\mathtt{a}}_1, \exists(\ \textcircled{\mathtt{a}}_1 \mid \mathtt{type(a) = int}) \vee \exists(\ \textcircled{\mathtt{a}}_1 \mid \mathtt{a = b\_c\ and\ type(b, c) = int})),$$

[10] That is, a substitution that replaces no variables.

that is, "every node is labelled by either an integer or a sequence of two integers". For brevity in what follows, we define $c'_1 = \exists(\ \textcircled{a}_1 \mid \texttt{type(a) = int})$ and $c'_2 = \exists(\ \textcircled{a}_1 \mid \texttt{a = b\_c and type(b, c) = int})$. With the definition of $\forall$, we yield:

$$c \equiv \neg\exists(\ \textcircled{a}_1, \neg c'_1 \wedge \neg c'_2).$$

Now, applying transformation A to $r$ and $c$, we get:

$$\begin{array}{lcl}
\mathrm{A}(r,c) & = & \neg\mathrm{A}(r, \exists(\ \textcircled{a}_1, \neg c'_1 \wedge \neg c'_2)) \\
& = & \neg\mathrm{A}'(\emptyset \hookrightarrow \textcircled{x_0}_1, \exists(\ \textcircled{a}_1, \neg c'_1 \wedge \neg c'_2)) \\
& = & \neg(\bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (\neg c'_1 \wedge \neg c'_2)^\sigma))) \\
& = & \neg(\exists(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1\,\textcircled{a}_2, \mathrm{A}'(s_1, \neg c'_1 \wedge \neg c'_2)) \\
& & \vee\exists(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1, \mathrm{A}'(s_2, (\neg c'_1 \wedge \neg c'_2)^{(\mathtt{a}\mapsto\mathtt{x_0})}))) \\
& = & \neg(\exists(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1\,\textcircled{a}_2, \neg\mathrm{A}'(s_1, c'_1) \wedge \neg\mathrm{A}'(s_1, c'_2)) \\
& & \vee\exists(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1, \neg\mathrm{A}'(s_2, (c'_1)^{(\mathtt{a}\mapsto\mathtt{x_0})}) \wedge \neg\mathrm{A}'(s_2, (c'_2)^{(\mathtt{a}\mapsto\mathtt{x_0})}))) \\
& = & \neg(\exists(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1\,\textcircled{a}_2, \\
& & \quad \neg\exists(\ \textcircled{x_0}_1\,\textcircled{a}_2 \mid \texttt{type(a) = int}, \mathrm{A}'(s_{11}, \mathrm{true})) \\
& & \quad \wedge \neg\exists(\ \textcircled{x_0}_1\,\textcircled{a}_2 \mid \texttt{a = b_c and type(b, c) = int}, \mathrm{A}'(s_{11}, \mathrm{true}))) \\
& & \vee\exists(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1, \\
& & \quad \neg\exists(\ \textcircled{x_0}_1 \mid \texttt{type(x_0) = int}, \mathrm{A}'(s_{21}, \mathrm{true})) \\
& & \quad \wedge\neg\exists(\ \textcircled{x_0}_1 \mid \texttt{x_0 = b_c and type(b, c) = int}, \mathrm{A}'(s_{21}, \mathrm{true})))) \\
& = & \forall(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1\,\textcircled{a}_2, \\
& & \quad \exists(\ \textcircled{x_0}_1\,\textcircled{a}_2 \mid \texttt{type(a) = int}) \vee\ \exists(\ \textcircled{x_0}_1\,\textcircled{a}_2 \mid \texttt{a = b_c and type(b, c) = int})) \\
& & \wedge\forall(\ \textcircled{x_0}_1 \hookrightarrow \textcircled{x_0}_1, \\
& & \quad \exists(\ \textcircled{x_0}_1 \mid \texttt{type(x_0) = int}) \vee \exists(\ \textcircled{x_0}_1 \mid \texttt{x_0 = b_c and type(b, c) = int})),
\end{array}$$

where $\Sigma = \{(), (\mathtt{a} \mapsto \mathtt{x\_0}), (\mathtt{a} \mapsto \mathtt{x}), (\mathtt{a} \mapsto \mathtt{0})\}$ (here, $()$ denotes the empty substitution that replaces no variables) and the particular instances of diagrams from the construction of $\mathrm{A}'$ are as in Figure 20. Note that both $(\mathtt{a} \mapsto \mathtt{x})$ and $(\mathtt{a} \mapsto \mathtt{0})$ can only yield redundant E-conditions and hence are excluded from the disjunction above. Note also that because $c'_1, c'_2$ contain only identity morphisms (and hence their codomains do not introduce new variables), each instance of $\mathrm{A}'(s_i, c'_j)$ for $i, j \in \{1, 2\}$ ranges over only one substitution: the empty substitution.

The E-app-condition arising from $\mathrm{A}(r, c)$ can be read as follows: "(1) every node that is not in the image of the right-hand side of $r$ is either labelled by an integer or a sequence of two integers, and (2) every node that is in the image of the right-hand side of $r$ is either labelled by an integer or a sequence of two integers". Note that we could already apply simplifications at this stage (e.g. the disjunct $\exists(\ \textcircled{x\_0}_1 \mid \texttt{type(x\_0) = int})$ can safely be discarded since it is unsatisfiable). However, we will wait until the end of this running example (i.e. once $\mathrm{Pre}(r, c)$ is given) before applying any, so that the effects of the transformations can be followed more easily. □

We remark that in the worst case, transformation A can result in a factorial blow-up of the size of an E-condition. One can construct an example where graphs $P'$ and $C$ in Figure 19 both have $n$ nodes and

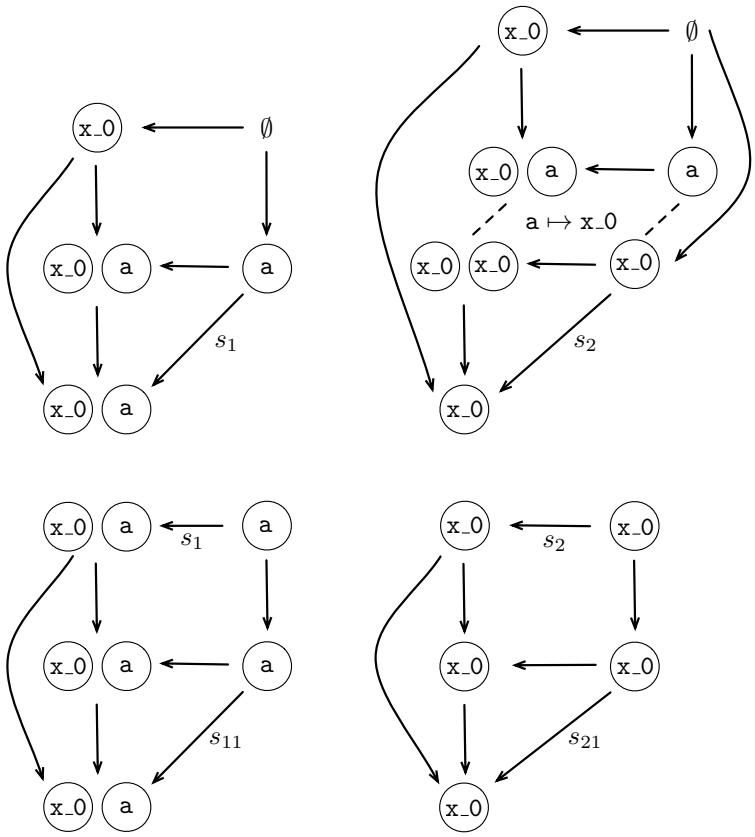



Figure 20. Instances of diagrams from the construction of A′

$n$ edges, and there are more than $n!$ pairwise non-isomorphic graphs $E$ that satisfy the conditions of the construction of Proposition 7.2.

In order to prove Proposition 7.2, we first prove a lemma stating that an E-condition $c$ over $P$ can be shifted along an injective graph morphism $p$ which has as its domain $P$. The proof is very similar to the proof of Lemma 3 in [10]. On the one hand, it is simplified since we consider only injective graph morphisms in our E-conditions, but on the other, it is made more complicated by the separation of graphs over the syntactic and semantic label alphabets.

**Lemma 7.5. (Shifting E-conditions over morphisms)**
Let $P \in \mathcal{G}(\text{Exp})$. Let $c$ be an E-condition true, or $\exists(a : P \hookrightarrow C \mid \gamma, c')$ in which the nodes and edges of each graph (except those also in $P$) are labelled by (sequences of) distinct variables. For all injective graph morphisms $p : P \hookrightarrow P'$ and $p'' : (P')^{\alpha} \hookrightarrow H$ where $P' \in \mathcal{G}(\text{Exp})$, $H \in \mathcal{G}(\mathcal{L})$, and $\alpha$ is a well-typed assignment,

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

□

**Proof:**
We proceed by structural induction, taking a similar approach to the proof of Lemma 3 in [10].

*Induction basis.* Let $c = \text{true}$. Then we have $p'' \models \mathrm{A}'(p, \text{true}) = \text{true}$ and $p'' \circ p^\alpha \models \text{true}$. All morphisms satisfy true.

*Induction hypothesis.* The statement holds for E-condition $c'$.

*Induction step.* Let $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$. For clarity, Figure 21 provides a diagram of the construction before and after the application of assignment $\alpha$.

*Only if.* Assume that $p'' \models \mathrm{A}'(p, c)$. We assume without loss of generality that it does so by $\alpha$ (see Remark 7.1), i.e. $p'' \models_\alpha \mathrm{A}'(p, c) = \bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b\colon P' \hookrightarrow E \mid \gamma^\sigma, \mathrm{A}'(s\colon C^\sigma \hookrightarrow E, (c')^\sigma))$. There exists at least one $\sigma \in \Sigma$ and one $e \in \varepsilon_\sigma$ such that $p'' \models_\alpha \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (c')^\sigma))$. By definition of $\models_\alpha$, there exists an injective graph morphism $q''\colon E^\alpha \hookrightarrow H$ with $p'' = q'' \circ b^\alpha$. Define $q' = q'' \circ s^\alpha$ and $p' = p'' \circ p^\alpha$, both of which are injective since injectivity is closed under composition. By construction, $a' \circ p = q \circ a$ is a pushout. Since $\sigma$ only replaces variables introduced in $C$, and thus also present in $C'$ but not $P$ or $P'$, we have that $P^\sigma = P$, $(P')^\sigma = P'$, and $q^\sigma \circ a^\sigma = (a')^\sigma \circ p$ is a pushout. Clearly, applying $\alpha$ to the morphisms of this pushout results in a pushout of graphs from $\mathcal{G}(\mathcal{L})$. By construction, we have $b = e \circ (a')^\sigma$ and $s = e \circ q^\sigma$. With everything together, we derive that $p'' \circ p^\alpha = p' = q' \circ (a^\sigma)^\alpha$ and get $p' = p'' \circ p^\alpha \models_\alpha \exists(a^\sigma \mid \gamma^\sigma)$.

Now, we want to apply the induction hypothesis, but first must rewrite the assumption into an appropriate form (without substitutions). The assumption gives us $q'' \models \mathrm{A}'(s\colon C^\sigma \hookrightarrow E, (c')^\sigma)^{\sigma_\alpha}$; with Lemma 7.2 we yield $q'' \models \mathrm{A}'(s, (c')^\sigma)$. By the construction, $\sigma$ is undefined for variables not present in $C$. Since $C^\sigma$ forms the common domain of the pushout in the construction of $\mathrm{A}'$, the E-condition generated by $\mathrm{A}'(s, (c')^\sigma)$ is the same as the E-condition $\mathrm{A}'(x\colon C \hookrightarrow X, c')^\sigma$ where intuitively, $X$ is the graph obtained from $E$ by reversing the substitution. More specifically, $X$ is the graph with the property $X^{\alpha'} = E^\alpha$ where $\alpha'$ is defined for all variables $\mathtt{x}$ as follows:

$$\alpha'(\mathtt{x}) = \begin{cases} \alpha(\mathtt{x}) & \text{if } \sigma(\mathtt{x}) \text{ is undefined,} \\ \sigma(\mathtt{x})^\alpha & \text{if } \sigma(\mathtt{x}) \text{ is defined.} \end{cases}$$

Using Lemma 7.2 again, we have $x''\colon X^{\alpha'} \hookrightarrow H \models \mathrm{A}'(x\colon C \hookrightarrow X, c')$. Now, we can use the induction hypothesis to yield $x'' \circ x^{\alpha'} \models c'$. Since $X^\sigma = E$, $X^{\alpha'} = E^\alpha$, and since that $\alpha'$ is "embedding" the effect of $\sigma$, we can bring the substitution back to the syntactic level to yield $q'' \circ s^\alpha \models (c')^\sigma$.

We have $p'' \circ p^\alpha \models_\alpha \exists(a^\sigma \mid \gamma^\sigma)$ and $q' = q'' \circ s^\alpha \models (c')^\sigma$. The latter is satisfied by an assignment that has at least the mappings of $\alpha$ (since the domain of $q'$ is $(C^\sigma)^\alpha$, and since from the assumption, $((\gamma')^\sigma)^{\sigma_\alpha}$ must evaluate to $\mathtt{tt}$ under some assignment), so $q' \models ((c')^\sigma)^{\sigma_\alpha}$ by Lemma 7.1. Together, this gives us $p'' \circ p^\alpha \models_\alpha \exists(a^\sigma \mid \gamma^\sigma, (c')^\sigma)$. By Lemma 7.2 and the definition of $\models$, we get the result that $p'' \circ p^\alpha \models \exists(a \mid \gamma, c')$.

*If.* Assume that $p'' \circ p^\alpha \models c$. We assume without loss of generality that it does so by $\alpha$ (see Remark 7.1), i.e. $p'' \circ p^\alpha \models_\alpha c$. Define $p' = p'' \circ p^\alpha$, which is injective since injectivity is closed under composition. By the definition of $\models_\alpha$, there exists an injective graph morphism $C^\alpha \hookrightarrow H$ with $(C^\alpha \hookrightarrow H) \circ a^\alpha = p'$. Consider substitutions $\sigma \in \Sigma$ where $(\gamma^\sigma)^\alpha = \mathtt{tt}$, and injective graph morphisms $q'\colon (C^\sigma)^\alpha \hookrightarrow H$ with $q' \circ (a^\sigma)^\alpha = p'$ and $q' \models ((c')^\sigma)^{\sigma_\alpha}$ (we assume that $\alpha$ has mappings for additional variables introduced by $\sigma$, see Remark 7.1). At least one such morphism is guaranteed to exist (i.e. if $\sigma$ is the empty substitution). From the construction yield pushouts $q^\sigma \circ a^\sigma = (a')^\sigma \circ p$ with pushout objects $(C')^\sigma$. Clearly, applying

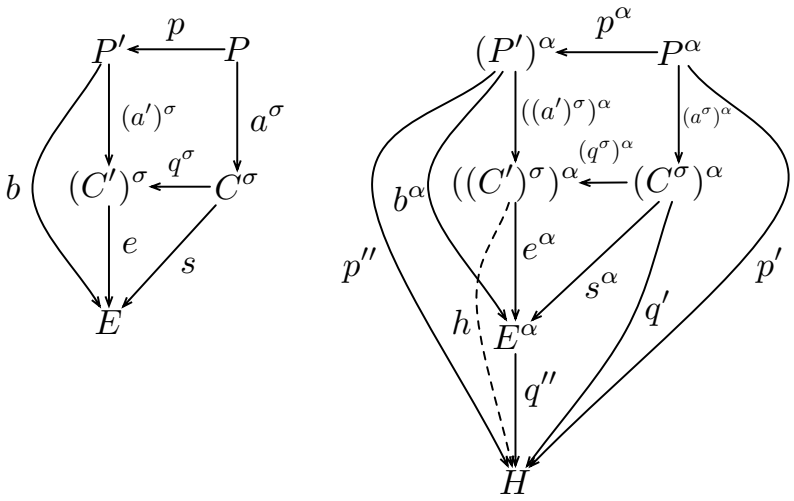



Figure 21. Instantiating the construction with an assignment

$\alpha$ to the morphisms yields pushouts of graphs from $\mathcal{G}(\mathcal{L})$. By the universal property of pushouts, each pushout has a unique morphism $h\colon ((C')^\sigma)^\alpha \to H$ with $p'' = h \circ ((a')^\sigma)^\alpha$ and $q' = h \circ (q^\sigma)^\alpha$. Consider $y \circ x = h$, a surjective-injective factorisation of $h$ with $x : ((C')^\sigma)^\alpha \to X$ surjective, $y : X \hookrightarrow H$ injective, $X \in \mathcal{G}(\mathcal{L})$, and injective morphisms $t = x \circ (q^\sigma)^\alpha$ and $u = x \circ ((a')^\sigma)^\alpha$. Now, we argue that for whichever $X \in \mathcal{G}(\mathcal{L})$ is yielded by the factorisation, the construction yields a graph $E \in \mathcal{G}(\mathrm{Exp})$ such that $E^\alpha \cong X$.

Suppose that $x$ is an injective morphism, and hence an isomorphism since it is also surjective, i.e. $((C')^\sigma)^\alpha \cong X$. The construction yields an isomorphism, i.e. $E \cong C^\sigma$. It follows that $E^\alpha \cong X$. Suppose now that $x$ is non-injective, i.e. some nodes (edges) in $(C^\sigma)^\alpha$ are merged. Since $t, u$ are injective, the images of $(P')^\alpha$ and $(C^\sigma)^\alpha$ in $X$ must overlap. Hence, a variable in $P'$ and another variable in $C$ must both be instantiated by $\alpha$ to the same label in $\mathcal{L}$. Yet these variables may be distinct and hence the labels they are in cannot be merged at the syntactic level. However, for non-empty $\sigma \in \Sigma$, such a variable in $C$, say $\mathtt{x}$, can be replaced with a corresponding variable in $P'$ by $\sigma$ such that $\alpha(\mathtt{x}) = \sigma(\mathtt{x})^\alpha$. Now, with $P'$ and $C$ sharing at least one label, the construction yields surjective morphisms $e : C^\sigma \to E$ where $E \ncong C^\sigma$ and $E^\alpha \cong X$. The construction gives us $s = e \circ q^\sigma$ and $b = e \circ (a')^\sigma$. It follows that $e^\alpha, q'', s^\alpha$, and $b^\alpha$ are equal to $x, y, t$, and $u$ up to isomorphism.

In all cases, $p'' = h \circ ((a')^\sigma)^\alpha$, $h = q'' \circ e^\alpha$, and $b^\alpha = e^\alpha \circ ((a')^\sigma)^\alpha$ yield $p'' = q'' \circ b^\alpha$, i.e. $p'' \models_\alpha \bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b \mid \gamma^\sigma)$. In each case we can apply a similar argument to that in the "Only if" section of the proof to obtain $q'' \models \mathrm{A}'(s, (c')^\sigma)$ from the induction hypothesis.

We have $p'' \models_\alpha \bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b \mid \gamma^\sigma)$ and $q'' \models \mathrm{A}'(s, (c')^\sigma)$. The latter is satisfied by an assignment that has at least the mappings of $\alpha$ (analogous to the reasons at the end of the "only if" section), so $q'' \models \mathrm{A}'(s, (c')^\sigma)^{\sigma_\alpha}$ by Lemma 7.1. Together, this gives us $p'' \models_\alpha \bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b \mid \gamma^\sigma, \mathrm{A}'(s, (c')^\sigma)) = \mathrm{A}'(p, c)$. Finally, we use the definition of $\models$ to yield the result that $p'' \models \mathrm{A}'(p, c)$.

When considering Boolean formulae over E-conditions, the statement follows from the definition and induction hypothesis. □

**Proof:**
Proof of Proposition 7.2.

From the construction of Proposition 7.2 and the statement of Lemma 7.5, we get $h \models \mathrm{A}(r, c)$ iff $h \models \mathrm{A}'(i_R, c)$ iff $h \circ i_R^\alpha \models c$ iff $i_H \colon \emptyset \hookrightarrow H \models c$ iff $H \models c$. □

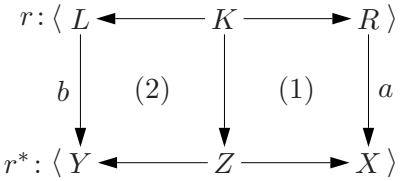
We now define and prove correct the transformation L, which transforms an E-app-condition over $R$ (the right-hand side of a rule schema) into an E-app-condition over $L$ (the left-hand side of a rule schema). Intuitively, one can think of the transformation as applying the rule schema in reverse to the graphs of the E-app-conditions.

**Proposition 7.3. (Transformation of E-app-conditions)**
There is a transformation L such that, for every rule schema $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ with rule schema condition $\Gamma$, every right E-app-condition $c$ for $r$, and every direct derivation $G \Rightarrow_{r,g,h} H$ with $g \colon L^\alpha \hookrightarrow G$ and $h \colon R^\alpha \hookrightarrow H$ where $G, H \in \mathcal{G}(\mathcal{L})$ and $\alpha$ is a well-typed assignment,

$$g \models_\alpha \mathrm{L}(r, c) \text{ if and only if } h \models_\alpha c.$$

□

*Construction.* All graphs used in the construction of the transformation belong to the class $\mathcal{G}(\mathrm{Exp})$. $\mathrm{L}(r, c)$ is inductively defined as follows. Let $\mathrm{L}(r, \mathrm{true}) = \mathrm{true}$ and $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \exists(b \mid \gamma, \mathrm{L}(r^*, c'))$ if $\langle K \hookrightarrow R, a \rangle$ has a natural pushout complement (1) with $r^* = \langle Y \hookleftarrow Z \hookrightarrow X \rangle$ denoting the "derived" rule by constructing natural pushout (2). If $\langle K \hookrightarrow R, a \rangle$ has no natural pushout complement, then $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \mathrm{false}$.




The transformation L is extended for Boolean formulae in the usual way, that is, $\mathrm{L}(r, \neg c) = \neg \mathrm{L}(r, c)$, and $\mathrm{L}(r, c_1 \wedge c_2) = \mathrm{L}(r, c_1) \wedge \mathrm{L}(r, c_2)$.

**Example 7.5.** Continuing from Example 7.4, we get:

$$\begin{aligned}
\mathrm{L}(r, \mathrm{A}(r, c)) \quad = \quad & \forall(\ \textcircled{x}_1 \hookrightarrow \textcircled{x}_1\ \textcircled{a}_2, \\
& \quad \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{type(a) = int}) \\
& \quad \vee \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b_c and type(b, c) = int})) \\
& \wedge \forall(\ \textcircled{x}_1 \hookrightarrow \textcircled{x}_1, \\
& \quad \exists(\ \textcircled{x}_1 \mid \texttt{type(x_0) = int}) \\
& \quad \vee \exists(\ \textcircled{x}_1 \mid \texttt{x_0 = b_c and type(b, c) = int})).
\end{aligned}$$

where the diagrams arising from applications of the construction are as given in Figure 22.

□

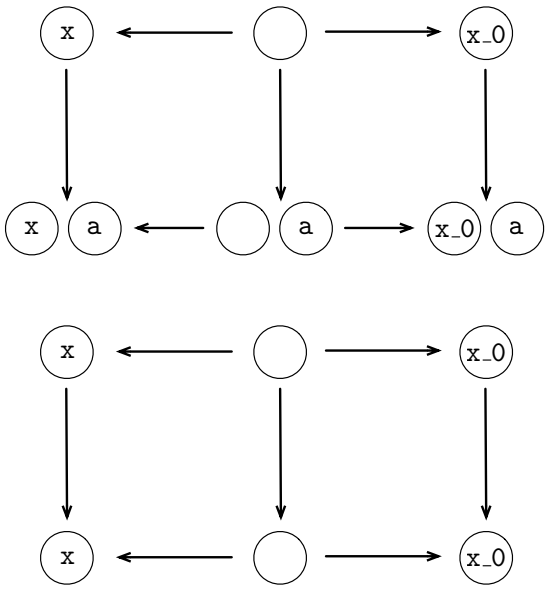



Figure 22. Instances of diagrams from the construction of L

Our proof of the proposition is similar to the proof of Theorem 6 in [10]. As earlier, the proof is simplified by the restriction to injective graph morphisms in E-conditions, but is made more complicated by the separation of graphs over the syntactic and semantic label alphabets.

**Proof:**
We prove the proposition by structural induction.

*Induction basis.* Let $c = \text{true}$. By construction, we get $\mathrm{L}(r, c) = \mathrm{L}(r, \text{true}) = \text{true}$. We have $g \models_\alpha \text{true}$ and $h \models_\alpha \text{true}$. All morphisms satisfy true.

*Induction hypothesis.* Assume that the proposition holds for E-condition $c'$.

*Induction step.* For a right E-app-condition of the form $c = \exists(a \mid \gamma, c')$, the construction distinguishes two cases. Let $l$ and $s$ denote the injective graph morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$, respectively. Let $(1)$ and $(2)$ denote the natural pushouts of the construction, and $(1)^\alpha$ and $(2)^\alpha$ denote the same diagrams but after the application of the well-typed assignment $\alpha$ to the morphisms (as in Figure 23). Clearly, for a given assignment $\alpha$, $(1)^\alpha$ and $(2)^\alpha$ are unique (up to isomorphism).

$$\begin{array}{ccccc} L & \xhookleftarrow{l} & K & \xhookrightarrow{s} & R \\ {\scriptstyle b}\downarrow & (2) & \downarrow & (1) & \downarrow{\scriptstyle a} \\ Y & \hookleftarrow & Z & \hookrightarrow & X \end{array} \qquad \begin{array}{ccccc} L^\alpha & \xhookleftarrow{l^\alpha} & K^\alpha & \xhookrightarrow{s^\alpha} & R^\alpha \\ {\scriptstyle b^\alpha}\downarrow & (2)^\alpha & \downarrow & (1)^\alpha & \downarrow{\scriptstyle a^\alpha} \\ Y^\alpha & \hookleftarrow & Z^\alpha & \hookrightarrow & X^\alpha \end{array}$$

Figure 23. Instantiating the construction with an assignment

*Case one.* The morphisms $\langle s, a\rangle$ have a natural pushout complement. By construction, we have $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \exists(b \mid \gamma, \mathrm{L}(r^*, c'))$ where $b\colon L \hookrightarrow Y$ and $r^* = \langle Y \hookleftarrow Z \hookrightarrow X\rangle$.

A. First, we show that given an injective graph morphism $q' : Y^\alpha \hookrightarrow G$ with $q' \circ b^\alpha = g$, there is a decomposition of the pushouts (see Figure 24) which yields the injective graph morphism $q\colon X^\alpha \hookrightarrow H$ with $q \circ a^\alpha = h$. Construct the pullback of $q'$ and $D \hookrightarrow G$, obtaining the pullback object $F \in \mathcal{G}(\mathcal{L})$. By the universal property of pullbacks, there is a unique graph morphism $K^\alpha \to F$ such that the arising diagrams commute. By the pushout-pullback decomposition, $(2')$ and $(4')$ are pushouts and pullbacks, i.e. natural pushouts. $K^\alpha \to F$ is injective as $b^\alpha$ is injective. Since the pushout complements of injective graph morphisms are unique up to isomorphism, and pushout $(2')$ is a natural pushout, we get that $(2')$ is equal to natural pushout $(2)^\alpha$ up to isomorphism and $F \cong Z^\alpha$.

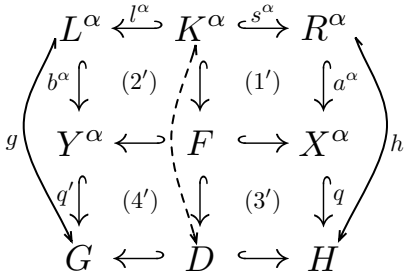



Figure 24. Decomposing a rule application

Now construct the natural pushout $(1')$ of $K^\alpha \hookrightarrow F$ and $s^\alpha$. By the uniqueness of pushout complements of injective morphisms, $(1')$ equals $(1)^\alpha$ (up to isomorphism). By the universal property of pushouts, there is a unique morphism $q\colon X^\alpha \to H$ with $q \circ a^\alpha = h$. By the decomposition lemma of pushouts, diagram $(3')$ is also a pushout. Since $q'$ and hence $F \hookrightarrow D$ are injective, it follows that $q$ is also injective.

B. Given an injective graph morphism $q\colon X^\alpha \hookrightarrow H$ with $q \circ a^\alpha = h$, one can yield $q' : Y^\alpha \hookrightarrow G$

with $q' \circ b^\alpha = g$ by instantiating $(1)$-$(2)$ into $(1)^\alpha$-$(2)^\alpha$, and decomposing these into $(1') - (4')$ as above, i.e. starting by constructing $(3')$ as a pullback of $q$ and $D \hookrightarrow H$.

C. For an assignment $\alpha'$ whose mappings comprise at least those of $\alpha$, the induction hypothesis states that $q' \colon Y^\alpha \hookrightarrow G \models_{\alpha'} \mathrm{L}(r^*, c')$ if and only if $q \colon X^\alpha \hookrightarrow H \models_{\alpha'} c$. By the definitions of L, $\models$, Lemma 7.1, and the statements above, we have:

$$
\begin{aligned}
& g \models_\alpha \mathrm{L}(r, \exists(a \mid \gamma, c')) = \exists(b \mid \gamma, \mathrm{L}(r^*, c')) \\
& \text{iff } \gamma^\alpha = \mathtt{tt} \text{ and there exists } q' \colon Y^\alpha \hookrightarrow G \text{ such that } q' \circ b^\alpha = g \text{ and } q' \models_{\alpha'} \mathrm{L}(r^*, c')^{\sigma_\alpha} \\
& \text{iff } \gamma^\alpha = \mathtt{tt} \text{ and there exists } q' \colon Y^\alpha \hookrightarrow G \text{ such that } q' \circ b^\alpha = g \text{ and } q' \models_{\alpha'} \mathrm{L}(r^*, c') \\
& \text{iff } \gamma^\alpha = \mathtt{tt} \text{ and there exists } q \colon X^\alpha \hookrightarrow H \text{ such that } q \circ a^\alpha = h \text{ and } q \models_{\alpha'} c' \\
& \text{iff } \gamma^\alpha = \mathtt{tt} \text{ and there exists } q \colon X^\alpha \hookrightarrow H \text{ such that } q \circ a^\alpha = h \text{ and } q \models_{\alpha'} (c')^{\sigma_\alpha} \\
& \text{iff } h \models_\alpha \exists(a \mid \gamma, c')
\end{aligned}
$$

*Case two.* The morphisms $\langle s, a \rangle$ do not have a natural pushout complement. By construction, we have $\mathrm{L}(r, \exists(a \mid \gamma, c')) = \text{false}$. The problem reduces to showing that $g \models_\alpha \text{false}$ iff $h \models_\alpha \exists(a \mid \gamma, c')$. By the definition of $\models_\alpha$, no morphism satisfies false, hence it is sufficient to argue that $h$ does not satisfy $\exists(a \mid \gamma, c')$.

Assume that $h \models_\alpha \exists(a \mid \gamma, c')$. Then there exists an injective graph morphism $q \colon X^\alpha \hookrightarrow H$ with $q \circ a^\alpha = h$. Then, as in case one, the pushout can be decomposed into pushouts $(1')$ and $(3')$. This means that the morphisms $\langle s, a \rangle$ have a pushout complement, which contradicts the assumption.

When considering Boolean formulae over E-app-conditions, the statement follows from the definition and induction hypothesis. □

We conclude this section by defining and proving correct the transformation Pre, which makes use of the transformations A and L. Pre transforms a postcondition into a precondition, intuitively by the following steps: (1) transform the postcondition into an E-app-condition over the right-hand side of the rule schema, (2) transform this into an E-app condition over the left-hand side of the rule schema, (3) nest this within an E-constraint quantified over all morphisms from $L$ which represent a match.

**Proposition 7.4. (Transformation of postconditions into preconditions)**
There is a transformation Pre such that, for every E-constraint $c$, every rule schema $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ with rule schema condition $\Gamma$, and every direct derivation $G \Rightarrow_r H$,

$$G \models \mathrm{Pre}(r, c) \text{ implies } H \models c.$$

□

*Construction.* Define $\mathrm{Pre}(r, c) = \forall(\emptyset \hookrightarrow L \mid \gamma_r, (\neg\mathrm{Dang}(r) \wedge \tau(L, \Gamma) \Rightarrow \mathrm{L}(r, \mathrm{A}(r, c))))$, where $\gamma_r$ is as defined in Proposition 7.1.

**Example 7.6.** Continuing from Examples 7.4 and 7.5, we get:

$$\begin{aligned}\text{Pre}(r,c) \quad = \quad & \forall(\ \textcircled{x}_1 \mid \texttt{type(x) = int}, \\ & \quad \forall(\ \textcircled{x}_1\ \textcircled{a}_2, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{type(a) = int}) \\ & \quad\quad \vee \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b\_c and type(b, c) = int})) \\ & \quad \wedge \forall(\ \textcircled{x}_1, \exists(\ \textcircled{x}_1 \mid \texttt{type(x\_0) = int}) \\ & \quad\quad \vee \exists(\ \textcircled{x}_1 \mid \texttt{x\_0 = b\_c and type(b, c) = int}))).\end{aligned}$$

Since $r$ does not delete any nodes, and does not have a rule schema condition, $\neg\text{Dang}(r) \wedge \tau(L,\Gamma) = \text{true}$, simplifying the nested E-constraint generated by Pre. We can simplify $\text{Pre}(r,c)$ by hand to yield:

$$\begin{aligned}\text{Pre}(r,c) \quad \equiv \quad & \forall(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{type(x) = int}, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{type(a) = int}) \\ & \quad\quad \vee \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \texttt{a = b\_c and type(b, c) = int})).\end{aligned}$$

□

**Proof:**
Define $i_G : \emptyset \hookrightarrow G$. Assume that $G \models \text{Pre}(r,c)$. Then there exists an assignment $\alpha$ such that $G \models_\alpha \text{Pre}(r,c)$. Then $i_G \models_\alpha \text{Pre}(r,c) = \forall(\emptyset \hookrightarrow L \mid \gamma_r, (\neg\text{Dang}(r) \wedge \tau(L,\Gamma) \Rightarrow \text{L}(r, \text{A}(r,c))))$. By the definition of $\models_\alpha$ for universally quantified E-conditions, for every $q\colon L^\alpha \hookrightarrow G$ with $q \circ (\emptyset \hookrightarrow L^\alpha) = i_G$, we have that $q \models \neg\text{Dang}(r)^{\sigma_\alpha} \wedge \tau(L,\Gamma)^{\sigma_\alpha} \Rightarrow \text{L}(r, \text{A}(r,c))^{\sigma_\alpha}$. By Lemma 7.2 and the definition of $\Rightarrow$, we have that $q \models \text{Dang}(r) \vee \neg\tau(L,\Gamma) \vee \text{L}(r, \text{A}(r,c))$.

Suppose that $q \not\models \text{L}(r, \text{A}(r,c))$. Then $q$ must satisfy $\text{Dang}(r) \vee \neg\tau(L,\Gamma)$ meaning that $G \not\Rightarrow_{\mathcal{R}}$ (this conclusion is clear from an examination of Proposition 7.1), i.e. a contradiction of the statement.

Suppose now that $q \models \text{L}(r, \text{A}(r,c))$. From Proposition 7.3 we get $h\colon R^\alpha \hookrightarrow H \models \text{A}(r,c)$. From Proposition 7.2 we get $H \models c$, the result. □

## 8. Soundness

In this section, we present our main result that the proof rules of our Hoare logic are sound for proving partial correctness of graph programs. That is, a graph program $P$ is partially correct with respect to a precondition $c$ and a postcondition $d$ (in the sense of Definition 6.1) if there exists a full proof tree whose root is the triple $\{c\}\ P\ \{d\}$.

**Theorem 8.1.** The proof system comprising the axioms and inference rules of Figures 12 is sound for graph programs, in the sense of partial correctness (Definition 6.1). □

**Proof:**
To prove soundness, we prove that each single proof rule is correct by appealing to the semantic function $[\![P]\!]G$ (see Section 4.3). The result then follows by structural induction on proof trees.

Let $c, d, e, inv$ be E-constraints, $P, Q$ be arbitrary graph programs, $\mathcal{R}$ be a set of conditional rule schemata, $r, r_i$ be conditional rule schemata, and $G, H, \overline{G}, G', H' \in \mathcal{G}(\mathcal{L})$. Recall that the symbol $\rightarrow$ denotes a small-step transition relation on configurations of graphs and programs.

[ruleapp]. Follows from Proposition 7.4.

[nonapp]. Suppose that $G \models \neg\text{App}(\mathcal{R})$. By Proposition 7.1, we get that there does not exist a graph $H$ such that $G \Rightarrow_{\mathcal{R}} H$, or equivalently, $G \not\Rightarrow_{\mathcal{R}}$. From the inference rule $[\text{Call}_2]_{\text{SOS}}$ we obtain the transition $\langle \mathcal{R}, G\rangle \to \text{fail}$ (intuitively, this indicates that the program terminates but without returning a graph). No graph will ever result; this is captured by the postcondition false, which no graph or morphism can satisfy.

[ruleset]. Suppose that we have a non-empty set of rule schemata $\{r_1, \ldots, r_n\}$ denoted by $\mathcal{R}$, that $G \models c$, and that we have a non-empty set of graphs $\bigcup_{r\in\mathcal{R}}\{H \in \mathcal{G}(\mathcal{L}) \mid G \Rightarrow_r H\}$ such that each $H \models d$ (if the set was empty, then [nonapp] would apply). For this set of graphs to be non-empty, at least one $r \in \mathcal{R}$ must be applicable to $G$. That is, there is a direct derivation $G \Rightarrow_{\mathcal{R}} H$ for some graph $H$ that satisfies $d$. From the inference rule $[\text{Call}_1]_{\text{SOS}}$ and the assumption, we get $[\![\mathcal{R}]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle \mathcal{R}, G\rangle \to H\}$ such that each $H \models d$.

[comp]. Suppose that $G \models c$, $[\![P]\!]G = \{G' \in \mathcal{G}(\mathcal{L}) \mid \langle P, G\rangle \to^+ G'\}$ such that each $G' \models e$, and $[\![Q]\!]G' = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle Q, G'\rangle \to^+ H\}$ such that each $H \models d$. Then $[\![P; Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle P; Q, G\rangle \to^+ \langle Q, G'\rangle \to^+ H\}$ such that each $H \models d$ follows from applications of the inference rules $[\text{Seq}_1]_{\text{SOS}}$ and $[\text{Seq}_2]_{\text{SOS}}$.

[cons]. Suppose that $G' \models c'$, $c \Rightarrow c'$, $d' \Rightarrow d$, and $[\![P]\!]G' = \{H' \in \mathcal{G}(\mathcal{L}) \mid \langle P, G'\rangle \to^+ H'\}$ such that each $H' \models d'$. If $G \models c$, we have $G \models c'$ since $c \Rightarrow c'$. By the assumption, we have for each $H \in [\![P]\!]G$ that $H \models d'$. From $d' \Rightarrow d$, we get $H \models d$.

[$\text{if}_1$]. *Case One.* Suppose that $G \models c$, $[\![P]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle P, G\rangle \to^+ H\}$ such that each $H \models d$, and $G \models \text{App}(\mathcal{R})$. By Proposition 7.1, executing $\mathcal{R}$ on $G$ will result in a graph. Hence by the assumption and the inference rule $[\text{If}_1]_{\text{SOS}}$, $[\![\texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle \to^+ H\}$ such that each $H \models d$.

*Case Two.* Suppose that $G \models c$, $[\![Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle Q, G\rangle \to^+ H\}$ such that each $H \models d$, and $G \models \neg\text{App}(\mathcal{R})$. By Proposition 7.1, executing $\mathcal{R}$ on $G$ will not result in a graph. Hence by the assumption and the inference rule $[\text{If}_2]_{\text{SOS}}$, $[\![\texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle \to^+ H\}$ such that each $H \models d$.

[!]. We prove the soundness of this proof rule by induction over the number of executions of $\mathcal{R}$ that do not result in finite failure. Assume that for any graph $G'$ such that $G' \models inv$, $[\![\mathcal{R}]\!]G' = \{H' \in \mathcal{G}(\mathcal{L}) \mid \langle \mathcal{R}, G'\rangle \to H'\}$ such that each $H' \models inv$.

*Induction basis.* Suppose that $G \models inv$. In the case that $\mathcal{R}$ cannot ever be applied to $G$ without finite failure, only the inference rule $[\text{Alap}_2]_{\text{SOS}}$ can be applied, that is, $[\![\mathcal{R}!]\!]G = \{G \in \mathcal{G}(\mathcal{L}) \mid \langle \mathcal{R}!, G\rangle \to G\}$. Since the graph is not changed, trivially, the invariant holds, i.e. $G \models inv$. Since the execution of $\mathcal{R}$ on $G$ does not result in a graph, $G \models \neg\text{App}(\mathcal{R})$.

*Induction hypothesis.* There is a configuration $\langle \mathcal{R}!, \overline{G}\rangle$ such that $\langle \mathcal{R}!, \overline{G}\rangle \to^* H$, with the property that if $\overline{G} \models inv$, then we have for each $H$ in $[\![\mathcal{R}!]\!]\overline{G} = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle \mathcal{R}!, \overline{G}\rangle \to^* H\}$ that $H \models inv$ and $H \models \neg\text{App}(\mathcal{R})$.

*Induction step.* Suppose that we have $[\![\mathcal{R}!]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle \mathcal{R}!, G\rangle \to \langle \mathcal{R}!, \overline{G}\rangle \to^* H\}$ where the first small-step transition arises from an application of $[\text{Alap}_1]_{\text{SOS}}$. Suppose that $G \models inv$. Then by assumption, $\overline{G} \models inv$. It follows from the induction hypothesis that each $H \models inv$ and $H \models \neg\text{App}(\mathcal{R})$. □

## 9. Conclusion

We have presented the first Hoare-style verification calculus for a practical graph transformation language. This required us to extend the nested graph conditions of Habel, Pennemann and Rensink with expressions as labels and with assignment constraints, in order to deal with GP's powerful rule schemata and infinite label alphabet. We have demonstrated the use of the calculus for proving partial correctness properties of a nondeterministic colouring program and a program checking for 2-colourability. Our main technical result is that our proof rules are sound with respect to GP's formal semantics.

It is an open problem whether the calculus is relatively complete, that is, whether for every program that is partially correct with respect to its pre- and postcondition, there exists a proof of this fact within the calculus when all valid implications $c \Rightarrow d$ of E-constraints are added as axioms. This would correspond to Cook's classical result that Hoare logic for imperative programs is relatively complete [7]. In both cases, the crucial problem is to show that for a given loop and its postcondition, the weakest precondition can be expressed in the assertion language. However, classical completeness proofs exploit that program states are mappings from program variables to values, while the states of graph programs are graphs. We remark that even in the simpler case of the nested graph conditions of Habel, Pennemann and Rensink, it is open whether the weakest preconditions of loops can be finitely expressed. This is why in [11], infinite weakest preconditions are generated for loops.

We want to extend our calculus so that the total correctness of graph programs can be proved. Then, besides ensuring that a program is partially correct, a proof would guarantee that all program runs terminate if started from graphs satisfying the program's precondition. To achieve this, the proof rule for loops could be extended by using a termination function $\#\colon \mathcal{G}(\mathcal{L}) \to \mathbb{N}$. The antecedent $\{inv\}\,\mathcal{R}\,\{inv\}$ would be strengthened to express that if $G \models inv$ and $G \Rightarrow_{\mathcal{R}} H$, then $H \models inv$ and $\#G > \#H$. The proof that $\mathcal{R}$ decreases the measure $\#$ would happen outside the Hoare calculus, similar to the proofs of the implications in the consequence rule.

Another topic for future work is to generalise the calculus such that it can handle conditions of branching statements and loop bodies that are arbitrary subprograms rather than sets of rule schemata. This may require a substantial strengthening of the assertion language, in order to incorporate the finite failure concept of GP's semantics.

Finally, we would like to increase the expressiveness of E-conditions by following Habel and Radke [14] in introducing graph variables that represent graphs generated by hyperedge-replacement systems. It is shown in [14] that this allows to specify graph properties such as connectedness and acyclicity, which are not first-order properties and hence beyond the power of (finite) nested conditions and E-conditions.

**Acknowledgements.** We are grateful to the anonymous referees for their detailed and thoughtful comments which helped to improve this paper.

## References

[1] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, third edition, 2009.

[2] Adam Bakewell, Detlef Plump, and Colin Runciman. Checking the shape safety of pointer manipulations. In *Int. Seminar on Relational Methods in Computer Science (RelMiCS 7), Revised Selected Papers*, volume 3051, pages 48–61. Springer-Verlag, 2004.

[3] Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062, pages 30–44. Springer-Verlag, 2004.

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

[5] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise semantics of EMF model transformations by graph transformation. In *Proc. Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301, pages 53–67. Springer-Verlag, 2008.

[6] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositional verification of architectural refactorings. In *Proc. Architecting Dependable Systems VI (WADS 2008)*, volume 5835, pages 308–333. Springer-Verlag, 2009.

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

[8] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178, pages 383–397. Springer-Verlag, 2006.

[9] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Dániel Varró. Using graph transformation for practical model-driven software engineering. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 91–117. Springer-Verlag, 2005.

[10] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[11] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proc. Graph Transformations (ICGT 2006)*, volume 4178, pages 445–460. Springer-Verlag, 2006.

[12] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030, pages 230–245. Springer-Verlag, 2001.

[13] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505, pages 135–147. Springer-Verlag, 2002.

[14] Annegret Habel and Hendrik Radke. Expressiveness of graph conditions with variables. In *Proc. Colloquium on Graph and Model Transformation on the Occasion of the 65th Birthday of Hartmut Ehrig*, volume 30 of *Electronic Communications of the EASST*, 2010.

[15] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal analysis of functional behaviour for model transformations based on triple graph grammars. In *Proc. Graph Transformations (ICGT 2010)*, volume 6372, pages 155–170. Springer-Verlag, 2010.

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

[17] Karsten Hölscher, Paul Ziemann, and Martin Gogolla. On translating UML models into graph transformation systems. *Journal of Visual Languages Computing*, 17(1):78–105, 2006.

[18] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare Logic with abrupt termination. In *Proc. Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783, pages 284–303. Springer-Verlag, 2000.

[19] Barbara König and Javier Esparza. Verification of graph transformation systems with context-free specifications. In *Proc. Graph Transformations (ICGT 2010)*, volume 6372, pages 107–122. Springer-Verlag, 2010.

[20] Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214, pages 305–320. Springer-Verlag, 2008.

[21] Sabine Kuske, Martin Gogolla, Hans-Jörg Kreowski, and Paul Ziemann. Towards an integrated graph-based semantics for UML. *Software and Systems Modeling*, 8:403–422, 2009.

[22] Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.

[23] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM Press, 2000.

[24] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag, 2007.

[25] Tobias Nipkow. Hoare logics in Isabelle/HOL. In Helmut Schwichtenberg and Ralf Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer Academic Publishers, 2002.

[26] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[27] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[28] Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725, pages 99–122. Springer-Verlag, 2009.

[29] Detlef Plump and Sandra Steinert. The semantics of graph programs. In *Proc. Rule-Based Programming (RULE 2009)*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 27–38, 2010.

[30] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In *Proc. Programming Languages and Systems (ESOP 1999)*, pages 162–176, 1999.

[31] Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372, pages 139–154. Springer-Verlag, 2010.

[32] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. Graph Transformations (ICGT 2004)*, volume 3256, pages 226–241. Springer-Verlag, 2004.

[33] Stefan Rieger and Thomas Noll. Abstracting complex data structures by hyperedge replacement. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214, pages 69–83. Springer-Verlag, 2008.

[34] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.

[35] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062, pages 446–453. Springer-Verlag, 2004.

[36] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[37] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.