Electronic Communications of the EASST
Volume 61 (2013)

Selected Revised Papers from the 4th International Workshop on Graph Computation Models (GCM 2012)

Verifying Total Correctness of Graph Programs

Christopher M. Poskitt and Detlef Plump

20 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verifying Total Correctness of Graph Programs

Christopher M. Poskitt[1] **and Detlef Plump**[2]

[1] ETH Zürich, Switzerland

[2] The University of York, UK

Abstract: GP 2 is an experimental nondeterministic programming language based on graph transformation rules, allowing for visual programming and the solving of graph problems at a high-level of abstraction. In previous work we demonstrated how to verify graph programs using a Hoare-style proof calculus, but only partial correctness was considered. In this paper, we add new proof rules and termination functions, which allow for proofs to additionally guarantee that program executions always terminate (weak total correctness), or that programs always terminate and do so without failure (total correctness). We show that the new proof rules are sound with respect to the operational semantics of GP 2, complete for termination, and demonstrate their use on some example programs.

Keywords: graph programs, verification, Hoare logic, total correctness, termination

1 Introduction

The verification of graph transformation systems is an area of active and growing interest, motivated by the many applications of graph transformation to specification and programming. Whilst much of the research in this area has focused on sets of rules or graph grammars (see e.g. [BCK08, BHE09, KE10, CR12]), the challenge of verifying graph-based programming languages is also beginning to be addressed. In particular, Habel, Pennemann, and Rensink [HPR06, HP09] contributed a verification framework – based on weakest preconditions – for a simple graph transformation language, expressing graph properties with nested conditions (a formalism based on graph morphisms). Their language however does not support important practical features such as computation on labels, and their weakest precondition calculus generates infinite preconditions for loops.

In [PP12a] we considered the verification of GP [Plu09], a nondeterministic programming language based on graph transformation. The states are directed labelled graphs, which are manipulated via the application of (conditional) rule schemata. These generalise double-pushout rules with relabelling and expressions. The verification framework of the paper is a Hoare calculus for partial correctness, with which one can prove that programs executed on graphs satisfying given preconditions will only ever result in graphs satisfying given postconditions. However, the calculus cannot be used to prove that such programs do eventually terminate, and cannot be used to prove the absence of failing executions. Addressing these two issues is the focus of this paper.

We define two notions of total correctness: a weaker one accounting for termination, and a stronger one accounting for termination as well as for absence of failures. We define two calculi for these notions of total correctness, using termination functions (that map graphs to natural

numbers) in the new proof rule for the iteration command. We demonstrate the proof calculi on programs that have loops and potential failure points, before proving the calculi to be sound as well as complete for termination.

In contrast to our previous papers, we present the work here in the setting of GP 2 [Plu12] (henceforth referred to as simply GP). This extended version of the language has an improved type system, a marking (shading) mechanism for nodes and edges, a new conditional construct, and a simplified semantics for branching and iteration to support a more efficient implementation. Our previous verification work has been updated in [Pos13] to support these new features, but due to space limitations we cannot present all of the revised definitions here. We attempt to make the intuition behind each concept clear, but refer the interested reader to [Pos13] for the full technical details and further explanations.

Section 2 reviews some technical preliminaries. Section 3 is an informal introduction to graph programs. Section 4 reviews our assertion language and the partial correctness proof rules of our previous calculus. Section 5 formalises the notion of (weak) total correctness and presents new proof rules which allow one to prove these properties. Section 6 demonstrates the use of the new calculi on some example programs. Section 7 presents a proof that the new calculi are sound for (weak) total correctness, and also a proof that the calculi are complete for termination. Finally, we conclude in Section 8.

This paper is a revised and extended version of [PP12b].

2 Preliminaries

Graph transformation in GP is based on the double-pushout (DPO) approach with relabelling [HP02], i.e. an approach in which both node and edge labels can be relabelled. This framework deals with rules containing partially labelled graphs, the definition of which we recall below. In this section we treat the label alphabet as a parameter because we will require two different alphabets for two classes of graphs: "syntactic" graphs labelled with expressions, and "semantic" graphs labelled with lists composed of integers and strings. We also introduce assignments which translate syntactic graphs into semantic graphs.

A *graph* over a label alphabet $\mathscr{C}$ 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, $l_G \colon V_G \to \mathscr{C}$ is the partial[1] node labelling function and $m_G \colon E_G \to \mathscr{C}$ is the (total) edge labelling function (edges can be relabelled by deletion and re-insertion, hence unlabelled edges are not necessary). 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 $\mathscr{G}(\mathscr{C})$ for the set of all totally labelled graphs over $\mathscr{C}$, and $\mathscr{G}(\mathscr{C}_\bot)$ for the set of all graphs over $\mathscr{C}$. The *empty graph*, denoted by $\emptyset$, has empty node and edge sets.

A *graph morphism* $g \colon G \to H$ between graphs G, H in $\mathscr{G}(\mathscr{C}_\bot)$ 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

[1] Unlabelled nodes appear in the interfaces of rule schemata to allow relabelling, see [PP12a, Pos13].

$l_H(g_V(v)) = \bot$ 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 our assertion language contain graphs labelled with expressions, while the graphs on which programs operate are labelled with lists composed of integers and character strings. In both cases nodes and edges can be marked; marked nodes are displayed as shaded, and marked edges are displayed as dashed (see Figure 2). 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 $\mathbb{Z}$ denote the set of integers and Char a finite set of characters. We fix the label alphabet:

$$\mathscr{L} = (\mathbb{Z} \cup \text{Char}^*)^* \times \mathbb{B}$$

where $\mathbb{B} = \{\text{true}, \text{false}\}$, i.e. all sequences over integers and character strings, along with a Boolean value indicating whether the node or edge is marked or not. Occasionally we will refer only to the list component $(\mathbb{Z} \cup \text{Char}^*)^*$, which shall be denoted by $\mathbb{L}$.

The other label alphabet we are using, Label, consists of a mark component and (colon delimited) sequences of arithmetical expressions and strings. These may contain variables from a set denoted by VarId. Variables represent values in $\mathbb{L}$, i.e. lists, and can be constrained in rule schemata to represent integers, strings, or atoms (an integer or a string). These types correspond to the semantic domains in Figure 1, in which we identify atoms and unit-length lists to establish a subtype hierarchy.

Figure 1: Subtype hierarchy for lists

We write $\mathscr{G}(\text{Label})$ to denote the set of all graphs labelled over Label (grammars defining the label alphabet are given in [Plu12, Pos13]). Examples of list components of labels in $\mathscr{G}(\text{Label})$ include `x*5` and `"root":y` (the variable `x` may only be instantiated to integers, whereas `y` be instantiated to any value in $\mathbb{L}$, unless otherwise constrained).

Each graph in $\mathscr{G}(\text{Label})$ represents a possibly infinite set of graphs in $\mathscr{G}(\mathscr{L})$. The latter are obtained by instantiating variables with values from $\mathbb{L}$ and evaluating expressions. An *assignment* is a partial function α: VarId $\to \mathbb{L}$. For a rule schema (see the next section), α must satisfy for all variables `x` with type `int` (resp. `string`, `atom`, `list`), $\alpha(\texttt{x}) \in \mathbb{Z}$ (resp. Char^*, $\mathbb{Z} \cup \text{Char}^*$, $\mathbb{L}$). For assertions (see Section 4), we require that α is *well-typed* for the expressions to which it is applied, i.e. it assigns values to variables of types determined by their contexts. For example, a well-typed assignment for `x+y:z` (with + interpreted as addition) must map `x`, `y` to integers.

Given a (well-typed) assignment α and label $(e\ b)$ with e a list and $b \in \mathbb{B}$, we define $(e\ b)^\alpha = (e^\alpha, b)$ where the value $e^\alpha \in \mathbb{L}$ is inductively defined as follows. If e is the empty list, then e^α is the empty sequence. If e is a numeral or a sequence of characters, then e^α is the integer or character string represented by e. (Note that the empty list and empty character string are distinct values.) If e is a variable identifier, then $e^\alpha = \alpha(e)$. For arithmetic and string expressions, e^α is defined inductively in the usual way. Finally, if e has the form $e_1 : e_2$ with e_1, e_2 list expressions, then $e^\alpha = e_1^\alpha e_2^\alpha$ (the concatenation of the sequences e_1^α and e_2^α). Given a graph G in $\mathscr{G}(\text{Label})$ and an assignment α well-typed for all expressions in G, we write G^α for the graph in $\mathscr{G}(\mathscr{L})$ that is obtained from G by replacing each label l with l^α (note that G^α has the same nodes, edges, source and target functions as G). If $g\colon G \to H$ is a graph morphism with $G, H \in \mathscr{G}(\text{Label})$, then g^α denotes the morphism $\langle g_V, g_E \rangle\colon G^\alpha \to H^\alpha$.

Remark 1 In [Plu12], variables belong to one of four distinct sets of variables – one set for each type – and assignments are families of mappings from the these sets to the appropriate semantic domains. We use a different definition in this paper, taking all variables to be members of the set VarId, and interpreting type declarations in rule schemata as constraints on possible assignments. This definition allows us to treat variables in rule schemata and our assertion language more uniformly, simplifying the presentation of this paper. Additionally, though GP 2 introduced indegree and outdegree functions in expressions, we do not consider them in this paper, as applicability properties of rule schemata that use them cannot be expressed by our assertion language. □

3 Graph Programs

We introduce graph programs informally and by example in this section. For technical details, further examples, and more discussion on the operational semantics, refer to [Plu12].

The "building blocks" of graph programs are (conditional) rule schemata: 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, in that labels contain (sequences of) expressions and relabelling is supported. Expressions may contain variables, which in rule schemata are associated with types integer, string, atom, or list, constraining the possible mappings for assignments. Conditional rule schemata further constrain assignments with a condition: one use is in requiring relations between expressions (e.g. `x < y + z`), but they can also be used to require (the absence of) edges between nodes in a match (e.g. `not edge(1,2)`). As the values of variables at execution are determined by graph matching, we require that expressions in the left graph have a simple shape: (1) expressions contain no arithmetic operators; (2) expressions contain at most one occurrence of a list variable; and (3) each string expression contains at most one occurrence of a string variable.

In Figure 2 we give an example of a conditional rule schema and a possible application of it. The first row of the diagram (in the box) contains the conditional rule schema. There is an identifier, here `bridge`, and a declaration of variables with their types. The left- and right-hand graphs describe the rule with the small numbers indicating which nodes correspond to each other. Here, `bridge` is applied to a path of length two across nodes in which only the first is marked, and across unmarked string-labelled edges, provided that there is not already a direct edge from

the first node to the third (as per the condition). The effect of applying `bridge` is to add a marked edge from the first node to the third, removing the mark from the former whilst adding a mark to the latter, and taking the composition of their list components for the new edge. The conditional rule schema describes an infinite number of "concrete" graph transformation rules with labels fitting the pattern described by the schema. The second row of the diagram shows one such rule, obtained by evaluating expressions according to the assignment:

$$\alpha = \{\mathtt{a} \mapsto \text{“AB”}, \mathtt{b} \mapsto \text{“BC”}, \mathtt{x} \mapsto 0:1:2, \mathtt{y} \mapsto 3, \mathtt{z} \mapsto 4\}$$

which adheres to the constraints of the type declaration. The bottom row shows an application of `bridge` to a graph via the same assignment α and an injective morphism g. It is applied in the double-pushout approach with relabelling [HP02], which intuitively means that nodes can be relabelled in an arbitrary context (edges can simply be deleted and reinserted with the new labels), and that the application is side-effect free (i.e. it is forbidden to delete a node unless the rule schema explicitly deletes all edges it is incident to).

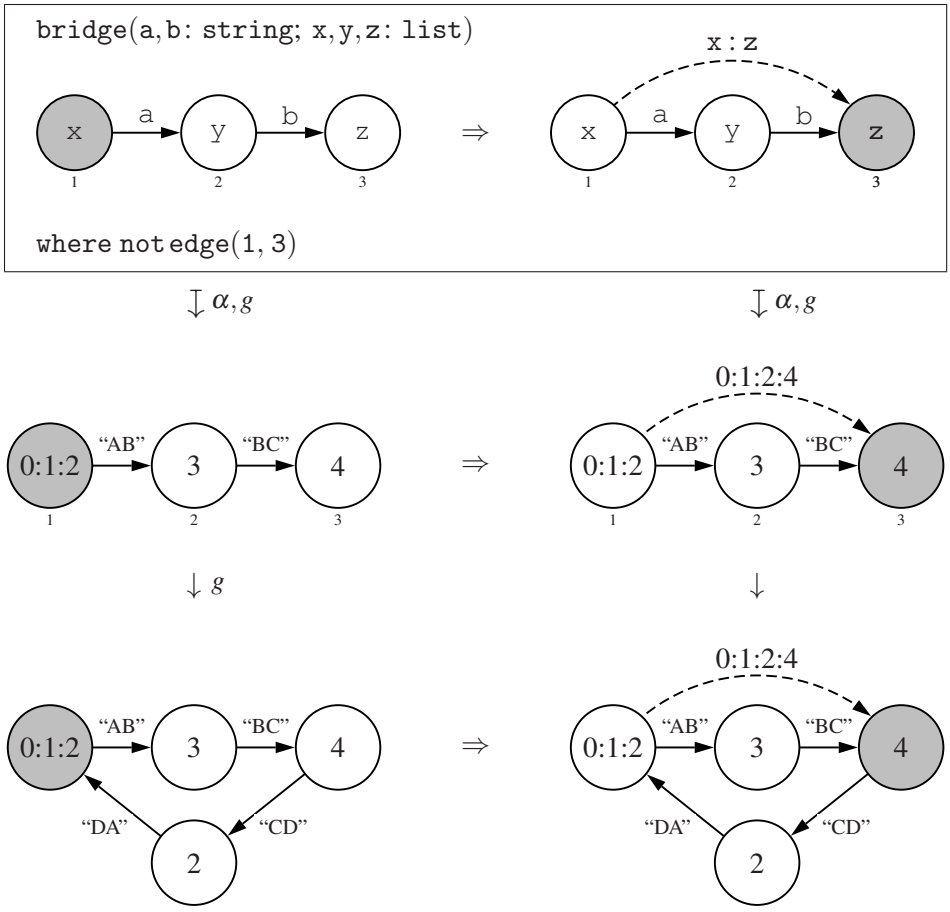


Figure 2: A conditional rule schema and a possible application of it

The application of a rule schema r to a graph $G \in \mathscr{G}(\mathscr{L})$ that yields a graph isomorphic to

$H \in \mathscr{G}(\mathscr{L})$, denoted $G \Rightarrow_r H$, proceeds roughly as follows:

1. Match the left graph L of r with a subgraph of G, ignoring labels, by means of a so-called premorphism $g : L \to G$. (A premorphism is a graph morphism that does not need to preserve labels.)

2. Check whether there is an assignment α of values to all the variables in r (adhering to the declared types) such that after evaluating the expressions in L, g is label-preserving.

3. If r is a conditional rule schema, check that the condition evaluates to true with respect to α and g (conditions are evaluated in the obvious way, with `edge` $(m,n)^{\alpha,g} =$ true for node identifiers m,n if and only if there is an edge with source $g_V(m)$ and target $g_V(n)$).

4. Apply the rule r^α (obtained from r by evaluating[2] all expressions in the left and right graph) to G with match g via the double-pushout approach with relabelling.

We also write $G \Rightarrow_{\mathscr{R}} H$ for a set of (conditional) rule schemata $\mathscr{R}$ if there is some $r \in \mathscr{R}$ such that $G \Rightarrow_r H$.

Declarations of (conditional) rule schemata are, in graph programs, applied according to a number of simple control constructs. GP provides non-deterministic choice, sequential composition, conditional constructs, and as-long-as-possible iteration. We demonstrate these informally with two example programs.

The program `colouring` in Figure 3 produces a colouring (assignment of integers to nodes such that adjacent nodes have different colours), provided that the input graph consists of unmarked items only, and the list components of nodes are atomic. Colours are recorded as so-called tags, i.e. information stored in a label by extending the list component.

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 3 shows two possible executions.

The program `reachable?` in Figure 4 checks whether or not there is a path from one distinguished node (tagged with 1) to another (tagged with 2), again provided that the input graph contains unmarked items only and the list components of nodes are atoms (except for the distinguished nodes). An execution of `reachable?` returns the original input graph if there exists such a path, otherwise it returns the same graph but with an additional edge linking the distinguished nodes. With `propagate!`, the program iteratively tags nodes with 0 that are reachable from the 1-tagged node. An if-then-else conditional is then encountered: if its "guard" `reachable` can be applied (to a copy of the graph), then `skip` is executed; otherwise `addlink`. The idea is as follows: if `reachable` can be applied, then there must be a tagged node adjacent to the second distinguished node, indicating the existence of a path. In this case, `skip` is applied which does not change the graph. If `reachable` cannot be applied, then there must not exist a path, and so `addlink` is applied to add an edge directly between the distinguished nodes. In both cases, the 0-tags used in the computation are removed by the iteration of `undo`.

[2] The evaluation of full GP 2 expressions (with in- and outdegree functions) depends on g as well as α.

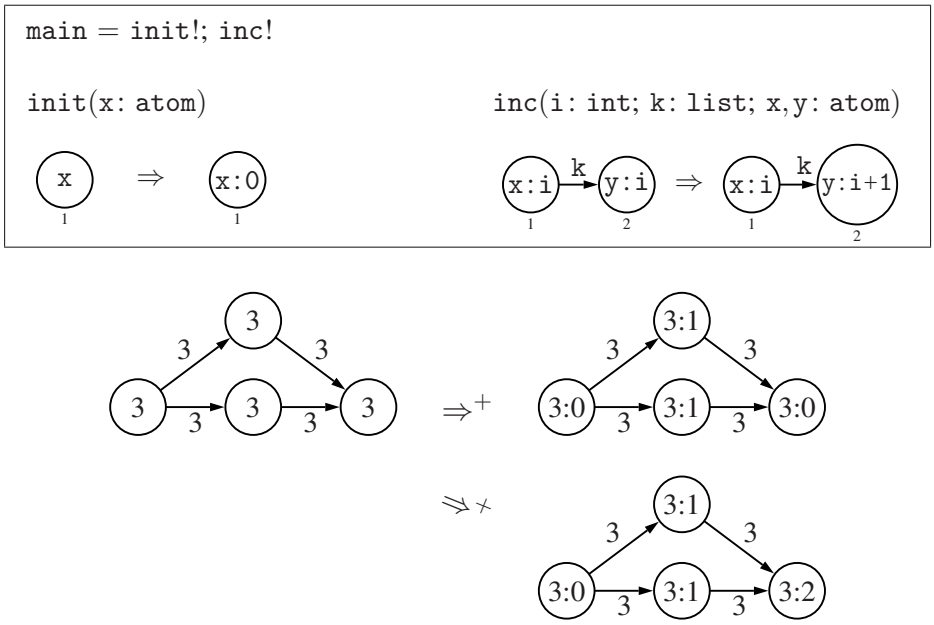


Figure 3: The program `colouring` and two of its executions

The formal semantics of GP is given in the style of structural operational semantics. Inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In our setting, a configuration is either a command sequence (ComSeq) together with a graph (i.e. an unfinished computation), just a graph, or the special element fail (representing a failure state). 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 values fail and $\perp$. The value fail indicates a failed program run whilst $\perp$ indicates a non-terminating or stuck computation. The *semantic function* $[\![\_]\!]$: ComSeq $\rightarrow (\mathscr{G}(\mathscr{L}) \rightarrow 2^{\mathscr{G}(\mathscr{L})\cup\{\text{fail},\perp\}})$ is defined by:

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

where P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \rightarrow \langle P_1, G_1\rangle \rightarrow \langle P_2, G_2\rangle \rightarrow \ldots$, and P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \rightarrow^* \langle Q, H\rangle$ where the rest program Q cannot be executed because no inference rule is applicable. A program can get stuck if it contains a non-terminating subprogram in a loop or in a conditional.

Figure 5 shows the inference rules of commands in GP 2. Each rule consists of a premise and a conclusion separated by a horizontal bar. Both parts contain meta-variables for command sequences and graphs, where $\mathscr{R}$ stands for a rule schema set call, C, P, P', Q stand for command sequences, and G, H stand for graphs in $\mathscr{G}(\mathscr{L})$. Meta-variables are considered to be universally quantified. The notation $G \not\Rightarrow_{\mathscr{R}}$ expresses that for graph G in $\mathscr{G}(\mathscr{L})$ there is no graph H such that $G \Rightarrow_{\mathscr{R}} H$. Derived commands such as `skip` can be expressed by semantically equivalent

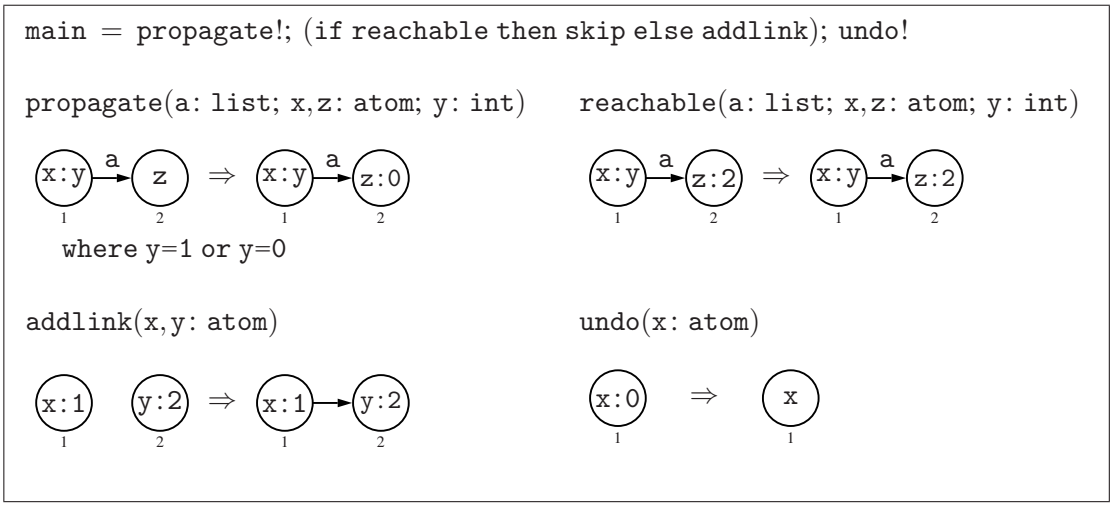


Figure 4: The program `reachable`?

programs made up of core commands only (in this case, the rule schema $\emptyset \Rightarrow \emptyset$). We refer to [Plu12] for details.

4 Proving Partial Correctness

In this section we first review E-conditions, the assertion language of our proof calculi. Then, we review the partial correctness proof calculus presented in previous work (updated for GP 2, e.g. the new [try] proof rule).

Nested graph conditions with expressions (or *E-conditions*) are a morphism-based formalism for expressing both structural properties of graphs and properties about their labels. E-conditions [PP12a] extend the nested conditions of [HPR06] with expressions for labels and *assignment constraints*, which are simple Boolean expressions used to restrict the instantiations of variables to values of particular types, or values that hold in particular relations. A simple example of an E-condition is:

$\neg\exists($ (x) $\mid$ int(x) $)$

which is satisfied by graphs that do not have any unmarked integer-labelled nodes. The formalism combines logical quantifiers with graph structure and constraints on labels: E-conditions demand the (non-)existence of particular subgraphs, subject to some constraint on the labels (the vertical bar can be read as "such that"). More generally, the formalism exploits nesting to allow universally quantified expressions. For example,

$\forall($ (x)$_1$ $\mid$ atom(x) $,\exists($ (x)$_1$ $)\vee\exists($ (x)$_1$ $))$

which expresses that every unmarked atom-labelled node is incident to a loop. Here, the number

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

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

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

$$[\text{if}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle} \qquad [\text{if}_2]\ \frac{\langle C, G\rangle \to^+ \text{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{try}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, H\rangle} \qquad [\text{try}_2]\ \frac{\langle C, G\rangle \to^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

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

Figure 5: Inference rules for core commands

identifies the nodes as being the same; the nesting adds more detail about the required context of the particular subgraph.

Similarly to rule schemata, in checking whether a graph satisfies a property described by an E-condition, a suitable assignment α must be found for the label expressions and assignment constraint. Note however that unlike in rule schemata, we are not declaring types for variables, but rather using predicates about types within assignment constraints. For example, we could write `not int(x)` as an assignment constraint, or even omit type predicates completely.

Due to space limitations we do not give a formal syntax or semantics of assignment constraints (we refer the reader to [PP12a, Pos13]) – there are several examples in this paper however. Example 1 includes a simple assignment constraint, `x > y`. An assignment is well-typed for this if it maps both `x` and `y` to integers. Such an assignment constraint γ is evaluated with respect to a well-typed assignment α, denoted γ^α, by instantiating variables with the values given by α and then replacing function and relation symbols with the obvious functions and relations.

In our formal definition of E-conditions, the part of the formalism immediately after each quantifier is a morphism – not simply a graph. In our examples, we draw only the codomains; in general there are chains of morphisms along the nesting (starting from the empty graph in this paper). We discuss this aspect of E-conditions in more detail shortly.

Definition 1 (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 graph morphism with $P, C \in \mathscr{G}(\text{Label})$, γ 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. □

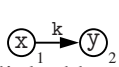
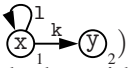
In order to define the satisfaction relation for E-conditions, we first define substitutions to allow the replacement of variables with lists (this is used to enforce equal assignment of variables in the nesting of E-conditions). A *substitution* is a partial function $\sigma\colon \text{VarId} \to \text{List}$. Given a label $(e\ b)$ with e a list and b a mark, σ is *well-typed* for e if it does not replace variables in arithmetic (resp. string) expressions with string (resp. arithmetic) expressions. In this case, the list e^σ is obtained from e by replacing every variable $\mathtt{x}$ for which σ is defined with $\sigma(\mathtt{x})$ (if σ is not defined for a variable $\mathtt{x}$, then $\mathtt{x}^\sigma = \mathtt{x}$). Given a graph G in $\mathscr{G}(\text{Label})$ such that σ is well-typed for all lists in G, we write G^σ for the graph in $\mathscr{G}(\text{Label})$ that is obtained by replacing each list e with e^σ. If $g\colon G \to H$ is a graph morphism between graphs in $\mathscr{G}(\text{Label})$, then g^σ denotes the morphism $\langle g_V, g_E\rangle\colon G^\sigma \to H^\sigma$. A substitution $\sigma\colon \text{VarId} \to \text{List}$ can be applied to an assignment constraint γ, if it is well-typed for all expressions in γ. The resulting assignment constraint, denoted by γ^σ, is simply γ with each expression e replaced by e^σ.

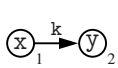
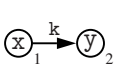
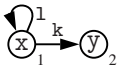
Given an assignment $\alpha\colon \text{VarId} \to \mathbb{L}$, the *substitution* $\sigma_\alpha\colon \text{VarId} \to \text{List}$ *induced by* α maps every variable $\mathtt{x}$ to the expression that is obtained from $\alpha(\mathtt{x})$ by replacing integers and strings with their syntactic counterparts. For example, if $\alpha(\mathtt{x})$ is the integer 23, then $\sigma_\alpha(\mathtt{x})$ is 23 (the syntactic digits). 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 $\sigma_\alpha(\mathtt{x}) = 56\mathtt{:"a":"bc"}$.

The *satisfaction* of E-conditions by injective graph morphisms between graphs in $\mathscr{G}(\mathscr{L})$ is defined inductively. Every such morphism satisfies the E-condition $\mathtt{true}$. An injective graph morphism $s\colon S \hookrightarrow G$ with $S, G \in \mathscr{G}(\mathscr{L})$ *satisfies* the E-condition $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$, denoted $s \models c$, if there exists an assignment α that is well-typed for all expressions in P, C, γ 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 $q \circ a^\alpha = s$, $\gamma^\alpha = \text{true}$, and $q \models (c')^{\sigma_\alpha}$. Here, σ_α is the substitution induced by α, which we require to be well-typed for all expressions in c'. If such an assignment α and morphism q exist, we say that s *satisfies* c *by* α, and write $s \models_\alpha c$. The satisfaction of Boolean formulae over E-conditions is defined inductively, in the obvious way.

For brevity, we write $\mathtt{false}$ for $\neg\mathtt{true}$, $\exists(a \mid \gamma)$ for $\exists(a \mid \gamma, \mathtt{true})$, $\exists(a, c')$ for $\exists(a \mid \mathtt{true}, c')$, 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*.

Example 1 The E-constraint $\forall($  $\mid \mathtt{x} > \mathtt{y}, \exists($  $))$ expresses that every pair of unmarked integer-labelled nodes linked by an unmarked edge with the source label greater than the target label, has an unmarked loop incident to the source node. The (fully) unabbreviated version of the E-constraint is as follows:

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

□

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

Definition 2 (Partial correctness) A graph program P is *partially correct* with respect to a precondition c and postcondition d (both of which are E-constraints), denoted $\models_{\text{par}} \{c\}\ P\ \{d\}$, if for every graph $G \in \mathscr{G}(\mathscr{L})$, $G \models c$ implies $H \models d$ for every graph H in $[\![P]\!]G$. □

In [PP12a] we defined axioms and inference rules for proving partial correctness specifications about graph programs. These are given in Figure 6 (with [try] new for GP 2), where r (resp. $\mathscr{R}$) ranges over conditional rule schemata (resp. sets of conditional rule schemata), c, c', d, d', e, inv over E-constraints, and P, Q over graph programs. Together, the axioms and rules define a proof system for partial correctness. If a Hoare triple $\{c\}\ P\ \{d\}$ can be proved via the axioms and inference rules (by constructing a proof tree, as in Section 6), we write $\vdash_{\text{par}} \{c\}\ P\ \{d\}$. The proof system is sound in the sense of partial correctness, that is, $\vdash_{\text{par}} \{c\}\ P\ \{d\}$ implies $\models_{\text{par}} \{c\}\ P\ \{d\}$ (see [PP12a] for GP 1, and [Pos13] for an analogous GP 2 proof).

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

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathscr{R}}{\{c\}\ \mathscr{R}\ \{d\}} \qquad [!]\ \frac{\{inv\}\ \mathscr{R}\ \{inv\}}{\{inv\}\ \mathscr{R}!\ \{inv \wedge \neg\text{App}(\mathscr{R})\}}$$

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

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

$$[\text{try}]\ \frac{\{c \wedge \text{App}(\mathscr{R})\}\ \mathscr{R}; P\ \{d\},\quad \{c \wedge \neg\text{App}(\mathscr{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ \mathscr{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

Figure 6: Partial correctness proof rules for core commands

Two transformations – App and Pre – appear in the axioms and rules. Intuitively, App takes as input a set $\mathscr{R}$ of conditional rule schemata, and transforms it into an E-condition satisfied only by graphs for which at least one rule schema in $\mathscr{R}$ is applicable. Pre on the other hand constructs an E-condition such that if $G \models \text{Pre}(r,c)$, and the application of r to G results in a graph H, then $H \models c$. Formal constructions of the transformations are omitted from this paper, but can be found in [PP12a] for GP 1 (and for GP 2, in [Pos13]).

We remark that the proof system is for a strict subset of graph programs. Specifically, as-long-as-possible iteration can only be applied to sets of rule schemata, and the guards of conditionals are restricted to sets of rule schemata (in both cases the semantics of GP allows arbitrary pro-

grams). Without this restriction, the proof rules would require an assertion language able to express that an arbitrary program will not fail.

5 Proving Total Correctness

If $\vdash_{\text{par}} \{c\}\ P\ \{d\}$, then if P is executed on a graph G satisfying c, we can be sure that any graph resulting will satisfy d. What we cannot be sure about is whether an execution of P will ever terminate (i.e. whether an execution might diverge or not). Moreover, if an execution of P does in fact terminate, we cannot be sure that it does so without failure. When referring to *total correctness*, we follow [Apt84] in meaning both absence of divergence and failure; and when referring to *weak total correctness*, we mean only absence of divergence.

Definition 3 (Weak total correctness) A graph program P is *weakly totally correct* with respect to a precondition c and postcondition d (both of which are E-constraints), denoted $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, if $\models_{\text{par}} \{c\}\ P\ \{d\}$ and for every graph $G \in \mathscr{G}(\mathscr{L})$ such that $G \models c$, we have $\bot \notin [\![P]\!]G$. □

Definition 4 (Total correctness) A graph program P is *totally correct* with respect to a precondition c and postcondition d (both of which are E-constraints), denoted $\models_{\text{tot}} \{c\}\ P\ \{d\}$, if $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, and for every graph $G \in \mathscr{G}(\mathscr{L})$ such that $G \models c$, we have fail $\notin [\![P]\!]G$. □

A weakly totally correct program executed on a graph satisfying the precondition will either produce an output graph or terminate with failure (it cannot diverge or get stuck). A totally correct program however has the additional guarantee that it will not fail, that is, a graph will eventually result from its execution.

Our proof system for weak total correctness is formed from the proof rules of Figure 6, but with $[!]_{\text{tot}}$ in Figure 7 substituted for [!]. If a triple $\{c\}\ P\ \{d\}$ can be obtained from this proof system, we write $\vdash_{\text{wtot}} \{c\}\ P\ \{d\}$. The issue of termination is localised to the proof rule for as-long-as-possible iteration: $[!]_{\text{tot}}$ has an additional premise to [!] which handles this. It requires, for a given rule schema set, that there is a function assigning natural numbers to graphs such that these naturals are decreasing along rule applications. Such a function # is called a *termination function*. If the #-values of graphs satisfying the invariant *inv* decrease under applications of $\mathscr{R}$, we say that $\mathscr{R}$ is #-*decreasing under inv*. These definitions are given more precisely below.

$$[!]_{\text{tot}}\ \frac{\vdash_{\text{par}} \{inv\}\ \mathscr{R}\ \{inv\},\quad \mathscr{R} \text{ is \#-decreasing under } inv}{\{inv\}\ \mathscr{R}!\ \{inv \wedge \neg \text{App}(\mathscr{R})\}}$$

$$[\text{ruleset}]_{\text{tot}}\ \frac{c \Rightarrow \text{App}(\mathscr{R}),\quad \vdash_{\text{par}} \{c\}\ r\ \{d\} \text{ for each } r \in \mathscr{R}}{\{c\}\ \mathscr{R}\ \{d\}}$$

Figure 7: Total correctness proof rules for two core GP commands

Definition 5 (Termination function; #-decreasing) A *termination function* is a mapping $\# : \mathscr{G}(\mathscr{L}) \to \mathbb{N}$ from (semantic) graphs to natural numbers. Given an E-constraint c, a set of conditional rule schemata $\mathscr{R}$ is *#-decreasing under c* if for all graphs G, H in $\mathscr{G}(\mathscr{L})$ with $G \models c$ and $H \models c$,

$$G \Rightarrow_{\mathscr{R}} H \text{ implies } \#G > \#H.$$

□

In an application of $[!]_{\text{tot}}$, one must find a suitable termination function # that returns smaller natural numbers along the graphs of direct derivations. The problem of deciding whether a set of rule schemata has a termination function or not is undecidable in general [Plu98]. Often however simple termination functions will suffice in several contexts. For example, a useful, intuitive termination function would be one that maps a graph to its size (e.g. total number of nodes and edges). If a rule schemata set is reducing the size of a graph upon each application, then clearly the iteration cannot continue indefinitely, and this is reflected by the output of # tending towards zero. However, in cases when rule schemata are not decreasing the size of the graph, less obvious termination functions may be needed (one such example will be discussed in Section 6). Note that the rule $[!]_{\text{tot}}$ requires only that # is decreasing for graphs that satisfy the invariant *inv*, i.e. it need not be decreasing for other graphs.

Our proof system for total correctness is formed of [comp], [cons], [if], [try], and the proof rules of Figure 7. If a triple $\{c\}\ P\ \{d\}$ can be derived from this proof system, we write $\vdash_{\text{tot}} \{c\}\ P\ \{d\}$. (We do not include a proof rule for a program that is just a single rule schema r, because this case is captured by proving $\vdash_{\text{tot}} \{c\}\ \{r\}\ \{d\}$.) This proof system allows one to prove that all program executions terminate without failure. Essentially, this is achieved by ensuring that the preconditions of rule schema sets imply their applicability, when they are applied outside of iterations or the guards of conditionals. Hence if graphs satisfy the preconditions, by implication the rule schema sets are applicable to those graphs, and thus we can be certain that no execution will fail.

The proof rule $[\text{ruleset}]_{\text{tot}}$ separates the issues of failure and partial correctness. In using the proof rule, one must show (outside the calculus) that the applicability of $\mathscr{R}$ is logically implied by the precondition c. In showing that this premise holds, we can be sure that at least one rule schema in $\mathscr{R}$ can be applied to a graph satisfying c, hence no execution on that graph will fail. Separately, it must be shown that $\vdash_{\text{par}} \{c\}\ r\ \{d\}$ for each $r \in \mathscr{R}$, that is, each rule schema in the set is partially correct with respect to the pre- and postcondition. Together, we derive that every execution of $\mathscr{R}$ will yield a graph, and that the graph will satisfy the postcondition.

The axiom [nonapp] is excluded from our proof system for total correctness, as the specification $\{\neg\text{App}(\mathscr{R})\}\ \mathscr{R}\ \{\text{false}\}$ does not hold in the sense of total correctness. Suppose that it did. Then $\mathscr{R}$ would never fail on graphs satisfying the precondition. But satisfying $\neg\text{App}(\mathscr{R})$ implies that $\mathscr{R}$ fails on that graph – a contradiction.

6 Example Proofs

In this section, we return to the example graph programs from Section 3, and demonstrate how to prove (weak) total correctness properties using our new proof calculi.

First, we revisit the program `colouring` of Figure 3. Though the program contains no failure points (since if a rule schema under as-long-as-possible iteration cannot be applied, the execution simply moves on to the next command), the iteration operator can introduce non-termination. In [PP12a] we proved that `colouring` is partially correct, in the sense that any graph resulting is properly coloured. In Figure 8, we strengthen this to $\vdash_{\text{tot}} \{c\}\ \texttt{colouring}\ \{d \wedge \neg\text{App}(\{\texttt{inc}\})\}$, i.e. if the program is executed on a graph containing only atom-labelled nodes, then (1) a graph will eventually be returned; (2) it will be properly coloured; and (3) for any colour n in the graph, every colour k with $0 \leq k < n$ is also in the graph. (The specification ignores marked nodes and edges for simplicity.) Note that the E-conditions resulting from Pre, implications in instances of [cons], and their justifications, are omitted to preserve space – but can be found in [Pos13].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{Pre}(\texttt{init}, e)\}\ \texttt{init}\ \{e\}}[\text{ruleapp}]}{\vdash_{\text{par}} \{e\}\ \texttt{init}\ \{e\}}[\text{cons}]}{\{e\}\ \texttt{init!}\ \{e \wedge \neg\text{App}(\{\texttt{init}\})\}}[!]_{\text{tot}} \quad X}{\{c\}\ \texttt{init!}\ \{d\}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{Pre}(\texttt{inc}, d)\}\ \texttt{inc}\ \{d\}}[\text{ruleapp}]}{\vdash_{\text{par}} \{d\}\ \texttt{inc}\ \{d\}}[\text{cons}]}{\{d\}\ \texttt{inc!}\ \{d \wedge \neg\text{App}(\{\texttt{inc}\})\}}[!]_{\text{tot}} \quad Y}{\vdash_{\text{tot}} \{c\}\ \texttt{init!; inc!}\ \{d \wedge \neg\text{App}(\{\texttt{inc}\})\}}[\text{comp}]$$

X : `init` is $\#_{\texttt{init}}$-decreasing under e; Y : `inc` is $\#_{\texttt{inc}}$-decreasing under d

$$\begin{aligned}
c &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ ))\\
d &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}))\\
&\quad \wedge \forall(\ \textcircled{b:c}_1 \mid \texttt{atom(b)}, \exists(\ \textcircled{b:c}_1 \mid \texttt{c = 0})\\
&\qquad \vee \exists(\ \textcircled{b:c}_1\ \textcircled{d:c-1} \mid \texttt{atom(d)}\ ))\\
e &= \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{atom(a)}\ )\\
&\qquad \vee \exists(\ \textcircled{a}_1 \mid \texttt{a = b:c and atom(b) and c >= 0}))\\
&\quad \wedge \forall(\ \textcircled{b:c}_1 \mid \texttt{atom(b)}, \exists(\ \textcircled{b:c}_1 \mid \texttt{c = 0})\\
&\qquad \vee \exists(\ \textcircled{b:c}_1\ \textcircled{d:c-1} \mid \texttt{atom(d)}\ ))\\
\neg\text{App}(\{\texttt{init}\}) &= \neg\exists(\ \textcircled{x} \mid \texttt{atom(x)}\ )\\
\neg\text{App}(\{\texttt{inc}\}) &= \neg\exists(\ \textcircled{x:i} \xrightarrow{k} \textcircled{y:i} \mid \texttt{atom(x,y) and int(i)}\ )
\end{aligned}$$

Figure 8: A proof tree for the program `colouring` of Figure 3

The key revision in the proof tree is in the two uses of $[!]_{\text{tot}}$, which unlike its partial correctness counterpart requires the definition of termination functions. For `init`, we define $\#_{\texttt{init}} : \mathscr{G}(\mathscr{L}) \to \mathbb{N}$ to map graphs to the number of their nodes labelled by an atom. The rule schema is clearly $\#_{\texttt{init}}$-decreasing under e, since every application of `init` reduces by one the number of nodes with such labels. The rule schema `inc` however requires a less obvious termination function

$\#_{\texttt{inc}} : \mathscr{G}(\mathscr{L}) \to \mathbb{N}$. For a graph $G \in \mathscr{G}(\mathscr{L})$, we define:

$$\#_{\texttt{inc}}(G) = \sum_{i=0}^{|V_G|} i - \sum_{v \in V_G} \text{colour}(v)$$

where $\text{colour}(v)$ for a node $v \in V_G$ is defined:

$$\text{colour}(v) = \begin{cases} i & \text{if } l_G(v) = x{:}i \text{ with } x \in \mathbb{Z} \cup \text{Char}^*,\ i \in \mathbb{N}; \\ 0 & \text{otherwise.} \end{cases}$$

We show that `inc` is $\#_{\texttt{inc}}$-decreasing under *inv*. Observe that if G is a graph with $\text{colour}(v) = 0$ for every node v in V_G, then for every derivation $G \Rightarrow^*_{\texttt{inc}} H$ there is some $0 \leq k < |V_H|$ such that k is the largest tag in V_H. We obtain an upper bound for the second summation:

$$\sum_{v \in V_H} \text{colour}(v) \leq 0 + 1 + \cdots + (|V_H| - 1) = 0 + 1 + \cdots + (|V_G| - 1) < \sum_{i=0}^{|V_G|} i.$$

Since $\sum_{v \in V_H} \text{colour}(v)$ equals the number of rule schema applications in $G \Rightarrow^*_{\texttt{inc}} H$, it follows that `inc` must eventually terminate (as it approaches the upper bound). By subtracting the summation from the upper bound, we instead have a number decreasing towards 0 after every application of `inc`. Hence $\#_{\texttt{inc}}$ is a suitable termination function, and `inc` is $\#_{\texttt{inc}}$-decreasing under *inv*.

We remark that `inc!` will terminate on any graph – not just those satisfying *inv*. A termination function however is harder to write without the assumptions the invariant allows us to make about the graphs.

Now, we return to the program `reachable?` of Figure 4, which unlike earlier, can fail on some input graphs (in particular, those graphs omitting the pair of 1- and 2-tagged nodes). We give a proof tree[3] for $\vdash_{\text{tot}} \{c\}$ `reachable?` $\{c\}$ in Figure 9, where the E-constraints are as in Figure 10. For clarity, we let `visited(p)` abbreviate:

```
p = a:0 and atom(a)
```

where `a` is a fresh variable.

If the program is executed on a graph that contains only atom-labelled nodes but with one tagged 1 and another tagged 2, then (1) the program is guaranteed to return a graph eventually; and (2) that graph will satisfy the same condition (i.e. an invariant). Again, due to space limitations, we have omitted the implications in instances of [cons] and their justifications. Moreover, we have only provided one of the E-constraints generated by Pre.

In this proof tree, there are simple suitable termination functions $\#_p, \#_u$. We define the termination function $\#_p : \mathscr{G}(\mathscr{L}) \to \mathbb{N}$ (resp. $\#_u$) to return the number of nodes in a graph that are labelled by an atom (resp. number of atom-labelled nodes tagged with a 0). That is, both termination functions exploit that each application of their respective rule schema explicitly reduces the number of remaining matches.

[3] For simplicity we use an obvious additional axiom [skip]: $\vdash_{\text{tot}} \{c\}$ `skip` $\{c\}$.

Let $P = \texttt{if reachable then skip else addlink}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{propagate}, e)\}\ \texttt{propagate}\ \{e\}}\,[\text{ruleapp}]}{\vdash_{\text{par}} \{e\}\ \texttt{propagate}\ \{e\}}\,[\text{cons}] \qquad \texttt{propagate is } \#_p\text{-decreasing under } e}{\{e\}\ \texttt{propagate!}\ \{e \wedge \neg \mathrm{App}(\{\texttt{propagate}\})\}}\,[!]_{\text{tot}}}{\{c\}\ \texttt{propagate!}\ \{e\}}\,[\text{cons}] \qquad \text{Subtree } X}{\vdash_{\text{tot}} \{c\}\ \texttt{propagate!}; P; \texttt{undo!}\ \{c\}}\,[\text{comp}]$$

Subtree X:

$$\dfrac{\dfrac{\dfrac{}{\{e\}\ \texttt{skip}\ \{e\}}\,[\text{skip}]}{\{e \wedge \mathrm{App}(\{\texttt{reachable}\})\}\ \texttt{skip}\ \{e\}}\,[\text{cons}] \qquad \text{Subtree } Y}{\{e\}\ P\ \{e\}}\,[\text{if}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{undo}, e)\}\ \texttt{undo}\ \{e\}}\,[\text{ruleapp}]}{\vdash_{\text{par}} \{e\}\ \texttt{undo}\ \{e\}}\,[\text{cons}] \qquad \texttt{undo is } \#_u\text{-decreasing under } e}{\{e\}\ \texttt{undo!}\ \{e \wedge \neg \mathrm{App}(\{\texttt{undo}\})\}}\,[!]_{\text{tot}}}{\{e\}\ \texttt{undo!}\ \{c\}}\,[\text{cons}]}{\{e\}\ P; \texttt{undo!}\ \{c\}}\,[\text{comp}]$$

Subtree Y:

$$\dfrac{e \wedge \neg \mathrm{App}(\{\texttt{reachable}\}) \Rightarrow \mathrm{App}(\{\texttt{addlink}\}) \qquad \dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{addlink}, e)\}\ \texttt{addlink}\ \{e\}}\,[\text{ruleapp}]}{\vdash_{\text{par}} \{e \wedge \neg \mathrm{App}(\{\texttt{reachable}\}\}\ \texttt{addlink}\ \{e\}}\,[\text{cons}]}{\{e \wedge \neg \mathrm{App}(\{\texttt{reachable}\})\}\ \texttt{addlink}\ \{e\}}\,[\text{ruleset}]_{\text{tot}}$$

Figure 9: Total correctness proof tree for the program `reachable`? of Figure 4

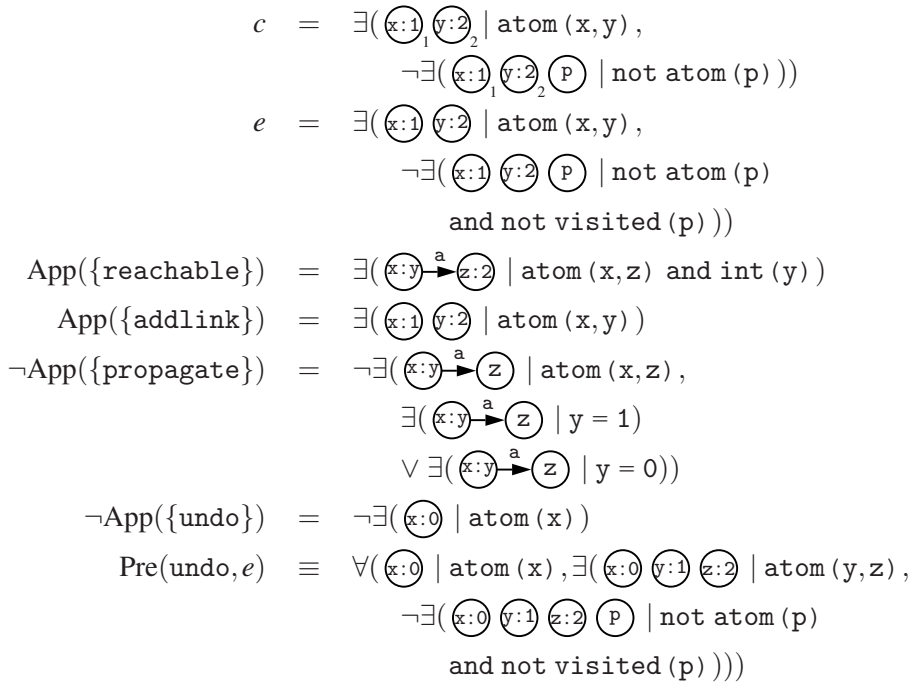


Figure 10: Partial list of E-constraints for Figure 9

The rule schema `addlink` is the only potential failure point of the program, and is addressed in the proof tree by the application of $[\text{ruleset}]_{\text{tot}}$. It must be shown that the precondition at that point implies the applicability of `addlink`. From Figure 10, it is clear that satisfying e is sufficient to deduce the applicability of `addlink`.

7 Soundness and Completeness for Termination

In this section we revise our soundness proof from [PP12a] to account for (weak) total correctness, before showing that any iterating rule schemata set that terminates can be proven to terminate by the rule $[!]_{\text{tot}}$. Soundness is relative to the operational semantics of the language. An updated version of the soundness proof for partial correctness with regards to the GP 2 semantics can be found in [Pos13].

Theorem 1 (Soundness of $\vdash_{\text{wtot}}$) For all graph programs P and E-constraints c, d, we have that $\vdash_{\text{wtot}} \{c\}\ P\ \{d\}$ implies $\models_{\text{wtot}} \{c\}\ P\ \{d\}$. □

Proof. For all weak total correctness proof rules except $[!]_{\text{tot}}$, this follows from (1) the soundness result for partial correctness in [PP12a], and (2) the semantics of graph programs, from which it is clear that only as-long-as-possible iteration can introduce divergence.

Let $\mathscr{R}$ be a set of (conditional) rule schemata, *inv* an E-constraint, and # a termination function. Assume $\vdash_{\text{par}} \{inv\}\ \mathscr{R}\ \{inv\}$. By soundness for partial correctness, we have $\models_{\text{par}} \{inv\}\ \mathscr{R}!\ \{inv \wedge \neg\text{App}(\mathscr{R})\}$. Assume also that $\mathscr{R}$ is #-decreasing under *inv*. By Definition 5, for all graphs

$G, H \in \mathscr{G}(\mathscr{L})$ with $G \models inv$ and $H \models inv$, $G \Rightarrow_{\mathscr{R}} H$ implies $\#G > \#H$. Assume that $\mathscr{R}!$ diverges for any such G. Since $\mathscr{R}$ is #-decreasing under inv, every derivation step yields a graph for which # returns a smaller natural number. Since $\mathscr{R}!$ diverges, there are infinitely many derivation steps. But from any natural n, there are only finitely many smaller numbers. A contradiction. It cannot be the case that $\mathscr{R}!$ diverges from any such G. Hence $\models_{\text{wtot}} \{inv\}\ \mathscr{R}!\ \{inv \wedge \neg \text{App}(\mathscr{R})\}$. □

Theorem 2 (Soundness of $\vdash_{\text{tot}}$) For all graph programs P and E-constraints c, d, we have that $\vdash_{\text{tot}} \{c\}\ P\ \{d\}$ implies $\models_{\text{tot}} \{c\}\ P\ \{d\}$. □

Proof. For the proof rules [comp], [cons], [if], [try], $[!]_{\text{tot}}$, this follows from (1) the soundness of $\vdash_{\text{wtot}}$ (see Theorem 1), and (2) the semantics of graph programs, from which it is clear that these proof rules are sound in the sense of total correctness. What remains to be shown is the soundness of $[\text{ruleset}]_{\text{tot}}$ in the sense of total correctness.

Let $\mathscr{R}$ denote a set of (conditional) rule schemata and c, d denote E-constraints. Assume that $\vdash_{\text{par}} \{c\}\ r\ \{d\}$ for each $r \in \mathscr{R}$. Then by soundness for partial correctness, we have $\models_{\text{par}} \{c\}\ \mathscr{R}\ \{d\}$. Now assume the validity of $c \Rightarrow \text{App}(\mathscr{R})$. Then if a graph $G \in \mathscr{G}(\mathscr{L})$ satisfies c, by assumption it will satisfy $\text{App}(\mathscr{R})$. By Proposition 7.1 of [PP12a] (updated for the GP 2 syntax in [Pos13]), there is a graph H such that $G \Rightarrow_{\mathscr{R}} H$. Then the semantic rule $[\text{call}_1]$ will be applied (and in particular, $[\text{call}_2]$ will not be), hence a graph is guaranteed from the execution and failure is avoided. We yield $\models_{\text{tot}} \{c\}\ \mathscr{R}\ \{d\}$. □

Now, we show that every iterating set of rule schemata that terminates can be proven to terminate using $[!]_{\text{tot}}$, by showing that there always exists a termination function for which the rule schemata set is decreasing under its invariant.

Theorem 3 (Completeness of $[!]_{\text{tot}}$ for termination) Let $\mathscr{R}$ be a set of conditional rule schemata and c be an E-constraint such that for every graph G in $\mathscr{G}(\mathscr{L})$, $G \models c$ implies that $\mathscr{R}!$ cannot diverge from G. Then there exists a termination function # such that $\mathscr{R}$ is #-decreasing under c. □

Proof. Let G be a graph such that $G \models c$. Then there cannot exist an infinite sequence $G \Rightarrow_{\mathscr{R}} G_1 \Rightarrow_{\mathscr{R}} G_2 \Rightarrow_{\mathscr{R}} \ldots$ as otherwise, by the semantics of GP, there would be an infinite sequence $\langle \mathscr{R}!, G\rangle \to \langle \mathscr{R}!, G_1\rangle \to \langle \mathscr{R}!, G_2\rangle \ldots$. To define the termination function #, we show that the length of $\Rightarrow_{\mathscr{R}}$-derivations starting from G is bounded. (Note that, in general, a terminating relation need not be bounded.)

We exploit that $\Rightarrow_{\mathscr{R}}$ is closed under isomorphism in the following sense: given graphs M, M', N and N' such that $M \cong M'$ and $N \cong N'$, then $M \Rightarrow_{\mathscr{R}} N$ implies $M' \Rightarrow_{\mathscr{R}} N'$. Hence we can lift $\Rightarrow_{\mathscr{R}}$ to a relation on isomorphism classes of graphs by defining: $[M] \Rightarrow_{\mathscr{R}} [N]$ if $M \Rightarrow_{\mathscr{R}} N$. Then, since $\mathscr{R}$ is finite, for every isomorphism class $[M]$ the set $\{[N] \mid [M] \Rightarrow_{\mathscr{R}} [N]\}$ is finite.

Now, since there is no infinite sequence of $\Rightarrow_{\mathscr{R}}$-steps starting from $[G]$, it follows from König's lemma [Kön36] that the length of $\Rightarrow_{\mathscr{R}}$-derivations starting from $[G]$ is bounded. (In the tree of all derivations starting from $[G]$, all nodes have a finite degree. Hence the tree cannot be infinite, as otherwise it would contain an infinite derivation.) Hence the length of $\Rightarrow_{\mathscr{R}}$-derivations starting from G is bounded as well. In general, given any graph M in $\mathscr{G}(\mathscr{L})$, let $\#M$ be the length of a longest $\Rightarrow_{\mathscr{R}}$-derivation starting from M if $M \models c$, and $\#M = 0$ otherwise. Then if $M, N \models c$ and

$M \Rightarrow_{\mathscr{R}} N$, we have $\#M > \#N$. Thus $\mathscr{R}$ is #-decreasing under c. □

8 Conclusion

In this paper we have presented two Hoare calculi which allow one to prove (weak) total correctness. Both proof systems have been shown to be sound. We have shown how to reason about termination via termination functions, and shown that the proof rule for termination is complete in the sense that all terminating loops (having a set of conditional rule schemata as the body) can be proven to be terminating. Finally, we have demonstrated the use of the proof systems on two non-trivial graph programs, showing how to prove the absence of divergence and failure.

Future work will explore how to implement the proof calculi in an interactive proof system. A first step towards this is made in [Pos13], in which translations from E-conditions to many-sorted formulae (and back) are defined, providing a suitable front-end logic for an implemented verification system. Future work will also address the question of whether or not the calculi are (relatively) complete. It would also be worthwhile to integrate a stronger assertion language into the calculi that can express non-local properties, such as the hyperedge-replacement conditions of [HR10].

Acknowledgements: We thank the anonymous referees for their comments which helped to improve the final version of this paper. Most of this work was completed whilst the first author was a Ph.D. student at the University of York, where he was supported by a scholarship of the Engineering and Physical Sciences Research Council. Since January 2013 he has been based at ETH Zürich, where he is supported with funding from the European Research Council (ERC grant agreement no. 291389).

Bibliography

[Apt84] K. R. Apt. Ten Years of Hoare's Logic: A Survey Part II: Nondeterminism. *Theoretical Computer Science* 28:83–109, 1984.

[BCK08] P. Baldan, A. Corradini, B. König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation* 206(7):869–907, 2008.

[BHE09] D. Bisztray, R. Heckel, H. Ehrig. Compositional Verification of Architectural Refactorings. In *Proc. Architecting Dependable Systems VI (WADS 2008)*. Volume 5835, pp. 308–333. Springer-Verlag, 2009.

[CR12] S. A. da Costa, L. Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming* 77(4):480–504, 2012.

[HP02] A. Habel, D. Plump. Relabelling in Graph Transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*. Volume 2505, pp. 135–147. Springer-Verlag, 2002.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19(2):245–296, 2009.

[HPR06] A. Habel, K.-H. Pennemann, A. Rensink. Weakest Preconditions for High-Level Programs. In *Proc. International Conference on Graph Transformation (ICGT 2006)*. Volume 4178, pp. 445–460. Springer-Verlag, 2006.

[HR10] A. Habel, H. 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*. Electronic Communications of the EASST 30. 2010.

[KE10] B. König, J. Esparza. Verification of Graph Transformation Systems with Context-Free Specifications. In *Proc. International Conference on Graph Transformation (ICGT 2010)*. Volume 6372, pp. 107–122. Springer-Verlag, 2010.

[Kön36] D. König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae* 8:114–134, 1936.

[Plu98] D. Plump. Termination of Graph Rewriting is Undecidable. *Fundamenta Informaticae* 33(2):201–209, 1998.

[Plu09] D. Plump. The Graph Programming Language GP. In *Proc. Algebraic Informatics (CAI 2009)*. Volume 5725, pp. 99–122. Springer-Verlag, 2009.

[Plu12] D. Plump. The Design of GP 2. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*. Electronic Proceedings in Theoretical Computer Science 82, pp. 1–16. 2012.

[Pos13] C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, The University of York, 2013. To appear.

[PP12a] C. M. Poskitt, D. Plump. Hoare-Style Verification of Graph Programs. *Fundamenta Informaticae* 118(1-2):135–175, 2012.

[PP12b] C. M. Poskitt, D. Plump. Verifying Total Correctness of Graph Programs. In *Proc. International Workshop on Graph Computation Models (GCM 2012)*. 2012. Available from http://gcm2012.imag.fr/.