A Hoare Calculus for Graph Programs

Christopher M. Poskitt and Detlef Plump

Department of Computer Science
The University of York, UK

Abstract. We present Hoare-style axiom schemata and inference rules for verifying the partial correctness of programs in the graph programming language GP. The pre- and postconditions of this calculus are the nested conditions of Habel, Pennemann and Rensink, extended with expressions for labels in order to deal with GP's conditional rule schemata and infinite label alphabet. We show that the proof rules are sound with respect to GP's operational semantics.

1 Introduction

Recent years have seen an increased interest in formally verifying properties of graph transformation systems, motivated by the many applications of graph transformation to specification and programming. Typically, this work has focused on verification techniques for sets of graph transformation rules or graph grammars, see for example [16,2,10,3,5].

Graph transformation languages and systems such as PROGRES [17], AGG [18], Fujaba [12] and GrGen [4], however, allow one to use control constructs on top of graph transformation rules for solving graph problems in practice. The challenge to verify programs in such 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 in [6], by providing a construction for weakest preconditions of so-called high-level programs. These programs allow one to use constructs such as sequential composition and as-long-as-possible iteration over sets of conditional graph transformation rules. The verification method follows Dijkstra's approach to program verification: one calculates the weakest precondition of a program and then needs to prove that it follows from the program's precondition. High-level programs fall short of practical graph transformation languages though, in that their rules cannot perform computations on labels (or attributes).

In this paper, we present an approach for verifying programs in the graph programming language GP [13,11]. Rather than adopting a weakest precondition approach, we follow Hoare's seminal paper [9] and devise a calculus of proof rules which are directed by the syntax of the language's control constructs. Similar to classical Hoare logic, the calculus aims at human-guided verification and allows the compositional construction of proofs.

The pre- and postconditions of our calculus are nested conditions [5], extended with expressions for labels and so-called assignment constraints; we refer to them as *E-conditions*. The extension is necessary for two reasons. Firstly, when a label alphabet is infinite, it is impossible to express a number of simple properties with finite nested conditions. For example, one cannot express with a finite nested condition that a graph over the set of integers is non-empty, since it is impossible to finitely enumerate every integer. Secondly, the conditions in [5] cannot express relations between labels such as "`x` and `y` are integers and $x^2 = y$". Such relations can be expressed, however, in GP's rule schemata.

We briefly review the preliminaries in Section 2 and graph programs in Section 3. Following this, we present E-conditions in Section 4, and then use them to define a proof system for GP in Section 5, where its use will be demonstrated by proving a property of a graph colouring program. In Section 6, we formally define the two transformations of E-conditions used in the proof system, before proving the axiom schemata and inference rules sound in the sense of partial correctness, with respect to GP's operational semantics [13,14]. Finally, we conclude in Section 7. A long version of this paper with the abstract syntax and operational semantics of GP, as well as detailed proofs of results, is available online [15].

2 Graphs, Assignments, and Substitutions

Graph transformation in GP is based on the double-pushout approach with relabelling [8]. This framework deals with partially labelled graphs, whose definition we recall below. We deal with 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.

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 changed 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 $l_V(v) = \bot$. 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 the levels of syntax and semantics.

Let $\mathbb{Z}$ be the set of integers and Char be a finite set of characters (that can be typed on a keyboard). We fix the label alphabet $\mathcal{L} = (\mathbb{Z} \cup \mathrm{Char}^*)^+$ of all non-empty sequences over integers and character strings, and denote by $\mathcal{G}(\mathcal{L})$ the set of all graphs over $\mathcal{L}$.

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

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

Fig. 1. Syntax of expressions

Each graph in $\mathcal{G}(\mathrm{Exp})$ 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 mapping $\alpha\colon \mathrm{VarId} \to \mathcal{L}$. Given an expression e, α is *well-typed* for e if for every term $t_1 \oplus t_2$ in e, with $\oplus \in \mathrm{ArithOp}$, we have $\alpha(\mathtt{x}) \in \mathbb{Z}$ for all variable identifiers $\mathtt{x}$ in $t_1 \oplus t_2$. 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^α 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 $\oplus \in \mathrm{ArithOp}$ and $t_1, t_2 \in \mathrm{Term}$, 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 $t \in \mathrm{Term} \cup \mathrm{String}$ and $e_1 \in \mathrm{Exp}$, then $e^\alpha = t^\alpha e_1^\alpha$ (the concatenation of t^α and e_1^α).

Given a graph G in $\mathcal{G}(\mathrm{Exp})$ and an assignment α that is well-typed for all expressions occurring in G, we write G^α for the graph in $\mathcal{G}(\mathcal{L})$ that is obtained from G by replacing each label e with e^α. If $g\colon G \to H$ is a graph morphism between graphs in $\mathcal{G}(\mathrm{Exp})$, then g^α denotes the morphism $\langle g_V^\alpha, g_E^\alpha \rangle\colon G^\alpha \to H^\alpha$.

A *substitution* is a mapping $\sigma\colon \mathrm{VarId} \to \mathrm{Exp}$. Given an expression e, σ is *well-typed* for e if for every term $t_1 \oplus t_2$ in e, with $\oplus \in \mathrm{ArithOp}$, we have $\sigma(\mathtt{x}) \in \mathrm{Term}$ for all variable identifiers $\mathtt{x}$ in $t_1 \oplus t_2$. In this case the expression e^σ is obtained from e by replacing every occurrence of a variable $\mathtt{x}$ with $\sigma(\mathtt{x})$. Given a graph G

[1] For simplicity, we use the non-terminals of our grammars to denote the syntactic classes of strings that can be derived from them.

in $\mathcal{G}(\text{Exp})$, we write G^σ for the graph that is obtained by replacing each label e with e^σ. If $g\colon G \to H$ is a graph morphism between graphs in $\mathcal{G}(\text{Exp})$, then g^σ denotes the morphism $\langle g_V^\sigma, g_E^\sigma \rangle\colon G^\sigma \to H^\sigma$.

Given an assignment α, the substitution σ_α *induced* by α maps every variable $\mathtt{x}$ 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 sequence $56, a, bc$, where 56 is an integer and a and bc are strings, then $\sigma_\alpha(\mathtt{x}) = 56\_\text{"a"}\_\text{"bc"}$.

3 Graph Programs

We briefly review GP's conditional rule schemata and discuss an example program. Technical details (including an operational semantics later used in our soundness proof) and further examples can be found in [13,14].

3.1 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 the application of the schemata. Rule schemata generalise graph transformation rules in the double-pushout approach with relabelling [8], in that labels can contain expressions over parameters of type integer or string. Figure 2 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.

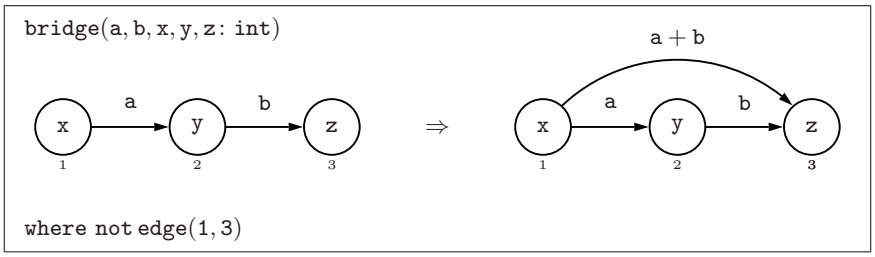


Fig. 2. A conditional rule schema

In the GP programming system [11], 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 (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1, 3)` in the condition of Figure 2 forbids an edge from node 1 to node 3 when the left graph is matched. The grammar of Figure 3 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
```

Fig. 3. Syntax of rule schema conditions

Conditional rule schemata represent possibly infinite sets of conditional graph transformation rules over graphs in $\mathcal{G}(\mathcal{L})$, and are applied according to the double-pushout approach with relabelling. A rule schema $L \Rightarrow R$ with condition Γ 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 are unlabelled) and $\Gamma^{\alpha,g}$ is a predicate on graph morphisms $g\colon L^\alpha \rightarrow G$ (see [13,14]).

3.2 Programs

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.

Example 1 (Colouring). 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 4 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.

The program initially colours each node with 1 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 highly nondeterministic: Figure 4 shows an execution producing a colouring with two colours, but a colouring with three colours could have been produced for the same input graph.

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 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 [13].

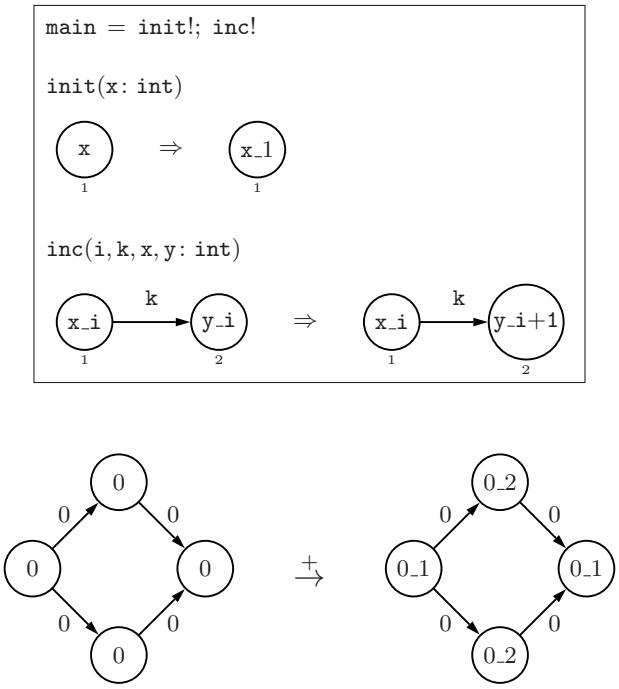


Fig. 4. The program `colouring` and one of its executions

4 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 [5] with expressions for labels, and assignment constraints which restrict the values that can be assigned to variables. The resulting conditions can be considered as representations of possibly infinite sets of ordinary nested conditions.

Definition 1 (Assignment constraint) An *assignment constraint* is a Boolean expression conforming to the grammar in Figure 5. We require that the arguments of the operators $>$, $<$, $\geq$ and $\leq$ belong to the syntactic class Term and that the arguments of $=$ and $\neq$ belong both to either Term, String or Exp $-$ (Term $\cup$ String). (See Figure 1 for the definition of Term, String and Exp.)

Given an assignment constraint γ and an assignment $\alpha\colon \mathrm{VarId} \to \mathcal{L}$, the *value* γ^α in $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ is inductively defined[2]. If $\gamma = \mathrm{true}$, then $\gamma^\alpha = \mathtt{tt}$. If γ has the form $e_1 \bowtie e_2$ with $\bowtie \in \mathrm{ACRelOp}$ and $e_1, e_2 \in \mathrm{Exp}$, then $\gamma^\alpha = \mathtt{tt}$

[2] We assume that α is well-typed for γ, which is defined in a similar way to before.

ACBoolExp ::= Exp ACRelOp Exp | '¬' ACBoolExp
| ACBoolExp ACBoolOp ACBoolExp
| 'type' '(' VarId ')' '=' Type | 'true'
ACRelOp ::= '=' | '≠' | '>' | '<' | '≥' | '≤'
ACBoolOp ::= '∧' | '∨'
Type ::= 'int' | 'string' | 'tagged'

Fig. 5. Syntax of assignment constraints

if and only if $e_1^\alpha \bowtie_{\mathcal{L}} e_2^\alpha$ where $\bowtie_{\mathcal{L}}$ is the obvious relation on $\mathcal{L}$ represented by $\bowtie$. If $\gamma = \neg\gamma_1$ with $\gamma_1 \in$ ACBoolExp, then $\gamma^\alpha = \mathtt{tt}$ if and only if $\gamma_1^\alpha = \mathtt{ff}$. If $\gamma = \gamma_1 \oplus \gamma_2$ with $\gamma_1, \gamma_2 \in$ ACBoolExp and $\oplus \in$ ACBoolOp, 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 γ has the form $\mathrm{type}(\mathtt{x}) = t$ with $\mathtt{x} \in$ VarId and $t \in$ Type, then $\gamma^\alpha = \mathtt{tt}$ if and only if $\mathit{type}(\alpha(\mathtt{x})) = t$, where the function $\mathit{type}\colon \mathcal{L} \to$ Type is defined by

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

Example 2 (Assignment constraint). Consider the assignment constraint $\gamma = \mathtt{a} > \mathtt{b} \wedge \mathtt{b} \neq 0 \wedge \mathrm{type}(\mathtt{a}) = \text{int}$. Let $\alpha_1 = (\mathtt{a} \mapsto 5, \mathtt{b} \mapsto 1)$ and $\alpha_2 = (\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 0)$. Then $\gamma^{\alpha_1} = \mathtt{tt}$ and $\gamma^{\alpha_2} = \mathtt{ff}$.

Note that variables in assignment constraints do not have a type per se, unlike the variables in GP rule schemata. Rather, the operator 'type' can be used to constrain the type of a variable. Note also that an assignment constraint such as $\mathtt{a} > 5 \wedge \mathrm{type}(\mathtt{a}) = \text{string}$ evaluates under every assignment to $\mathtt{ff}$, because we assume that assignments are well-typed.

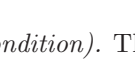
A substitution $\sigma\colon$ VarId $\to$ Exp is well-typed for an assignment constraint γ if the replacement of every occurrence of a variable $\mathtt{x}$ in γ with $\sigma(\mathtt{x})$ results in an assignment constraint. In this case the resulting constraint is denoted by γ^σ.

Definition 2 (E-condition) An *E-condition* c over a graph P is of the form true or $\exists(a|\gamma, c')$, where $a\colon P \hookrightarrow C$ is an injective[3] graph morphism with $P, C \in \mathcal{G}(\text{Exp})$, γ is an assignment constraint, and c' is an E-condition over C. Moreover, Boolean formulas 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, where c_1 and c_2 are E-conditions over P.

For brevity, we write false for $\neg$true, $\exists(a|\gamma)$ for $\exists(a|\gamma, \text{true})$, $\exists(a, c')$ for $\exists(a|\text{true}, c')$, and $\forall(a|\gamma, c')$ for $\neg\exists(a|\gamma, \neg c')$. In 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

[3] For simplicity, we restrict E-conditions to injective graph morphisms since this is sufficient for GP.

$\exists(C, \exists(C'))$, 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 3 (E-condition). The E-condition $\forall($  $\mid \mathtt{x} > \mathtt{y}, \exists($  $))$ (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:

$$\neg\exists(\emptyset \hookrightarrow$$ $$\mid \mathtt{x} > \mathtt{y}, \neg\exists($$ $$\hookrightarrow$$ $$\mid \text{true}, \text{true})).$$

The *satisfaction* of E-conditions by injective graph morphisms over $\mathcal{L}$ 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 condition $c = \exists(a\colon P \hookrightarrow C | \gamma, c')$, denoted $s \models c$, if there exists an assignment α such that $S = P^\alpha$ and there is an injective graph morphism $q\colon C^\alpha \hookrightarrow G$ with $q \circ a^\alpha = s$, $\gamma^\alpha = \mathtt{tt}$, and $q \models (c')^{\sigma_\alpha}$, where σ_α is the substitution induced by α.

A graph G in $\mathcal{G}(\mathcal{L})$ satisfies an E-condition c, denoted $G \models c$, if the morphism $\emptyset \hookrightarrow G$ satisfies c.

The application of a substitution σ to an E-condition c is defined inductively, too. We have $\text{true}^\sigma = \text{true}$ and $\exists(a|\gamma, c')^\sigma = \exists(a^\sigma | \gamma^\sigma, (c')^\sigma)$, where we assume that σ is well-typed for all components it is applied to.

5 A Hoare Calculus for Graph Programs

We present a system of partial correctness proof rules for GP, in the style of Hoare [1], using E-constraints as the assertions. We demonstrate the proof system by proving a property of our earlier `colouring` graph program, and sketch a proof of the rules' soundness according to GP's operational semantics [13,14].

Definition 3 (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 $[\![P]\!]G$.

Here, $[\![\_]\!]$ is GP's semantic function (see [14]), and $[\![P]\!]G$ is the set of all graphs resulting from executing program P on graph G. Note that partial correctness of a program P does not entail that P will actually 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 whenever a graph G satisfies c, then any graphs resulting from the application of P to G will satisfy d. Our proof system in Figure 6 operates on such triples. As in classical Hoare logic [1], we use the proof system to construct proof trees, combining axiom schemata and inference rules (an

example will follow). 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.

$$[\text{rule}]\ \frac{}{\{\text{Pre}(r,c)\}\ r\ \{c\}} \qquad [\text{ruleset}_1]\ \frac{}{\{\neg\text{App}(\mathcal{R})\}\ \mathcal{R}\ \{\text{false}\}}$$

$$[\text{ruleset}_2]\ \frac{\{c\}\ r_1\ \{d\}\ \ldots\ \{c\}\ r_n\ \{d\}}{\{c\}\ \{r_1,\ldots,r_n\}\ \{d\}} \qquad [!]\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}}$$

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

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

Fig. 6. Partial correctness proof system for GP

Two transformations — App and Pre — are required in some of the assertions. Intuitively, App takes as input a set $\mathcal{R}$ of conditional rule schemata, and transforms it into an E-condition specifying the property that a rule in $\mathcal{R}$ is applicable to the graph. 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 for the possible overlappings of c and the right-hand side of the rule schema r, (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 L and also accounts for the applicability of r.

Note that 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. This restriction does not affect the computational completeness of the language, because in [7] it is shown that a graph transformation language is complete if it contains single-step application and as-long-as-possible iteration of (unconditional) sets of rules, together with sequential composition.

Example 4 (Colouring). Figure 7 shows a proof tree for the `colouring` program of Figure 4. It proves that if `colouring` is executed on a graph in which

the node labels are exclusively integers, then any graph resulting will have the property that each node label is an integer with a colour attached to it, and that adjacent nodes have distinct colours. That is, it proves the triple $\{\neg\exists(\,\textcircled{\texttt{a}} \mid \text{type}(\texttt{a}) \neq \text{int})\}\ \texttt{init!};\ \texttt{inc!}\ \{\forall(\,\textcircled{\texttt{a}}_1, \exists(\,\textcircled{\texttt{a}}_1 \mid \texttt{a} = \texttt{b\_c} \wedge \text{type}(\texttt{b},\texttt{c}) = \text{int})) \wedge \neg\exists(\,\textcircled{\texttt{x\_i}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y\_i}} \mid \text{type}(\texttt{i},\texttt{k},\texttt{x},\texttt{y}) = \text{int})\}$. For conciseness, we abuse our notation (in this, and later examples), and allow $\text{type}(\texttt{x}_1,\ldots,\texttt{x}_n) = \text{int}$ to represent $\text{type}(\texttt{x}_1) = \text{int} \wedge \ldots \wedge \text{type}(\texttt{x}_n) = \text{int}$.

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

$$\begin{aligned}
c &= \neg\exists(\,\textcircled{\texttt{a}} \mid \text{type}(\texttt{a}) \neq \text{int}) \\
d &= \forall(\,\textcircled{\texttt{a}}_1, \exists(\,\textcircled{\texttt{a}}_1 \mid \texttt{a} = \texttt{b_c} \wedge \text{type}(\texttt{b},\texttt{c}) = \text{int})) \\
e &= \forall(\,\textcircled{\texttt{a}}_1, \exists(\,\textcircled{\texttt{a}}_1 \mid \text{type}(\texttt{a}) = \text{int}) \vee \exists(\,\textcircled{\texttt{a}}_1 \mid \texttt{a} = \texttt{b_c} \wedge \text{type}(\texttt{b},\texttt{c}) = \text{int})) \\
\neg\text{App}(\{\texttt{init}\}) &= \neg\exists(\,\textcircled{\texttt{x}} \mid \text{type}(\texttt{x}) = \text{int}) \\
\neg\text{App}(\{\texttt{inc}\}) &= \neg\exists(\,\textcircled{\texttt{x_i}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y_i}} \mid \text{type}(\texttt{i},\texttt{k},\texttt{x},\texttt{y}) = \text{int}) \\
\text{Pre}(\texttt{init}, e) &= \forall(\,\textcircled{\texttt{x}}_1\ \textcircled{\texttt{a}}_2 \mid \text{type}(\texttt{x}) = \text{int}, \exists(\,\textcircled{\texttt{x}}_1\ \textcircled{\texttt{a}}_2 \mid \text{type}(\texttt{a}) = \text{int}) \\
&\quad \vee \exists(\,\textcircled{\texttt{x}}_1\ \textcircled{\texttt{a}}_2 \mid \texttt{a} = \texttt{b_c} \wedge \text{type}(\texttt{b},\texttt{c}) = \text{int})) \\
\text{Pre}(\texttt{inc}, d) &= \forall(\,\textcircled{\texttt{x_i}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y_i}}_2\ \textcircled{\texttt{a}}_3 \mid \text{type}(\texttt{i},\texttt{k},\texttt{x},\texttt{y}) = \text{int}, \\
&\quad \exists(\,\textcircled{\texttt{x_i}}_1 \xrightarrow{\texttt{k}} \textcircled{\texttt{y_i}}_2\ \textcircled{\texttt{a}}_3 \mid \texttt{a} = \texttt{b_c} \wedge \text{type}(\texttt{b},\texttt{c}) = \text{int}))
\end{aligned}$$

Fig. 7. A proof tree for the program `colouring` of Figure 4

6 Transformations and Soundness

We provide full definitions of the transformations App and Pre in this section. In order to define Pre, it is necessary to first define the intermediary transformations A and L, which are adapted from basic transformations of nested conditions [5]. Following this, we will show that our proof system is sound according to the operational semantics of GP.

Proposition 1 (Applicability of a set of rule schemata) *For every set $\mathcal{R}$ of conditional rule schemata, there exists an E-constraint* $\mathrm{App}(\mathcal{R})$ *such that for every graph $G \in \mathcal{G}(\mathcal{L})$,*

$$G \models \mathrm{App}(\mathcal{R}) \iff G \in \mathrm{Dom}(\Rightarrow_{\mathcal{R}}),$$

where $G \in \mathrm{Dom}(\Rightarrow_{\mathcal{R}})$ if there is a direct derivation $G \Rightarrow_{\mathcal{R}} H$ for some graph H.

The transformation App gives 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, with nested E-conditions for handling restrictions on the application of the rule schemata (such as the dangling condition when deleting nodes).

Construction. Define $\mathrm{App}(\{\}) = \text{false}$ and $\mathrm{App}(\{r_1, \ldots, r_n\}) = \mathrm{app}(r_1) \vee \ldots \vee \mathrm{app}(r_n)$. For a rule schema $r_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ with rule schema condition Γ_i, define $\mathrm{app}(r_i) = \exists(\emptyset \hookrightarrow L_i | \gamma_{r_i}, \neg\mathrm{Dang}(r_i) \wedge \tau(L_i, \Gamma_i))$ where γ_{r_i} is a conjunction of expressions constraining 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` (Figure 4), then γ_{r_i} would be the Boolean expression $\mathrm{type}(\mathtt{i}) = \mathrm{int} \wedge \mathrm{type}(\mathtt{k}) = \mathrm{int} \wedge \mathrm{type}(\mathtt{x}) = \mathrm{int} \wedge \mathrm{type}(\mathtt{y}) = \mathrm{int}$.

Define $\mathrm{Dang}(r_i) = \bigvee_{a \in A} \exists a$, where the index set A ranges over all[4] injective graph morphisms $a \colon L_i \hookrightarrow L_i^{\oplus}$ such that the pair $\langle K_i \hookrightarrow L_i, a \rangle$ has no natural pushout[5] complement, and each $L_i^{\oplus}$ is a graph that can be obtained from L_i by adding either (1) a loop, (2) a single edge between distinct nodes, or (3) a single node and a non-looping edge incident to that node. All items in $L_i^{\oplus} - L_i$ are labelled with single variables, distinct from each other, and distinct from those in L_i. If the index set A is empty, then $\mathrm{Dang}(r_i) = \text{false}$.

We define $\tau(L_i, \Gamma_i)$ inductively (see Figure 3 for the syntax of rule schema conditions). If there is no rule schema condition, then $\tau(L_i, \Gamma_i) = \text{true}$. If Γ_i has the form $t_1 \bowtie t_2$ with t_1, t_2 in Term and $\bowtie$ in RelOp, then $\tau(L_i, \Gamma_i) = \exists(L_i \hookrightarrow L_i | t_1 \bowtie_{\mathrm{ACRelOp}} t_2)$ where $\bowtie_{\mathrm{ACRelOp}}$ is the symbol in ACRelOp that corresponds to the symbol $\bowtie$ from RelOp. If Γ_i has the form `not` b_i with b_i in BoolExp, then $\tau(L_i, \Gamma_i) = \neg\tau(L_i, b_i)$. If Γ_i has the form $b_1 \oplus b_2$ with b_1, b_2 in BoolExp and $\oplus$ in BoolOp, then $\tau(L_i, \Gamma_i) = \tau(L_i, b_1) \oplus_{\wedge,\vee} \tau(L_i, b_2)$ where $\oplus_{\wedge,\vee}$ is $\wedge$ for `and` and $\vee$ for `or`. Finally, if Γ_i is of the form `edge`(n_1,n_2) with n_1, n_2 in Node, then $\tau(L_i, \Gamma_i) = \exists(L_i \hookrightarrow L_i')$ where L_i' is a graph isomorphic to L_i, except for an additional edge whose source is the node with identifer n_1, whose target is the node with identifier n_2, and whose label is a variable distinct from all others in use.

Proposition 2 (From E-constraints to E-app-conditions) *There is a transformation* A *such that, for all E-constraints c, all rule schemata $r \colon L \Rightarrow R$*

[4] We equate morphisms with isomorphic codomains, so A is finite.
[5] A pushout is *natural* if is simultaneously a pullback [8].

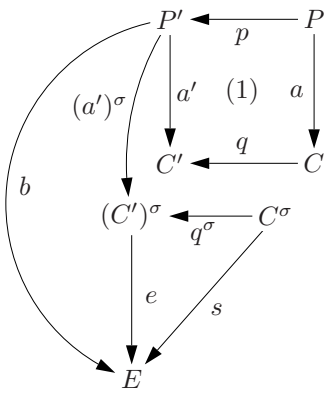
sharing no variables with c[6]*, and all injective graph morphisms* $h : R^\alpha \hookrightarrow H$ *where* $H \in \mathcal{G}(\mathcal{L})$ *and* α *is a well-typed assignment,*

$$h \models \mathrm{A}(r, c) \iff H \models c.$$

The idea of A is to consider a disjunction of all possible overlappings of R and the graphs of the E-constraint. Substitutions are used to replace label variables in c with portions of labels from R, facilitating the overlappings.

Construction. All graphs used in the construction of the transformation belong to the class $\mathcal{G}(\mathrm{Exp})$. For E-constraints $c = \exists(a : \emptyset \hookrightarrow C|\gamma, c')$ and rule schemata r, define $\mathrm{A}(r, c) = \mathrm{A}'(i_R : \emptyset \hookrightarrow R, c)$. For injective graph morphisms $p\colon P \hookrightarrow P'$, and E-conditions over P,

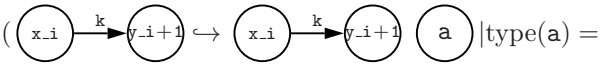
$$\begin{aligned} \mathrm{A}'(p, \mathrm{true}) &= \mathrm{true}, \\ \mathrm{A}'(p, \exists(a|\gamma, c')) &= \bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_\sigma} \exists(b|\gamma^\sigma, \mathrm{A}'(s, (c')^\sigma)). \end{aligned}$$



Construct the pushout (1) of p and a leading to injective graph morphisms $a'\colon P' \hookrightarrow C'$ and $q : C \hookrightarrow C'$. The finite double disjunction $\bigvee_{\sigma \in \Sigma} \bigvee_{e \in \varepsilon_\sigma}$ ranges first over substitutions from Σ, which have the special 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, and each β_i is a portion (or the entirety) of some label from P'. 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 ε_σ is the set of such surjective graph morphisms for a particular σ, the codomains of which we consider up to isomorphism. For a surjective graph morphism $e_1 : (C_1')^{\sigma_1} \to E_1$, E_1 is considered redundant and is excluded from the disjunction if there exists a surjective graph morphism, $e_2\colon (C_2')^{\sigma_2} \to E_2$, such that $E_2 \ncong E_1$, and there exists some $\sigma \in \Sigma$ such that $E_2^\sigma \cong E_1$.

The transformation A is extended for Boolean formulas over E-conditions in the same way as transformations over conditions (see [5]).

Example 5. Let r correspond to the rule schema `inc` (Figure 4), and E-constraint $c = \neg\exists(\ \textcircled{\mathtt{a}}\ \mid \mathrm{type}(\mathtt{a}) = \mathrm{int})$. Then,

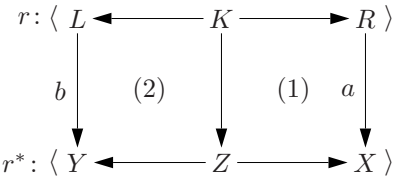
$$\mathrm{A}(r, c) = \neg\exists($$  $$\mid \mathrm{type}(\mathtt{a}) = \mathrm{int})$$

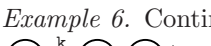
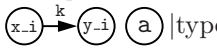
[6] It is always possible to replace the label variables in c with new ones that are distinct from those in r.

Proposition 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* Γ*, 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* α *is a well-typed assignment,*

$$g \models \mathrm{L}(r, c) \iff h \models c.$$

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 $\mathrm{L}(r, \exists(a|\gamma, c')) = \exists(b|\gamma, \mathrm{L}(r^*, c'))$ 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 $\mathrm{L}(r, \exists(a|\gamma, c')) = \mathrm{false}$.

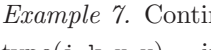
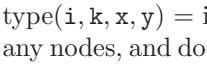


Example 6. Continuing from Example 5, we get $\mathrm{L}(r, \mathrm{A}(r, c)) = \neg\exists($  $\hookrightarrow$  $|\mathrm{type}(\mathtt{a}) = \mathrm{int})$.

Proposition 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* Γ*, and every direct derivation* $G \Rightarrow_r H$*,*

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

Construction. Define $\mathrm{Pre}(r, c) = \forall(\emptyset \hookrightarrow L|\gamma_r, (\neg\mathrm{Dang}(r) \wedge \tau(L, \Gamma) \Longrightarrow \mathrm{L}(r, \mathrm{A}(r, c))))$, where γ_r is as defined in Proposition 1.

Example 7. Continuing from Examples 5 and 6, we get $\mathrm{Pre}(r, c) = \forall($  $|$ $\mathrm{type}(\mathtt{i}, \mathtt{k}, \mathtt{x}, \mathtt{y}) = \mathrm{int}, \neg\exists($  $|\mathrm{type}(\mathtt{a}) = \mathrm{int}))$. Since r does not delete any nodes, and does not have a rule schema condition, $\neg\mathrm{Dang}(r) \wedge \tau(L, \Gamma) = \mathrm{true}$, simplifying the nested E-condition generated by Pre.

Our main result is that the proof rules of Figure 6 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 3) if there exists a full proof tree whose root is the triple $\{c\}\ P\ \{d\}$.

Theorem 1. *The proof system of Figure 6 is sound for graph programs, in the sense of partial correctness.*

Proof. To prove soundness, we consider each proof rule in turn, appealing to the semantic function $[\![P]\!]G$ (defined in [13,14]). The result then follows by induction on the length of proofs.

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 $G, H, \overline{G}, G'$, $H' \in \mathcal{G}(\mathcal{L})$. $\rightarrow$ is a small-step transition relation on configurations of graphs and programs. We decorate the names of the semantic inference rules of [14] with "SOS", in order to fully distinguish them from the names in our Hoare calculus.

[rule]. Follows from Proposition 4.

[ruleset$_1$]. Suppose that $G \models \neg\mathrm{App}(\mathcal{R})$. Proposition 1 implies that $G \notin \mathrm{Dom}(\Rightarrow_{\mathcal{R}})$, hence from the inference rule $[\mathrm{Call}_2]_{\mathrm{SOS}}$ we obtain the transition $\langle \mathcal{R}, G\rangle \rightarrow \mathrm{fail}$ (intuitively, this indicates that the program terminates but without returning a graph). No graph will ever result; this is captured by the post-condition false, which no graph can satisfy.

[ruleset$_2$]. 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 $\bigcup_{r\in\mathcal{R}}\{H \in \mathcal{G}(\mathcal{L}) | G \Rightarrow_r H\}$ such that each $H \models d$ (if the set was empty, then [ruleset$_1$] would apply). For the set 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 $[\mathrm{Call}_1]_{\mathrm{SOS}}$ and the assumption, we get $[\![\mathcal{R}]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}, G\rangle \rightarrow H\}$ such that each $H \models d$.

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

[cons]. Suppose that $G' \models c'$, $c \Longrightarrow c'$, $d' \Longrightarrow d$, and $[\![P]\!]G' = \{H' \in \mathcal{G}(\mathcal{L}) | \langle P, G'\rangle \rightarrow^+ H'\}$ such that each $H' \models d'$. If $G \models c$, we have $G \models c'$ since $c \Longrightarrow c'$. The assumption then gives us an $H \in [\![P]\!]G$ such that $H \models d'$. From $d' \Longrightarrow d$, we get $H \models d$.

[if]. *Case One.* Suppose that $G \models c$, $[\![P]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle P, G\rangle \rightarrow^+ H\}$ such that each $H \models d$, and $G \models \mathrm{App}(\mathcal{R})$. Then by Proposition 1, executing $\mathcal{R}$ on G will result in a graph. Hence by the assumption and the inference rule $[\mathrm{If}_1]_{\mathrm{SOS}}$, $[\![\texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightarrow \langle P, G\rangle \rightarrow^+ H\}$ such that each $H \models d$. *Case Two.* Suppose that $G \models c$, $[\![Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle Q, G\rangle \rightarrow^+ H\}$ such that each $H \models d$, and $G \models \neg\mathrm{App}(\mathcal{R})$. Then by Proposition 1, executing $\mathcal{R}$ on G will not result in a graph. Hence by the assumption and the inference rule $[\mathrm{If}_2]_{\mathrm{SOS}}$, $[\![\texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \rightarrow \langle Q, G\rangle \rightarrow^+ H\}$ 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, which we denote by n. Assume that for any graph G' such that $G' \models inv$, $[\![\mathcal{R}]\!]G' = \{H' \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}, G'\rangle \rightarrow^+ H'\}$ such that each $H' \models inv$. *Induction Basis ($n = 0$).* Suppose that $G \models inv$. Only the inference rule $[\mathrm{Alap}_2]_{\mathrm{SOS}}$ can be applied, that is, $[\![\mathcal{R}!]\!]G = \{G \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}!, G\rangle \rightarrow G\}$. 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 ($n = k$).* Assume that there exists a configuration $\langle \mathcal{R}!, G\rangle$ such that $\langle \mathcal{R}!, G\rangle \to^* \langle \mathcal{R}!, H\rangle \to H$. Hence for $[\![\mathcal{R}!]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}!, G\rangle \to^* \langle \mathcal{R}!, H\rangle \to H\}$, we assume that if $G \models inv$, then each $H \models inv$ and $H \models \neg\text{App}(\mathcal{R})$. *Induction Step ($n = k + 1$).* We have $[\![\mathcal{R}!]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}!, G\rangle \to \langle \mathcal{R}!, \overline{G}\rangle \to^* \langle \mathcal{R}!, H\rangle \to H\}$. Let $G \models inv$, and $G \cong G'$. From the assumption, we get $H' \models inv$, $H' \cong \overline{G}$, and hence $\overline{G} \models inv$. It follows from the induction hypothesis that each $H \models inv$ and $H \models \neg\text{App}(\mathcal{R})$.

7 Conclusion

We have presented the first Hoare-style verification calculus for an implemented graph transformation language. This required us to extend the nested graph conditions of Habel, Pennemann and Rensink with expressions for labels and 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 the partial correctness of a highly nondeterministic colouring program, and have shown that our proof rules are sound with respect to GP's formal semantics.

Future work will investigate the completeness of the calculus. Also, we intend to add termination proof rules in order to verify the total correctness of graph programs. Finally, we will consider how the calculus can be generalised to deal with GP programs in which the conditions of branching statements and the bodies of loops can be arbitrary subprograms rather than sets of rule schemata.

Acknowledgements. We are grateful to the anonymous referees for their comments which helped to improve the presentation of 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. 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.
3. Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositional verification of architectural refactorings. In *Proc. Architecting Dependable Systems VI (WADS 2008)*, volume 5835 of *Lecture Notes in Computer Science*, pages 308–333. Springer-Verlag, 2009.
4. 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 of *Lecture Notes in Computer Science*, pages 383–397. Springer-Verlag, 2006.
5. 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.

6. Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, Lecture Notes in Computer Science, pages 445–460. Springer-Verlag, 2006.
7. 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 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.
8. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.
9. Tony Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
10. Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 305–320. Springer-Verlag, 2008.
11. 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.
12. 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.
13. Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.
14. 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.
15. Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs (long version), 2010. Available online: http://www.cs.york.ac.uk/plasma/publications/pdf/PoskittPlump.ICGT.10.Long.pdf.
16. Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. Graph Transformations (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2004.
17. 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.
18. 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 of *Lecture Notes in Computer Science*, pages 446–453. Springer-Verlag, 2004.