# Hoare Logic for Graph Programs

Christopher M. Poskitt and Detlef Plump

Department of Computer Science
The University of York, UK

**Abstract.** We present a new approach for verifying programs written in GP (for Graph Programs), an experimental programming language for performing computations on graphs at a high level of abstraction. Taking a labelled graph as input, a graph program nondeterministically applies to it a number of graph transformation rules, directed by simple control constructs such as sequential composition and as-long-as-possible iteration. We adapt classical Hoare logic to the domain of graphs, and describe a system of sound proof rules for showing the partial correctness of graph programs.

## 1 Introduction

Rule-based graph transformation (or graph rewriting) has been studied since the 1970s, motivated by its many applications to programming and specification, and the natural visualisation that graphs and graph transformation rules give to dynamic systems (see the recent monograph [4]). Recently, graph-based programming languages have seen increased interest as a way of controlling the application of rules to graphs, in order to solve graph problems in practice. For example, in implementing a graph algorithm, a graph program might direct the application of rules to a graph such that they compute its transitive closure. In a setting where a graph represents a system state, graph programs might represent the system's operational behaviour.

Often, it is desirable to be able to prove that a graph program is correct according to some specification. Suppose that a graph program computes a colouring of a graph, encoding the colours in the labels of nodes. Can we prove that the graph program will always produce properly coloured graphs? Suppose that we model the states of an access control system with graphs, and describe the operation of logging out a user by a graph program. Can we prove that certain safety properties are conformed to by the design of the operation?

Up until now, research has tended to focus on proving the correctness of graph grammars, and sets of graph transformation rules applied arbitrarily to graphs (see, for example, [19,2,11,3,6]). A first step towards verifying graph programs was taken by Habel, Pennemann, and Rensink [7], who adopted Dijkstra's weakest preconditions approach for so-called high-level programs, which provide control constructs such as sequential composition and as-long-as-possible iteration over sets of conditional graph transformation rules. However, to the best of our knowledge, the challenge of verifying programs written in implemented graph transformation languages — such as PROGRES [20], AGG [21], Fujaba [13] and GrGen [5] — has yet to be addressed.

In this paper, we present an approach for verifying programs in the graph programming language GP (for Graph Programs) [16,12], a nondeterministic and computationally complete language for solving problems in the domain of graphs, and for which a prototype implementation exists. Rather than adopting a weakest precondition approach, we follow Hoare's seminal paper [10] and devise a calculus of proof rules which are directed by the syntax of GP's control constructs. Similar to classical Hoare logic, our calculus aims to facilitate human-guided verification and the compositional construction of proofs.

We intend in this paper to give the reader an informal understanding of our approach, favouring intuition and examples over the full technical details. These however can be found in [18] (a preprint of which is available from the authors' websites).

The organisation of this paper is as follows. Section 2 provides a brief introduction to GP, and explores its features through an example program. Section 3 introduces E-conditions, a graph specification formalism we use in the assertions of our Hoare triples. Section 4 presents the axiom schemata and inference rules of our partial correctness proof system, and demonstrates their use in an example proof of a simple graph program. Finally, in Section 5, we conclude.

## 2 Graph Programs

Graph programs are constructed from two components. First, a set of conditional rule schemata; intuitively, these are graph transformation rules with variables and expressions allowed as labels. Second, a sequence of commands controlling the application of the conditional rule schemata to a provided input graph. We review conditional rule schemata and programs in turn, and discuss an example program. Technical details and further examples can be found in [16,17].

### 2.1 Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of graph programs, each one describing a single-step transformation of a graph. Rule schemata comprise two graphs: a left graph which describes a part to be matched, and a right graph which describes what the match should be replaced with. The labels of their graphs contain expressions whose variables are instantiated in the graph matching process to integers or strings. The possible instantiations of these variables can be restricted by a rule schema condition, a simple predicate demanding particular relationships between variables, or the non-existence of edges. The expressions in labels are evaluated to integers or strings after the variables have been instantiated. Rule schemata are entirely syntactic constructs, representing possibly infinite sets of graph transformation rules (since the graphs GP operates on are labelled over an infinite label alphabet of integers and strings).

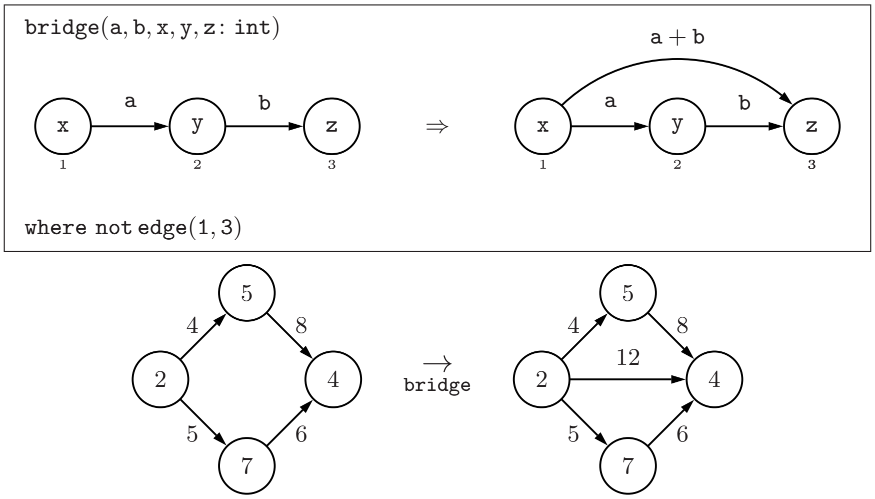


**Fig. 1.** A conditional rule schema and one of its applications

Figure 1 shows an example of a conditional rule schema, and a possible result of its application to a graph. The rule schema consists of the identifier `bridge` followed by the declaration of integer variables, the left and right graphs of the rule schema, the node identifiers `1`, `2`, `3` specifying which nodes are preserved, and the keyword `where` followed by a rule schema condition. Variables are instantiated with values (integers or character strings) in the graph matching process. Informally, in the application of a rule schema to a graph, a match is found for an instantiation of the left-hand side, and is replaced with the corresponding instantiation of the right-hand side. In our example, we have the following instantiation of variables: $\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 4, \mathtt{a} \mapsto 4, \mathtt{b} \mapsto 8$. Observe that this is not the only possible instantiation; matches are chosen nondeterministically.

GP allows nodes and edges in rule schemata to be labelled with underscore delimited sequences, for example, `5_0` and ”`York`”_`1`. Sequences can contain items of type integer and string. They are typically used to encode information into a graph, for example, to mark a node as reachable, or “tag” a node with an integer that represents its colour.

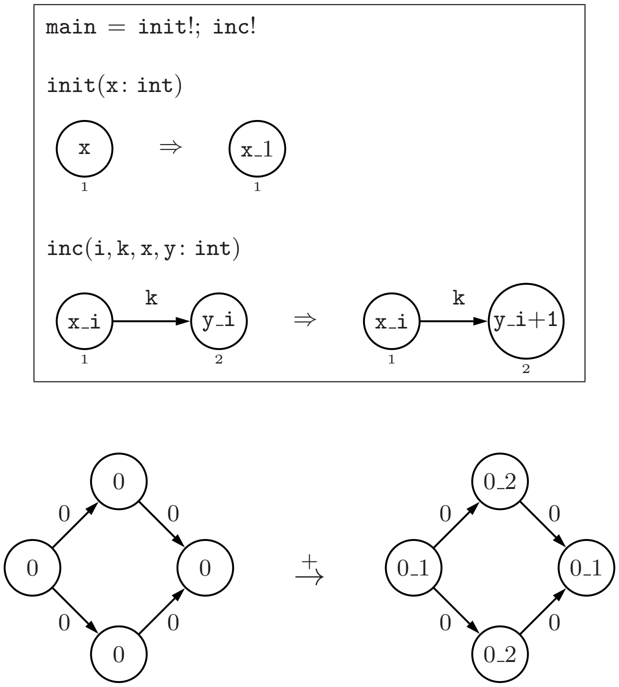
In the prototype GP programming system [12], rule schemata are constructed with a graphical editor. Labels in the left graph may only contain variables or constants (no composite expressions) because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`; 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 1 forbids an edge from node 1 to node

3 when the left graph is matched. The full syntax of conditions is given in [18]. Technically speaking, a rule schema is applied to a graph according to a generalisation of the double-pushout approach with relabelling [9].

### 2.2 Programs

We discuss an example program to familiarise the reader with GP's features. We will return to this program later in the paper to prove its correctness.

*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 2, with the command sequence `init!; inc!`, produces a colouring for every integer-labelled input graph, recording colours as tags.



```
main = init!; inc!

init(x: int)

inc(i, k, x, y: int)
```

**Fig. 2.** The program `colouring` and one of its executions

The program initially colours each node with 1 by applying the rule schema `init` for 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 2 shows an execution producing a colouring with two colours, but a colouring with three colours could have been produced for the same input graph.

Control constructs not used in `colouring` are $\{r_1, \ldots, r_n\}$, which denotes the nondeterministic application of a rule $r_i$ from the set, and `if` $C$ `then` $P$ `else` $Q$ which executes program $P$ if $C$ terminates with output[1], and $Q$ otherwise.

A full structural operational semantics is defined for GP in [17]. Each graph program is assigned a semantic function, which takes a graph as input, and returns as output the set of all graphs that could result from the execution of the program to that input graph.

## 3 Nested Graph Conditions with Expressions

Since the states of graph programs are graphs, and the pre- and postconditions of Hoare triples describe properties of program states, we require a specification formalism for precisely describing and reasoning about properties of graphs. The nested graph conditions of Habel and Pennemann [6] are such a specification formalism, expressively equivalent to first-order logic on graphs. Graph conditions however are unable to finitely express many properties when graphs are labelled over infinite label alphabets. For example, if we consider graphs labelled over the set of integers, it is impossible to finitely express a property as simple as "there exists an integer-labelled node"; we would require the following infinite graph condition:

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

Since GP's label alphabet is infinite (it consists of sequences of arbitrary integers and character strings), we extend the formalism to allow expressions with variables in labels, and to have a Boolean expression restricting the instantiations of variables; we refer to what results as *E-conditions*. E-conditions are able to finitely represent infinite graph conditions. The infinite graph condition above, for example, expresses the same property as the finite E-condition $\exists(\;\textcircled{\texttt{x}}\mid \mathrm{type}(\texttt{x}) = \mathrm{int})$, where `x` is a variable that can be instantiated to any integer.

A simple example of an E-condition is $c = \exists(\;\textcircled{\texttt{x}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y}}\;)$, which is read "there exists at least one non-looping edge". A graph $G$ would satisfy this E-condition, denoted $G \models c$, if variables `k`, `x`, and `y` could be instantiated to labels, which together with the nodes and edge, form a subgraph of $G$.

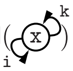
An *assignment constraint* (Boolean expression) allows one to restrict the types and values of variable instantiations. For example,

$$\exists(\;\textcircled{\texttt{x}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y}} \mid \mathrm{type}(\texttt{x}, \texttt{y}) = \mathrm{int} \wedge \texttt{x} < \texttt{y})$$

is read "there exists at least one pair of adjacent integer-labelled nodes, of which the label of the target node is larger than that of the source node".

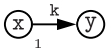
[1] Program $C$ is tested on a *copy* of the input graph, which is subsequently discarded.

Boolean expressions over E-conditions are also E-conditions. For example,

$d = \neg\exists($  $)$

is read "there does not exist a node incident to more than one loop". Suppose that $G \models d$. Then it is the case that no instantiation of `i`, `k`, and `x` will give labels, together with a node and two loops, that form a subgraph of $G$.

E-conditions may also be nested. For example,

$e = \forall($ $\mid \mathrm{type}(\mathtt{x}) = \mathrm{int}, \neg\exists($  $))$

is read "no integer-labelled node has an outgoing edge to another node (with any label)". When an E-condition contains nesting, for a graph to satisfy it, we need to look further than for some simple subgraph. Suppose that $G \models e$. Then for every instantiation of `x` to an integer that, together with the single node, gives a subgraph of $G$, there must not be an outgoing edge from that node to any node in $G$ with a label that `y` can be instantiated to (i.e. any node).

The formal definition of E-conditions is based on injective graph morphisms (i.e. structure preserving mappings between graphs), and allows an arbitrary amount of nesting; technical details can be found in [18].

## 4 A Hoare Calculus for Graph Programs

We present in this section a system of partial correctness axiom schemata and inference rules for GP, in the style of Hoare [1], using E-conditions as the assertions. We demonstrate the proof system by proving a property of our earlier `colouring` graph program.

First, we discuss what *partial correctness* means in the sense of graph programs. In the classical sense, a Hoare triple $\{s\}\ P\ \{t\}$ with $s, t$ formulas of predicate logic and $P$ a program fragment, is read "if $P$ is executed when the program state satisfies $s$, then should $P$ terminate, the program state will satisfy $t$". The execution of a graph program can follow one of three paths: it terminates with an output graph (referred to as *successful termination*), it terminates without an output graph (this is referred to as *failure*, and occurs when a rule schema, or a set of rule schemata, cannot be applied to the current graph), or it does not terminate at all (for example, `r!` will never terminate if the left graph of the rule schema is the empty graph $\emptyset$). Because of GP's nondeterminism, all three outcomes may be possible for the same program and input graph. We consider for partial correctness the successful termination case, in that if a program does terminate with an output graph, whatever that output graph may be, it satisfies some property expressed by an E-condition.

Given E-conditions $c, d$ and a graph program $P$, a triple of the form $\{c\}\ P\ \{d\}$ expresses the claim that whenever a graph $G$ satisfies $c$, then any graph that results from the application of $P$ to $G$ will satisfy $d$. The axiom schemata and inference rules that follow operate on such triples. As in classical Hoare logic [1], we use the proof system to construct proof trees, combining axioms and inference rules (an example will follow). We let $c, d, e, inv$ range over E-conditions, $P, Q$ over arbitrary programs, $r, r_i$ over conditional rule schemata, and $\mathcal{R}$ over sets of conditional rule schemata.

$$[\text{rule}]\ \frac{}{\{\text{Pre}(r, c)\}\ r\ \{c\}}$$

The axiom [rule] for the application of a single conditional rule schema works "backwards". Starting with a rule schema $r$ and E-condition $c$ as a postcondition, the transformation Pre is used to construct a 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 based on graph morphisms and pushout constructions (see [18]), but informally can be described by the following steps: (1) form a disjunction of E-conditions over all possible overlappings of E-condition $c$ and the right graph of rule schema $r$, (2) shift the disjunction of E-conditions from the right- to the left-hand side of $r$, (3) nest this within another E-condition that is universally quantified over the left graph of $r$.

We have that $\text{Pre}(r, c)$ implies $\text{App}(\{r\})$, where App constructs an E-condition expressing the weakest property that must be satisfied for a given rule schema set to be applicable to a graph (see below). The transformation Pre considers applicability, since otherwise, we would have to deal with failing computations.

Whereas assignment is basic to imperative programs and assignment axioms core to their correctness proofs, rule application is basic to graph programs and the [rule] axiom core to their correctness proofs.

$$[\text{ruleset}_1]\ \frac{}{\{\neg\text{App}(\mathcal{R})\}\ \mathcal{R}\ \{\text{false}\}}$$

The inference rule [ruleset$_1$] is applied in the case that no rule schema $r \in \mathcal{R}$ can be applied to the graph. App takes as input a set $\mathcal{R}$ of conditional rule schemata, and transforms it into an E-condition describing the weakest property that a graph $G$ must satisfy for $\mathcal{R}$ to be applicable to it. If $\mathcal{R}$ is applicable to $G$, then at least one rule schema $r \in \mathcal{R}$ satisfies the following: (1) it has an instantiation of variables such that its left-hand graph is isomorphic to a subgraph of $G$, (2) it can be applied to $G$ without leaving dangling edges (i.e. edges which are not incident to nodes at both ends), and (3) the rule schema condition evaluates to true. The postcondition false cannot be satisfied by any graph.

$$[\text{ruleset}_2]\ \frac{\{c\}\ r_1\ \{d\}\ \ldots\ \{c\}\ r_n\ \{d\}}{\{c\}\ \{r_1, \ldots, r_n\}\ \{d\}}$$

The inference rule [ruleset$_2$] is applied when the non-applicability of a rule schema set is not implied by the precondition. Since the rule schema to be applied is nondeterministically chosen from the set, it must be shown that the successful termination of *any* rule schema in the set results in a graph satisfying the desired postcondition, $d$. Note that the transformation App does not appear, since its effects are encapsulated by the transformation Pre in the axiom [rule].

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

The sequential composition rule [comp] follows its counterpart for imperative programming languages, in that we have to find an appropriate intermediate assertion, the E-condition $e$.

$$[\text{cons}]\ c \Longrightarrow c'\ \frac{\{c'\}\ P\ \{d'\}}{\{c\}\ P\ \{d\}}\ d' \Longrightarrow d$$

Similar to its classical counterpart, the rule of consequence [cons] allows us to strengthen the precondition and weaken the postcondition (or replace them with equivalent assertions), provided that the side conditions $c \Longrightarrow c'$ and $d' \Longrightarrow d$ are valid (mechanically proving such implications of E-conditions to be valid is a problem we have not yet addressed, however, Pennemann in [14,15] has developed a resolution-like theorem prover for implications of graph conditions).

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

The conditional rule [if] formalises a case distinction based on the applicability of $\mathcal{R}$ to the input graph, utilising the transformation App.

$$[!]\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\mathrm{App}(\mathcal{R})\}}$$

The as-long-as-possible iteration rule [!] states that if an assertion *inv* (for invariant) is satisfied after each application of $\mathcal{R}$, then once the iteration has ended, the graph will still satisfy *inv*. Additionally, since $\mathcal{R}$ is applied for as-long-as-possible, we can also deduce that $\mathcal{R}$ is no longer applicable to the graph, hence $\neg\mathrm{App}(\mathcal{R})$ in the postcondition.

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 rule-set calls, that is, sets of conditional rule schemata. We gain from this restriction definability of the transformations Pre and App, but we hope to be able to modify the transformations in the future to allow arbitrary programs as input. However, despite the inconvenience of the restrictions, the computational completeness of the language is not affected, because in [8] 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 2 (Colouring).* Figure 3 is a proof tree for the `colouring` program of Figure 2. 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, the proof tree proves the triple $\{\neg\exists(\textcircled{\texttt{a}} \mid \mathrm{type}(\texttt{a}) \neq \mathrm{int})\}$ `init!; inc!` $\{\forall(\textcircled{\texttt{a}}_1, \exists(\textcircled{\texttt{a}}_1 \mid \texttt{a} = \texttt{b\_c} \wedge \mathrm{type}(\texttt{b}, \texttt{c}) = \mathrm{int})) \wedge \neg\exists(\textcircled{\texttt{x\_i}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y\_i}} \mid \mathrm{type}(\texttt{i}, \texttt{k}, \texttt{x}, \texttt{y}) = \mathrm{int})\}$.

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

$$\begin{aligned}
c &= \neg\exists(\ \text{\textcircled{\texttt{a}}}\ \mid \mathrm{type}(\texttt{a}) \neq \mathrm{int})\\
d &= \forall(\ \underset{1}{\text{\textcircled{\texttt{a}}}}, \exists(\ \underset{1}{\text{\textcircled{\texttt{a}}}} \mid \texttt{a} = \texttt{b\_c} \wedge \mathrm{type}(\texttt{b},\texttt{c}) = \mathrm{int}))\\
e &= \forall(\ \underset{1}{\text{\textcircled{\texttt{a}}}}, \exists(\ \underset{1}{\text{\textcircled{\texttt{a}}}} \mid \mathrm{type}(\texttt{a}) = \mathrm{int}) \vee \exists(\ \underset{1}{\text{\textcircled{\texttt{a}}}} \mid \texttt{a} = \texttt{b\_c} \wedge \mathrm{type}(\texttt{b},\texttt{c}) = \mathrm{int}))\\
\neg\mathrm{App}(\{\texttt{init}\}) &= \neg\exists(\ \text{\textcircled{\texttt{x}}}\ \mid \mathrm{type}(\texttt{x}) = \mathrm{int})\\
\neg\mathrm{App}(\{\texttt{inc}\}) &= \neg\exists(\ \text{\textcircled{\texttt{x\_i}}} \xrightarrow{\texttt{k}} \text{\textcircled{\texttt{y\_i}}} \mid \mathrm{type}(\texttt{i},\texttt{k},\texttt{x},\texttt{y}) = \mathrm{int})\\
\mathrm{Pre}(\texttt{init},e) &= \forall(\ \underset{1}{\text{\textcircled{\texttt{x}}}}\ \ \underset{2}{\text{\textcircled{\texttt{a}}}} \mid \mathrm{type}(\texttt{x}) = \mathrm{int}, \exists(\ \underset{1}{\text{\textcircled{\texttt{x}}}}\ \ \underset{2}{\text{\textcircled{\texttt{a}}}} \mid \mathrm{type}(\texttt{a}) = \mathrm{int})\\
&\quad \vee \exists(\ \underset{1}{\text{\textcircled{\texttt{x}}}}\ \ \underset{2}{\text{\textcircled{\texttt{a}}}} \mid \texttt{a} = \texttt{b\_c} \wedge \mathrm{type}(\texttt{b},\texttt{c}) = \mathrm{int}))\\
\mathrm{Pre}(\texttt{inc},d) &= \forall(\ \underset{1}{\text{\textcircled{\texttt{x\_i}}}} \xrightarrow{\texttt{k}} \underset{2}{\text{\textcircled{\texttt{y\_i}}}}\ \ \underset{3}{\text{\textcircled{\texttt{a}}}} \mid \mathrm{type}(\texttt{i},\texttt{k},\texttt{x},\texttt{y}) = \mathrm{int},\\
&\quad \exists(\ \underset{1}{\text{\textcircled{\texttt{x\_i}}}} \xrightarrow{\texttt{k}} \underset{2}{\text{\textcircled{\texttt{y\_i}}}}\ \ \underset{3}{\text{\textcircled{\texttt{a}}}} \mid \texttt{a} = \texttt{b\_c} \wedge \mathrm{type}(\texttt{b},\texttt{c}) = \mathrm{int}))
\end{aligned}$$

**Fig. 3.** A proof tree for the program `colouring` of Figure 2

The following theorem is our main technical result.

**Theorem 1.** *The proof system comprising* [rule], [ruleset$_1$], [ruleset$_2$], [comp], [cons], [if], *and* [!] *is sound for graph programs in the sense of partial correctness.*

This theorem is proven in [18], by showing the soundness of each of the axioms and inference rules with respect to the structural operational semantics of GP.

## 5 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 by proving the partial correctness of a nondeterministic colouring program.

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 just 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. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
11. 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.
12. 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.
13. 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.

14. Karl-Heinz Pennemann. Resolution-like theorem proving for high-level conditions. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 289–304. Springer-Verlag, 2008.
15. Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.
16. 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.
17. 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.
18. Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2010. To appear.
19. 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.
20. 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.
21. 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.