Verifying Total Correctness of Graph Programs

Christopher M. Poskitt and Detlef Plump
Department of Computer Science
The University of York, UK

Abstract. GP (for Graph Programs) is an experimental nondeterministic programming language for solving problems on graphs and graph-like structures. The language is based on graph transformation rules, allowing visual programming at a high level of abstraction. Previous work has demonstrated how to verify such programs using a Hoare-style proof system, but only partial correctness was considered. In this paper, we extend our calculus with new rules and termination functions, allowing proofs that program executions always terminate (weak total correctness) and that programs always terminate without failing program runs (total correctness). We show that the new proof system is sound with respect to GP’s operational semantics, complete for termination, and demonstrate how it can be used.

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. While much of the research in this area (see e.g. [2,3,8,4]) has focused on sets of rules or graph grammars, the challenge of verifying graph-based programming languages is also beginning to be addressed. In particular, Habel, Pennemann, and Rensink [6,5] contributed a weakest precondition based verification framework for a simple graph transformation language, using nested conditions as the assertions. The language however, does not support important practical features such as computations on labels.

In [12] we consider the verification of GP [11], a nondeterministic graph programming language whose states are directed labelled graphs. These are manipulated directly via the application of (conditional) rule schemata, which generalise double-pushout rules with expressions over labels and relabelling. The framework of [12] is a Hoare-style proof calculus for partial correctness. However, the calculus cannot be used to prove that programs eventually terminate if their preconditions are satisfied, nor that their executions are absent of failure states. This paper aims to address these issues.

We define two notions of total correctness: a weaker one accounting for termination, and a stronger one accounting for that as well as for absence of failures. We define two calculi for these notions of total correctness by modifying our previous proof rules, addressing divergence via the use of termination functions.
that map graphs to natural numbers. We demonstrate the proof calculi on programs that have loops and failure points, before proving them to be sound, and proving that the proof rule for loops is complete for termination.

Section 2 reviews some technical preliminaries; Section 3 is an informal refresher on 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; 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; and finally, Section 8 concludes.

2 Preliminaries

Graph transformation in GP is based on the double-pushout approach with relabelling [7]. This framework deals with partially labelled graphs, whose definition we recall below. We consider two classes of graphs, “syntactic” graphs labelled with expressions and “semantic” graphs labelled with (sequences of) integers and character strings. We also introduce assignments which translate syntactic graphs into semantic graphs, and substitutions which operate on syntactic graphs.

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

A graph morphism \( g : G \to H \) between graphs \( G \) and \( H \) consists of two functions \( g_V : V_G \to V_H \) and \( g_E : E_G \to E_H \) 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 \( l_G(v) \neq \perp \). Morphism \( g \) is an inclusion if \( g(x) = x \) for all nodes and edges \( x \). It is injective (surjective) if \( g_V \) and \( g_E \) are injective (surjective). It is an isomorphism if it is injective, surjective and satisfies \( l_H(g_V(v)) = \perp \) for all nodes \( v \) with \( l_G(v) = \perp \). In this case \( G \) and \( H \) are isomorphic, which is denoted by \( G \cong H \).

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 (sequences of) integers and character strings. We consider graphs of the first type as syntactic objects and graphs of the second type as semantic objects, and aim to clearly separate these levels of syntax and semantics.

Let \( \mathbb{Z} \) be the set of integers and \( \text{Char} \) be a finite set of characters. We fix the label alphabet \( \mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^+ \) of all non-empty sequences over integers and character strings. The other label alphabet we are using, \( \text{Exp} \), consists of underscore delimited sequences of arithmetical expressions and strings. These
may contain (untyped) variable identifiers, the class of which we denote VarId. For example, \(x+5\) and "root_\(y\)" are both elements of \(\text{Exp}\) with \(x, y\) in VarId. (See [12] for a formal grammar defining \(\text{Exp}\).) We write \(\mathcal{G}(\text{Exp})\) for the set of all graphs over the syntactic class \(\text{Exp}\).

Each graph in \(\mathcal{G}(\text{Exp})\) represents a possibly infinite set of graphs in \(\mathcal{G}(\mathcal{L})\). The latter are obtained by instantiating variables with values from \(\mathcal{L}\) and evaluating expressions. An assignment \(\alpha\) is a partial function \(\alpha: \text{VarId} \rightarrow \mathcal{L}\). Given an expression \(e\), \(\alpha\) is well-typed for \(e\) if it is defined for all variables occurring in \(e\) and if all variables within arithmetical (resp. string) expressions are mapped to integers (resp. strings). In this case we inductively define the value \(e^\alpha \in \mathcal{L}\) as follows. If \(e\) is a numeral or a sequence of characters, then \(e^\alpha\) is the integer or character string represented by \(e\). If \(e\) is a variable identifier, then \(e^\alpha = \alpha(e)\).

For arithmetical and string expressions, \(e^\alpha\) is defined inductively in the usual way. Finally, if \(e\) has the form \(t \cdot x_1\) with \(t\) a string or arithmetical expression and \(x_1 \in \text{Exp}\), then \(e^\alpha = t^\alpha \cdot e_1^\alpha\) (the concatenation of the sequences \(t^\alpha\) and \(e_1^\alpha\)). Given a graph \(G\) in \(\mathcal{G}(\text{Exp})\) and an assignment \(\alpha\) that is well-typed for all expressions in \(G\), we write \(G^\alpha\) for the graph in \(\mathcal{G}(\mathcal{L})\) that is obtained from \(G\) by replacing each label \(e\) with \(e^\alpha\) (note that \(G^\alpha\) has the same nodes, edges, source and target functions as \(G\)). If \(g: G \rightarrow H\) is a graph morphism with \(G, H \in \mathcal{G}(\text{Exp})\), then \(g^\alpha\) denotes the morphism \(\langle g^\alpha_v, g^\alpha_E \rangle: G^\alpha \rightarrow H^\alpha\).

A substitution is a partial function \(\sigma: \text{VarId} \rightarrow \text{Exp}\). Given an expression \(e\), \(\sigma\) is well-typed for \(e\) if it does not replace variables in arithmetical expressions with strings (and similarly for string expressions). In this case, the expression \(e^\sigma\) is obtained from \(e\) by replacing every variable \(x\) for which \(\sigma\) is defined with \(\sigma(x)\) (if \(\sigma\) is not defined for a variable \(x\), then \(x^\sigma = x\)). Given a graph \(G\) in \(\mathcal{G}(\text{Exp})\) such that \(\sigma\) is well-typed for all labels in \(G\), we write \(G^\sigma\) for the graph in \(\mathcal{G}(\text{Exp})\) that is obtained by replacing each label \(e\) with \(e^\sigma\). If \(g: G \rightarrow H\) is a graph morphism between graphs in \(\mathcal{G}(\text{Exp})\), then \(g^\sigma\) denotes the morphism \(\langle g^\sigma_v, g^\sigma_E \rangle: G^\sigma \rightarrow H^\sigma\). Given an assignment \(\alpha: \text{VarId} \rightarrow \mathcal{L}\), the substitution \(\sigma_\alpha: \text{VarId} \rightarrow \text{Exp}\) induced by \(\alpha\) maps every variable \(x\) to the expression that is obtained from \(\alpha(x)\) by replacing integers and strings with their syntactic counterparts. For example, if \(\alpha(x)\) is the integer 23, then \(\sigma_\alpha(x) = 23\) (the syntactic digits). Consider another example: if \(\alpha(x)\) is the sequence 56, \(a, bc\), where 56 is an integer and \(a\) and \(bc\) are strings, then \(\sigma_\alpha(x) = 56\, .\, \text"a", \text"bc"\).

3 Graph Programs

We introduce graph programs informally and by example in this section. For technical details, further examples, and the operational semantics, refer to [11].

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 can contain (sequences of) expressions over parameters of type integer or string. Labels in the left graph comprise only variables and constants (no composite expressions) because their
values at execution time are determined by graph matching. The condition of a rule schema is a simple Boolean expression over the variables.

The program **colouring** in Figure 1 produces a colouring (assignment of integers to nodes such that adjacent nodes have different colours) for every (untagged) integer-labelled input graph, recording colours as so-called tags. In general, a tagged label is a sequence of expressions separated by underscores.

The program **colouring** and one of its executions

The program **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 1 shows one possible execution; there is another execution resulting in a graph with three colours.

The program **reachable?** in Figure 2 checks if there is a path from one distinguished node (tagged with 1, i.e. \(x_1\)) to another (tagged with 2, i.e. \(y_2\)), returning the input graph if there is one, otherwise returning the same graph but with a new direct link between them. It repeatedly propagates 0-tagged nodes from the 1-tagged node (and subsequent 0-tagged nodes) for as long as possible via **prop!**. It then tests via **reachable** whether there is a direct link between the distinguished nodes, or a link from a 0-tagged node to the 2-tagged node (indicating a path). If so, nothing happens; otherwise, a direct link is added via **addlink**. In both cases, the 0-tags are removed by the iteration of **undo**.

GP’s formal semantics is given in the style of structural operational semantics. Inference rules (omitted here, but given in [12]) 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
main = prop!; (if reachable then skip else addlink); undo!

prop(a, x, y, z: int) reachable(a, x, y, z: int)

\[
\begin{align*}
&x \cdot y \to a \cdot z \quad \Rightarrow \quad x \cdot y \to a \cdot z_0 \\
&\text{where } y = 1 \text{ or } y = 0
\end{align*}
\]

addlink(x, y: int) undo(x: int)

\[
\begin{align*}
&x_1 \oplus y_2 \Rightarrow x_1 \oplus y_2 \\
&x_0 \Rightarrow x
\end{align*}
\]

Fig. 2. The program reachable?

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

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

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).

4 Proving Partial Correctness

In this section we first review E-conditions, the assertion language of our proof calculus. Then, we review the partial correctness proof calculus presented in previous work.

Nested graph conditions with expressions (or E-conditions) are a morphism-based formalism for expressing graph properties. E-conditions [12] extend the nested conditions of [6] with expressions for labels, and assignment constraints, which are simple Boolean expressions that restrict the types of – and relations between – values that instantiate variables. An assignment constraint \(\gamma\) is evaluated with respect to an assignment \(\alpha\), denoted \(\gamma^\alpha\), by instantiating variables with the values given by \(\alpha\) then replacing function and relation symbols with the obvious functions and relations. Because of space limitations, we do not give a formal syntax or semantics, but refer the reader to [12].
A substitution $\sigma : \text{VarId} \rightarrow \text{Exp}$ can be applied to an assignment constraint $\gamma$, if it is well-typed for all expressions in $\gamma$. The resulting assignment constraint, denoted by $\gamma^\sigma$, is simply $\gamma$ with each expression $e$ replaced by $e^\sigma$.

**Definition 1 (E-condition).** An $E$-condition $c$ over a graph $P$ is of the form true or $\exists (a \mid \gamma, c')$, where $a : P \rightarrow C$ is an injective graph morphism with $P, C \in \mathcal{G}(\text{Exp})$, $\gamma$ is an assignment constraint, and $c'$ is an $E$-condition over $C$. Boolean formulae over $E$-conditions over $P$ yield $E$-conditions over $P$, that is, $\neg c$ and $c_1 \land c_2$ are $E$-conditions over $P$ if $c, c_1, c_2$ are $E$-conditions over $P$. \hfill \Box

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

Example 1. The $E$-constraint $\forall(\xrightarrow{k} \text{V}_1 | x > y, \exists(\xrightarrow{k} \text{V}_2 | \true, \true))$ expresses that every pair of adjacent integer-labelled nodes with the source label greater than the target label has a loop incident to the source node. The unabbreviated version of the condition is as follows:

$$\neg \exists(\emptyset \leftrightarrow \xrightarrow{k} \text{V}_1 \xrightarrow{k} \text{V}_2 | x > y, \neg \exists(\xrightarrow{k} \text{V}_1 \xrightarrow{k} \text{V}_2 | \true, \true)).$$

$\Box$

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

The satisfaction of (resp. application of well-typed substitutions to) Boolean formulae over $E$-conditions is defined inductively, in the usual way.

**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 $G \in \mathcal{G}(\mathcal{L})$, $G \models c$ implies $H \models d$ for every graph $H$ in $[P]_G$. \hfill \Box

The unabbreviated version of the condition is as follows:

$$\neg \exists(\emptyset \leftrightarrow \xrightarrow{k} \text{V}_1 \xrightarrow{k} \text{V}_2 | x > y, \neg \exists(\xrightarrow{k} \text{V}_1 \xrightarrow{k} \text{V}_2 | \true, \true)).$$
In [12] we defined axioms and rules for deriving Hoare triples from graph programs. These are given in Figure 3, where $r$ (resp. $R$) ranges over conditional rule schemata (resp. sets of conditional rule schemata), $c, c', d, d', e, \text{inv}$ 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 derived from the proof system, 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 [12]).

Two transformations – App and Pre – appear in the axioms and rules. Intuitively, App takes as input a set $R$ of conditional rule schemata, and transforms it into an E-condition satisfied only by graphs for which at least one rule schema in $R$ 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 $H \models c$.

We note 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 programs). 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 should $P$ be 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 [1] 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 if for every graph \( G \in \mathcal{G}(L) \) such that \( G \models c \), there is no infinite sequence \( \langle P, G \rangle \to \langle P_1, G_1 \rangle \to \langle P_2, G_2 \rangle \to \cdots \).

**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 if for every graph \( G \in \mathcal{G}(L) \) such that \( G \models c \), there is no derivation \( \langle P, G \rangle \to^\ast \text{fail} \).

Our proof system for weak total correctness is formed from the proof rules of Figure 3, but with \([!]\) in Figure 4 substituted for \([!]\). If a triple \( \models \text{wtot}\{c\} \ P \ \{d\} \) can be derived 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: \([!]\) has an additional premise to \([!]\) which handles this. It requires, for a particular rule schemata set, that there is a function assigning naturals to graphs such that these naturals are decreasing along derivation steps. Such a function \( \# \) is called a termination function. If \( \# \) decreases along derivation steps yielded from applying \( \mathcal{R} \) to graphs satisfying \( \text{inv} \), we say that \( \mathcal{R} \) is \( \#\)-decreasing under \( \text{inv} \).

These definitions are given more precisely below.

\[
\begin{align*}
[!]_{\text{tot}} & \quad \vdash \text{par}\{\text{inv}\} \ \mathcal{R}\{\text{inv}\}, \quad \mathcal{R} \text{ is } \#\text{-decreasing under } \text{inv} \\
& \quad \vdash \text{par}\{c\} \ r \ \{d\} \text{ for each } r \in \mathcal{R} \\
\end{align*}
\]

**Fig. 4.** Total correctness proof rules for two core GP commands

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

\[
G \Rightarrow_{\mathcal{R}} H \text{ implies } \#G > \#H.
\]

\( \Box \)
In an application of \([!]_{\text{tot}}\), one must find a suitable termination function \(\#\) that returns smaller natural numbers along the graphs of direct derivations. A simple, 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 necessarily decreasing the size of the graph, much less trivial termination functions may be needed. We also mention that the problem to decide whether a set of rule schemata is terminating or not, is undecidable in general \([10]\). Note that the rule \([!]_{\text{tot}}\) requires only that \(\#\) is decreasing for graphs that satisfy the invariant \(\text{inv}\), i.e. it need not be decreasing for graphs outside of the particular context.

Our proof system for total correctness is formed of \([\text{comp}]\), \([\text{cons}]\), \([\text{if}]\), and the proof rules of Figure 4. 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 schemata sets imply their applicability. Hence if graphs satisfy the preconditions, by implication the rule schemata 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 \(R\) is logically implied by the precondition \(c\). In showing that this premise holds, we can be sure that at least one rule schema in \(R\) 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 \(r \in R\), 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 \(R\) will yield a graph, and that the graph will satisfy the postcondition.

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

First, we revisit the program \textit{colouring} of Figure 1. 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), the iteration operator can introduce non-termination. In \([12]\) we proved that \(\vdash_{\text{par}} \{c\} \text{colouring} \{d \land\)
\[ \neg \text{App}\{\text{inc}\}\}, \text{where } c \text{ expresses that every node is integer-labelled, and } d \land \neg \text{App}\{\text{inc}\}\text{ expresses that the graph is properly coloured. In Figure } 5 \text{ we strengthen this to } \Gamma_{\text{tot}} \{c\} \text{ colouring } \{d \land \neg \text{App}\{\text{inc}\}\}, \text{i.e. if the program is executed on a graph containing only integer-labelled nodes, then a graph will eventually be returned and that graph will be properly coloured. 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 [12].}

\[ \begin{align*}
\text{[ruleapp]} & \quad \frac{\{\text{Pre}\{\text{init},e\}\} \text{ init } \{e\}}{\Gamma_{\text{par}} \{e\} \text{ init } \{e\}} \\
\text{[cons]} & \quad \frac{\{e\} \text{ init } \{e\} \{c \land \neg \text{App}\{\text{init}\}\}}{\Gamma_{\text{par}} \{d\} \text{ init } \{d\}} \\
\text{[comp]} & \quad \frac{\{c\} \text{ init } \{d\}}{\Gamma_{\text{tot}} \{c\} \text{ init } \{d\}} \\
\end{align*} \]

\[ X : \text{init is } \#_1\text{-decreasing under } e; \ Y : \text{inc is } \#_2\text{-decreasing under } d \]

\[ c = \neg \exists( \exists \mid \text{not type}(a) = \text{int}) \]
\[ d = \forall( \exists, \exists( \exists, \exists \mid a = b \land \text{type}(b, c) = \text{int}) \]
\[ e = \forall( \exists, \exists( \exists \mid \text{type}(a) = \text{int}) \]
\[ \land \exists( \exists \mid a = b \land \text{type}(b, c) = \text{int}) \]
\[ \neg \text{App}\{\text{init}\} = \neg \exists( \exists \mid \text{type}(x) = \text{int}) \]
\[ \neg \text{App}\{\text{inc}\} = \neg \exists( \exists, k \rightarrow \exists \mid \text{type}(i, k, x, y) = \text{int}) \]

\[ \text{Fig. 5. A proof tree for the program colouring of Figure 1} \]

The key revision in the proof tree is in the two uses of \[\Gamma_{\text{tot}}\], which unlike its partial correctness counterpart requires the definition of termination functions. For \text{init}, we define \[#_1 : G(L) \rightarrow \mathbb{N}\] to map graphs to the number of their nodes labelled by a single integer. The rule schema is clearly \#-decreasing under \text{init}, since every application of \text{init} reduces by one the number of nodes with such labels. The rule schema \text{inc} however requires a less obvious termination function \[#_2 : G(L) \rightarrow \mathbb{N}\]. For a graph \(G \in G(L)\), we define:

\[ \#_2 G = \sum_{i=0}^{\left| V_G \right|-1} i - \sum_{v \in V_G} \text{tag}(v) \]

where \text{tag}(v) for a node \(v\) returns the tag of its label (that is, the second element of a sequence \(x_i\)). We show that \text{inc} is \#-decreasing under \(d\) (rather, under
any E-condition). Observe that if $G$ is a graph with $\text{tag}(v) = 0$ for every node $v$, then for every derivation $G \Rightarrow^{*_{\text{inc}}} H$ there is some $0 \leq k \leq |V_H| - 1$ such that $k$ is the largest tag in $V_H$. We obtain an upper bound for the second summation:

$$\sum_{v \in V_H} \text{tag}(v) < 1 + 2 + \cdots + |V_H| = 1 + 2 + \cdots + |V_G|.$$ 

Since this summation equals the number of rule schema applications in $G \Rightarrow^{*_{\text{inc}}} H$, by subtracting it from the upper bound, we have a termination function that decreases towards 0 after every application of $\text{inc}$, hence is suitable for our proof tree.

Now, we return to the program $\text{reachable}?$ of Figure 2, 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 for the program in Figure 7 where the E-conditions are as in Figure 6 showing $\vdash_{\text{tot}} \{c \land d\} \text{reachable}? \{c \land d\}$. This means that if the program is executed on a graph that contains only integer-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 $\lbrack\text{cons}\rbrack$ and their justifications. We have also omitted from Figure 6 the E-conditions $\text{Pre}\{\text{addlink}, d \land e\}$ and $\text{Pre}\{\text{undo}, d \land e\}$. Rather, we note that these are very similar to $\text{Pre}\{\text{prop}, d \land e\}$ which is given.

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

The rule schema $\text{addlink}$ is the program’s only potential failure point, and is addressed in the proof tree by the application of $\lbrack\text{ruleset}\rbrack_{\text{tot}}$. It must be shown that the precondition at that point implies the applicability of $\text{addlink}$. From Figure 6 it is clear that satisfying $\varepsilon$ is sufficient to deduce the applicability of $\text{addlink}$.

### 7 Soundness and Completeness for Termination

In this section we revise our soundness proof from [12] to account for (weak) total correctness, before showing that any iterating rule schemata set that terminates can be proven to terminate by the rule $\lbrack!\rbrack_{\text{tot}}$. The proofs use GP’s semantic inference rules which are given in [12].

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

1 For simplicity we use an obvious additional axiom $\vdash_{\text{tot}} \{c\} \text{skip} \{c\}$. We could have used the core proof rules since $\text{skip}$ is syntax for the rule schema $\emptyset \Rightarrow \emptyset$. 


\( c = \exists (x, y, z \mid \text{type}(x, y) = \text{int}, \neg \exists (x, y, z \mid \text{not type}(x) = \text{int})) \)

\( d = \neg \exists (x, y, z \mid \text{not } x = y, z \text{ and not } \text{type}(x) = \text{int}) \)

\( e = \exists (x, y, z \mid \text{type}(x, y) = \text{int}) \land \neg \exists (x, y, z, p, q \mid \text{not } q = 0) \)

\[
\text{App} \{\text{reachable}\} = \exists (x, y, z, a \mid \text{type}(a, x, y, z) = \text{int})
\]

\[
\text{App} \{\text{addlink}\} = \exists (x, y, z \mid \text{type}(x, y) = \text{int})
\]

\[
\neg \text{App} \{\text{prop}\} \equiv \neg \exists (x, y, z, a \mid \text{type}(a, x, y, z) = \text{int} \land (y = 1 \text{ or } y = 0))
\]

\[
\neg \text{App} \{\text{undo}\} = \neg \exists (x \mid \text{type}(x) = \text{int})
\]

**Fig. 6.** Partial list of E-conditions for Figure 7

**Proof.** For all weak total correctness proof rules except \(\lceil\lceil\text{![tot]}\rceil\rceil\), this follows from (1) the soundness result for partial correctness in [12], and (2) the semantics of graph programs, from which it is clear that only as-long-as-possible iteration can introduce divergence.

Let \( R \) be a set of (conditional) rule schemata, \( \text{inv} \) an E-constraint, and \( \# \) a termination function. Assume \( \vdash_{\text{par}} \{\text{inv}\} R \{\text{inv}\} \). By soundness for partial correctness, we have \( \vdash_{\text{par}} \{\text{inv}\} R! \{\text{inv} \land \neg \text{App}(R)\} \). Assume also that \( R \) is \#-decreasing under \( \text{inv} \). By Definition 4 for all graphs \( G, H \in G(L) \) with \( G \models \text{inv} \) and \( H \models \text{inv} \), \( G \Rightarrow_R H \) implies \( \#G > \#H \). Assume that \( R \) diverges for any such \( G \). Since \( R \) is \#-decreasing under \( \text{inv} \), every derivation step yields a graph for which \( \# \) returns a smaller natural number. Since \( R \) 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 \( R \) diverges from any such \( G \). Hence \( \models_{\text{wtot}} \{\text{inv}\} R! \{\text{inv} \land \neg \text{App}(R)\} \).

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

**Proof.** For the proof rules \( \lceil\lceil\text{comp}\rceil\rceil, \lceil\lceil\text{cons}\rceil\rceil, \lceil\lceil\text{if}\rceil\rceil, \lceil\lceil\text{![tot]}\rceil\rceil \), 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 \( \lceil\lceil\text{ruleset}\rceil\rceil_{\text{tot}} \) in the sense of total correctness.

Let \( R \) denote a set of (conditional) rule schemata and \( c, d \) denote E-constraints. Assume that \( \vdash_{\text{par}} \{c\} r \{d\} \) for each \( r \in R \). Then by soundness for partial cor-
Let $P = \text{if reachable then skip else addlink}$

Subtree $X$:

Subtree $Y$:

Fig. 7. Total correctness proof tree for the program reachable? of Figure 2
rectness, we have $|=_{\text{par}} \{c\} \ R \ \{d\}$. Now assume the validity of $c \Rightarrow \text{App}(R)$. By Proposition 7.1 of [12], there is a graph $H$ such that $G \Rightarrow_R H$. Then the semantic rule $[\text{Call}_1]_{\text{SOS}}$ will be applied (and in particular, $[\text{Call}_2]_{\text{SOS}}$ will not be), hence a graph is guaranteed from the execution and failure is avoided. We yield $|=_{\text{tot}} \{c\} \ R \ \{d\}$.

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 $R$ be a set of conditional rule schemata and $c$ be an E-constraint such that for every graph $G$ in $G(L)$, $G\models c$ implies that $R!$ cannot diverge from $G$. Then there exists a termination function $#$ such that $R$ is $#$-decreasing under $c$.

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

We exploit that $\Rightarrow_R$ 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 $M \Rightarrow_R N$ implies $M' \Rightarrow_R N'$. Hence we can lift $\Rightarrow_R$ to a relation on isomorphism classes of graphs by defining: $[M] \Rightarrow_R [N]$ if $M \Rightarrow_R N$. Then, since $R$ is finite, for every isomorphism class $[M]$ the set $\{[N] \mid [M] \Rightarrow_R [N]\}$ is finite.

Now, since there is no infinite sequence of $\Rightarrow_R$-steps starting from $[G]$, it follows from König’s lemma [9] that the length of $\Rightarrow_R$-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 $\Rightarrow_R$-derivations starting from $G$ is bounded as well. In general, given any graph $M$ in $G(L)$, let $\#M$ be the length of a longest $\Rightarrow_R$-derivation starting from $M$ if $M \models c$, and $\#M = 0$ otherwise. Then if $M, N \models c$ and $M \Rightarrow_R N$, we have $\#M > \#N$. Thus $R$ 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 was made in [13], where translations from E-conditions to many-sorted formulae (and back) were 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.

Acknowledgements. We are grateful to the anonymous referees for their helpful comments.

References