Towards the Verification of Graph Programs

Qualifying Dissertation

Christopher M. Poskitt

9th September 2010

Abstract

GP (for Graph Programs) is an experimental programming language which allows one to manipulate graphs at a very high level of abstraction. There are numerous applications for graph programs: from solving graph problems without the need to consider low level data structures, to specifying the operational behaviour of systems, to simulating the behaviour of pointers. To reason about the correctness of such programs however, one would need to construct an ad hoc proof on a case-by-case basis.

We are hoping to address this situation by developing verification techniques for graph programs. This poses a number of challenges given the differences between GP and traditional programming languages. GP's global program states are simply graphs — how do we precisely describe and reason about their properties? Furthermore, these states (i.e. graphs) are changed by the nondeterministic application of graph transformation rules — how do we guarantee that any graph resulting exhibits some desired property?

In this report, we begin to address the challenges involved in verifying graph programs. We do so by reviewing literature relevant to our task, and then by presenting some preliminary results in the form of a Hoare logic proof system for graph programs.

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

Rule-based graph transformation has been studied since the 1970s, motivated by its many applications to programming and specification (see the handbook [Roz97, HER99a, HER99b]). Graphs are a natural visualisation of many things, from pointer structures to class diagrams, and it is desirable to have a way of manipulating them, particularly when the things we are visualising are dynamic. Recent years have seen increased interest in graph computation models that facilitate such manipulations.

GP (for Graph Programs) [Plu09] is a graph-based programming language, which allows one to write graph manipulation programs at a very high level of abstraction. Graph programs comprise a set of rule schemata (descriptions of single step graph transformations), and a command sequence directing their nondeterministic application to a graph by some simple control constructs. Despite its simplicity, GP is rather powerful in that it can implement every computable function on graphs [HP01]. The language is not just theoretical — a prototype implementation exists, that faithfully matches its semantics.

Many of the applications of graph programs — for example, visualising a system's state as a graph and using graph programs to model its operational behaviour — would benefit from techniques for proving correctness. Suppose that we have a graph program that specifies, at a high level, the operation of logging a user out of a session-based system. Can we prove that the design of the operation deletes session nodes properly? Suppose we have a program that implements a graph algorithm, such as one that computes a graph colouring. Can we prove that the resulting graphs are always properly coloured?

Up to now, research has tended to focus on proving the correctness of graph grammars and sets of graph transformation rules (see, for example, [RSV04, BCK08, KK08, BHE09, HP09]). A first step towards verifying graph programs was taken by Habel et al. [HPR06], who followed Dijkstra's weakest preconditions approach. However, much work remains to be done.

This report begins to address the challenge of graph program verification. We review the fundamentals of graph transformation and graph programming, before turning our attention to a number of verification techniques for imperative programming languages, describing how we might adapt them to the domain of graphs. We go on to describe a number of graph specification formalisms for facilitating reasoning on the states of graph programs (which are simply graphs), before describing some of the research already undertaken in the area of graph transformation and graph program verification. The report also presents some preliminary results, in the form of a Hoare logic proof system for graph programs, and finishes with a proposal for future research.

1.1 Structure of the Report

The rest of the report is structured into three parts, as follows:

Literature Review: Here, we present a survey of the literature in our area of research. We begin in Chapter 2 by reviewing some fundamentals of graph transformation, before turning out attention to graph programming in Chapter 3 (primarily in GP, but we also discuss some other graph transformation languages). In Chapter 4, we consider some techniques for verifying programs in imperative programming languages, and discuss the challenges involved in adapting them to the domain of graphs. In Chapter 5, we look at specification formalisms and logics for describing graph properties, to help us to reason about the states of graph programs (which are graphs). Finally, in Chapter 6, we review some previous work for verifying graph transformation systems and graph programs.

Preliminary Results: In this part of the report, we present a sound Hoare logic proof system for showing the partial correctness of graph programs.

Research Proposal: Here, we suggest a number of lines of research, and discuss how the rest of the research programme might be scheduled.

The reader is assumed to be familiar with a little graph theory, in particular, basic graph properties like connectivity. Many introductory texts are available; two suitable examples are [Har69, Wil85].

1.2 Declaration

Section 3 and Chapter 7 of this report are largely based upon (and in parts follow word for word) the accepted publications [PP10a, PP10c], and a long version of the first paper [PP10b].

Part II

Literature Review

Chapter 2

Fundamentals of Graph Transformation

We aim here to introduce some fundamentals of graph transformation, which are required in the later chapters of this report. In Section 2.1 we first define graphs, before defining the structure preserving graph morphisms we frequently use to relate them. We then proceed to introduce our chosen framework for graph transformation — the double-pushout approach — in Section 2.2, before describing a variant of it that facilitates node relabelling.

We refer the reader who is interested in studying the theory in more depth to the monograph by Ehrig et al. [EEPT06].

2.1 Graphs and Graph Morphisms

We use a definition of graphs in which edges are directed, nodes and edges are labelled, and parallel edges are allowed to exist. In general, we consider graphs that are totally labelled[1]. However, the framework for rule application in GP requires the node labelling function to be partial (this will be explained in Section 2.2.3).

Definition 1 (Label alphabet) A label alphabet $\mathcal{L} = \langle \mathcal{L}_V, \mathcal{L}_E \rangle$ is a pair comprising a set $\mathcal{L}_V$ of node labels and a set $\mathcal{L}_E$ of edge labels.

Definition 2 (Graph) A graph over a label alphabet $\mathcal{L}$ is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ comprising a finite set V_G of nodes, a finite set E_G of edges, source and target functions $s_G, t_G \colon E_G \to V_G$, a partial node labelling function $l_G \colon V_G \to \mathcal{L}_V$, and a total edge labelling function $m_G \colon E_G \to \mathcal{L}_E$. If $V_G = \emptyset$, then G is the empty graph, which we denote by $\emptyset$.

Given a node $v \in V_G$, 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.

[1] If we are working in a problem domain where labels play no role, we can label all nodes and edges with the blank label $\square$, which is usually not drawn.

The reader might be more familiar with graphs defined as an ordered pair, $G = \langle V, E \rangle$, where $E \subseteq V \times V$. The reason we deviate from this traditional definition is to allow parallel edges to exist in graphs.

Example 1 Consider the graph $G = \langle \{1, 2, 3\}, \{a, b, c\}, (a \mapsto 2, b \mapsto 2, c \mapsto 2), (a \mapsto 1, b \mapsto 3, c \mapsto 3), (1 \mapsto \alpha, 2 \mapsto \alpha, 3 \mapsto \gamma), (a \mapsto \square, b \mapsto \square, c \mapsto \square) \rangle$ over the label alphabet $\mathcal{L} = \langle \{\alpha, \beta, \gamma\}, \{\square\} \rangle$. Figure 2.1 is a picture of G, and represents its isomorphism class. We follow the convention of drawing circles for nodes and arrows for edges. Node labels are written inside the circles, and edge labels next to the arrows. Node and edge identifiers from V_G and E_G are not written, and neither is the blank label $\square$.

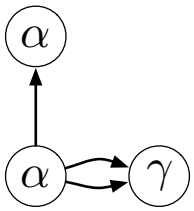


Figure 2.1: A graph

We often need to be able to relate graphs in a formal way. For this purpose, we use *graph morphisms*, which are structure preserving mappings from the nodes and edges of one graph to another. Graph morphisms are ubiquitous in the theory of graph transformation, from their role in the construction of rules (Section 2.2), to their appearance in the underlying theory of the nested condition specification formalism (Section 5.1).

Definition 3 (Graph morphism) Given graphs G, H over $\mathcal{L}$, a graph morphism $f : G \to H$ is a pair of mappings $\langle f_V : V_G \to V_H, f_E : E_G \to E_H \rangle$ that preserve the source, target, and labelling functions. That is, $s_H \circ f_E = f_V \circ s_G$, $t_H \circ f_E = f_V \circ t_G$, $m_H \circ f_E = m_G$, and $l_H(f_V(v)) = l_G(v)$ for all nodes v for which $l_G(v)$ is defined.

As usual, $\circ$ denotes function composition. Given graph morphisms $f : F \to G$ and $g : G \to H$, the *composition* $g \circ f : F \to H$ is defined $g \circ f = \langle g_V \circ f_V, g_E \circ f_E \rangle$.

A graph morphism f is *injective* (*surjective*) if both f_V and f_E are injective (surjective); injective morphisms are usually denoted by a hooked arrow, $\hookrightarrow$. A graph morphism $f : A \to B$ is an *isomorphism* if it is both injective and surjective (i.e. a bijection); in this case, graphs A and B are said to be *isomorphic*, denoted by $A \cong B$. A graph morphism $f : A \to B$ is an *inclusion* if $f(x) = x$ for every node and edge $x \in A$.

Example 2 Consider Figure 2.2, which shows the two possible morphisms from the graph on the left to the graph on the right (one injective, and one not). The small numbers next to the nodes identify the mappings of the morphism (in the non-injective morphism, nodes 1 and 2 are merged).

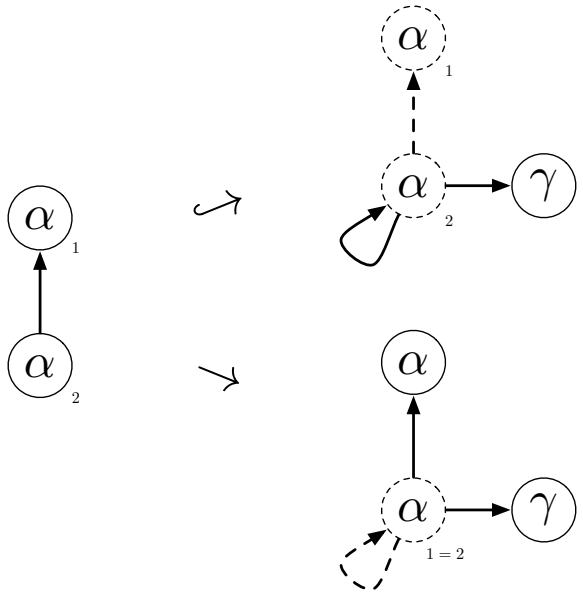


Figure 2.2: An injective and a non-injective graph morphism

2.2 Double-Pushout Approach

Fundamentally, in graph transformation we want to be able to write *rules* (or *productions*) of the form $L \rightsquigarrow R$, where L is some graph to be matched and R is some graph to replace the match. One can view this as a generalisation of rules in Chomsky string grammars, which take the form $\alpha \to \beta$ with strings α, β. Applied to some string w, the rule would replace some substring of w matching α with β. Applying a rule $L \rightsquigarrow R$ to a graph G is not quite as trivial. What is a "match" of L in G? What happens when a rule deletes nodes but not all of the edges to which they are incident? How is R connected, or glued, to the graph G being transformed?

The fact that these questions, and others, do not have obvious answers, has led years of research into frameworks for applying rules to graphs. The framework we focus on is the *double-pushout (DPO) approach*, first described by Ehrig, Pfender, and Schneider in [EPS73], and more recently treated in [CMR+97, HMP01, EEPT06]. The DPO approach finds widespread use in graph transformation, from graph grammars to graph programs; GP uses a variation of the approach to allow relabelling of nodes, described in Section 2.2.3.

The DPO approach is an algebraic approach to graph transformation, treating graphs as algebras, and defining direct derivations by an algebraic construction modelled as two pushouts [CMR+97]. These pushouts are formed in the category **Graphs** of graphs and graph morphisms, the idea being, that general results from category theory can be applied to this approach to graph transformation.

We introduce the DPO approach first by defining rules and matches, then by defining pushouts and using them to construct direct derivations.

2.2.1 Rules and Matches

A *rule* $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$ over $\mathcal{L}$ comprises totally labelled graphs L, K, R, and inclusions $K \hookrightarrow L$, $K \hookrightarrow R$. We call L the *left-hand side*, R the *right-hand side*, and K the *interface* of rule r. For brevity, we will often write $r : \langle L \Rightarrow R \rangle$.

L describes what is to be matched, and R describes what it should be replaced with. The interface graph K describes a graph part which is required to exist for the rule to be applicable, but is not actually changed by the application of the rule. $L - K$ describes the graph part to be deleted, and $R - K$ describes the graph part to be added.

Given rule r and a totally labelled graph G, an injective graph morphism $g : L \hookrightarrow G$ is a *match* for r if it satisfies the *dangling condition*.

Definition 4 (Dangling condition) Given a rule $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$, graph G, and injective graph morphism $g : L \hookrightarrow G$, the dangling condition states that no edge in $G - g(L)$ is incident to any node in $g(L - K)$.

In other words, we forbid g to be a match for r if there are nodes in its image, designated by r for deletion, that are incident to edges not also designated for deletion.

Remark 2.2.1 The requirement that the graphs in rules are all totally labelled makes it difficult to relabel nodes. One solution is to delete a node and recreate it with a new label. However, such a rule will not always be applicable due to the dangling condition. Section 2.2.3 presents a modification to the DPO approach to allow relabelling in an arbitrary context, in part by allowing the nodes in K to be partially labelled. Fortunately, edges need not be relabelled in either approach, as they can always be deleted and recreated in an arbitrary context.

2.2.2 Pushouts and Direct Derivations

We first review pushouts in this section, the gluing construction for graphs in the DPO approach. We follow this by reviewing direct derivations, a construction comprising two pushouts, for generating a graph from another via a rule.

Definition 5 (Pushout) Given graph morphisms $A \to B$ and $A \to C$, the *pushout* (1) of these morphisms is formed by the graph D and graph morphisms $B \to D$ and $C \to D$ as in Figure 2.3 if the following properties are satisfied:

Commutativity. $A \to B \to D = A \to C \to D$.

Universal Property. For all graph morphisms $B \to D'$ and $C \to D'$ such that $A \to B \to D' = A \to C \to D'$, there is a unique graph morphism $D \to D'$ such that $B \to D \to D' = B \to D'$ and $C \to D \to D' = C \to D'$.

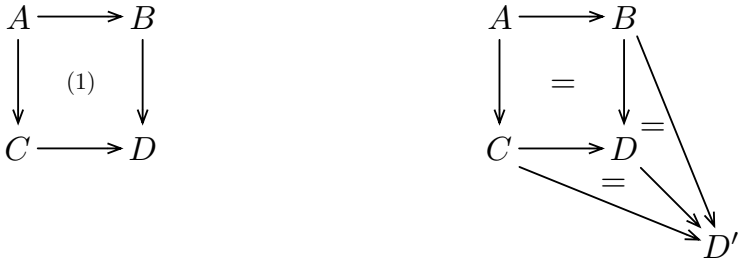


Figure 2.3: A pushout, and the universal property of pushouts

Definition 5 leads to pushouts having a number of properties. First, every item in D has a preimage in B or C (that is, every node and edge in D can be found in either B or C; or both, if they are also in A). Secondly, if $A \to B$ is injective (surjective), then $C \to D$ is also injective (surjective). A final property of interest is the uniqueness of D, following from the universal property of pushouts. A graph D' together with morphisms $B \to D'$ and $C \to D'$ is a pushout of $A \to B$ and $A \to C$ iff there is an isomorphism $D \to D'$ such that $B \to D \to D' = B \to D'$ and $C \to D \to D' = C \to D'$.

Example 3 Figure 2.4 depicts a simple pushout, in which all the morphisms are injective. All nodes and edges are labelled with the blank symbol.

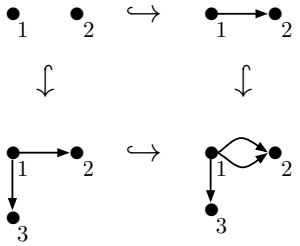


Figure 2.4: A pushout

Pushouts are used as a gluing construction for graphs. The technique allows us to glue graphs together along a common subgraph [EEPT06]. Intuitively, A is the common subgraph; all of the other nodes and edges in B and C are added to this to form the graph D.

Definition 6 (Direct derivation) Given a rule $r\colon \langle L \hookleftarrow K \hookrightarrow R\rangle$, graph G, and an injective graph morphism $g\colon L \hookrightarrow G$ that satisfies the dangling condition, a *direct derivation* $G \Rightarrow_{r,g} H$ from graph G to graph H is given by the double-pushout diagram in Figure 2.5 with pushouts (1) and (2).

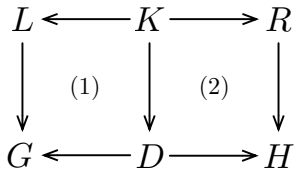


Figure 2.5: A direct derivation

The pushouts (1) and (2) only exist if g satisfies the dangling condition. Since pushouts are unique only up to isomorphism, it follows that direct derivations $G \Rightarrow_{r,g} H$ are also unique only up to isomorphism.

We will often write $G \Rightarrow_r H$ or $G \Rightarrow H$ in place of $G \Rightarrow_{r,g} H$ when no ambiguity arises. We write $G \Rightarrow_{\mathcal{R}} H$ when r is a member of the set $\mathcal{R}$.

Given graphs G and H, and a set of rules $\mathcal{R}$, G *derives* H by $\mathcal{R}$ if $G \cong H$ or there is a sequence of direct derivations

$$G \Rightarrow_{r_1} G' \Rightarrow_{r_2} \ldots \Rightarrow_{r_n} H$$

with $r_1 \ldots r_n \in \mathcal{R}$. We write $G \Rightarrow^*_{\mathcal{R}} H$, or $G \Rightarrow^* H$, denoting that H is derived from G in zero or more direct derivations.

Example 4 Consider the direct derivation in Figure 2.6. The rule being applied comprises the top three graphs; it identifies a pair of adjacent α-labelled nodes, removes one of the nodes and the edge, and replaces them with a looping edge. The graph to which the rule is applied is the one at the bottom left. The graph resulting from this particular rule application is the one at the bottom right.

2.2.3 Double-Pushout Approach with Relabelling

The traditional DPO approach is not ideal for writing rules that relabel nodes, due to the requirement that L, K, and R are totally labelled graphs. While an edge can effectively be relabelled in an arbitrary context by deleting and replacing it with a new label, the same is not true for nodes; the dangling

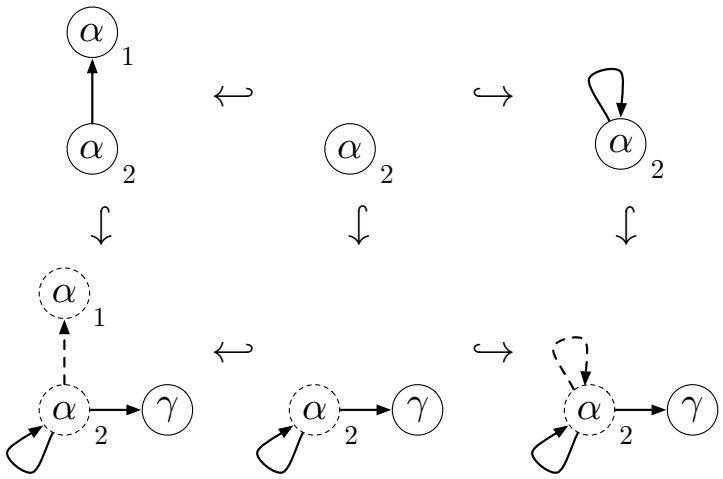


Figure 2.6: A direct derivation (rule application)

condition means that a node can only be deleted if all edges incident to it are also deleted.

Habel and Plump proposed in [HP02] a modification to the traditional approach, specifically to facilitate the relabelling of nodes, while preserving the uniqueness of direct derivations. Their approach relaxes the requirement that the graphs of rules are totally labelled, allowing partially labelled graphs in their place (subject to conditions). Steinert [Ste07] tailored the approach to GP by insisting that L and R are totally labelled, but allowing the nodes of K to be partially labelled (we need not allow partially labelled edges in K since rules can delete and replace edges in an arbitrary context), and disallowing rules that merge nodes together.

Before we define direct derivations with relabelling, it is necessary to review the definition of pullbacks.

Definition 7 (Pullback) Given graph morphisms $B \to D$ and $C \to D$, the *pullback* (1) of these morphisms is formed by the graph A and graph morphisms $A \to B$ and $A \to C$ as in Figure 2.7 if the following properties are satisfied:

Commutativity. $A \to B \to D = A \to C \to D$.

Universal Property. For all graph morphisms $A' \to B$ and $A' \to C$ such that $A' \to B \to D = A' \to C \to D$, there is a unique graph morphism $A' \to A$ such that $A' \to A \to B = A' \to B$ and $A' \to A \to C = A' \to C$.

Definition 8 (Direct derivation (with relabelling)) Given a graph transformation rule $r : \langle L \hookleftarrow K \hookrightarrow R \rangle$ with L and R totally labelled, and an

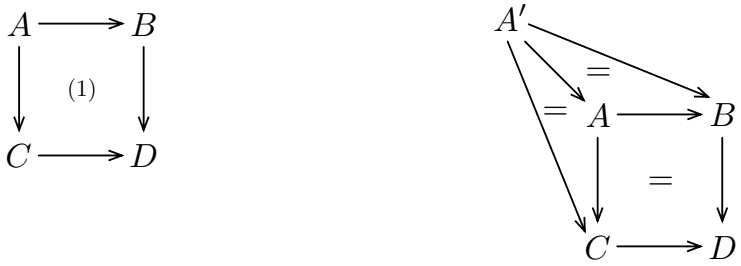


Figure 2.7: A pullback, and the universal property of pullbacks

injective graph morphism $g\colon L \hookrightarrow G$ that satisfies the dangling condition, a *direct derivation (with relabelling)* $G \Rightarrow_{r,g} H$ from graph G to graph H is given in Figure 2.8, where (1) and (2) are "natural" pushouts, that is, both pushouts and pullbacks.

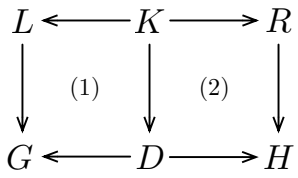
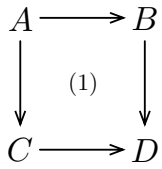


Figure 2.8: A direct derivation with relabelling

The graph H is unique up to isomorphism, and is totally labelled if G is totally labelled [HP02].

[HP02] give a characterisation of natural pushouts. Consider the pushout diagram (1):



(1) is a natural pushout if and only if, for every node v in A that is unlabelled, the image of v in B or C is unlabelled.

Example 5 Consider the pushout diagrams in Figure 2.9. The diagram on the left comprises natural pushouts; the diagram on the right comprises non-natural pushouts (example from [Plu09]).

2.3 Summary

We conclude this chapter on the fundamentals of graph transformation theory, having:

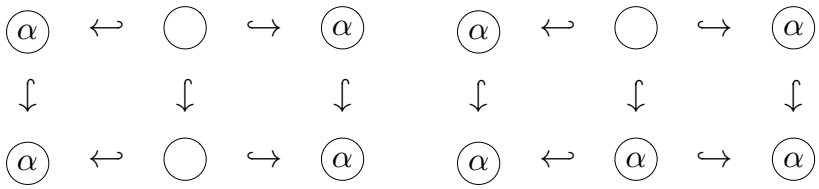


Figure 2.9: Natural (left) and non-natural (right) double-pushout diagrams

1. Defined graphs and the structure preserving graph morphisms that relate them.
2. Reviewed a traditional approach to graph transformation based on the construction of two pushouts, as well as a more recent modification to facilitate the relabelling of nodes (a necessity later, for graph programming).
3. Explored a number of examples in order to make the theory clearer.

Chapter 3

Graph Programs

This chapter is largely based upon parts of [PP10a], as well as the long version of the same paper [PP10b].

This chapter introduces GP (for Graph Programs) [Plu09, PS10], the graph programming language we are interested in developing verification techniques for. Graph programs are formed from two main components: first, a set of (conditional) rule schemata. These are graph transformation rules over expressions and variables (with a condition restricting the possible values of variables and existence of edges); the application of rule schemata instantiates the variables and evaluates expressions, giving a graph transformation rule in the usual sense (infact, rule schemata can represent a possibly infinite number of graph transformation rules). The other component to graph programs is the command sequence, which directs the application of the rule schemata to a graph according to a number of simple control constructs. The syntax of command sequences is very simple, yet GP is computationally complete, in that every computable function on graphs can be programmed [HP01].

The programs mainly considered in this report are rule-based implementations of graph algorithms, for example, computing properties of graphs like colourings. However, there is potential for graph programs to be more widely applicable. For example, in specifying the operational behaviour of real systems, where graphs represent states and rules represent a high-level view of their operations. Verification techniques would facilitate reasoning about the correctness of the specification. Another example might be the simulation of pointers, and proving that certain "shapes" will never result.

GP is not the only programming language based on graph transformation. Others include AGG [Tae04], Fujaba [NNZ00], and GrGen [GBG$^+$06], none of which have a formal semantics. PROGRES [SWZ99] on the other hand does, but it consists of more than 300 rules. GP, in contrast, has a structural operational semantics consisting of only nine inference rules; the designers of the language hope for this to facilitate verification of graph programs and

other types of formal reasoning [Plu09].

We begin by fixing our label alphabets in Section 3.1. Next, we introduce conditional rule schemata in Section 3.2, and the syntax of command sequences in Section 3.3. Then, in Section 3.4, we present a simple graph program (both its rule schemata and command sequence) that finds a colouring for a provided input graph. Finally, we present a structural operational semantics for GP in Section 3.5, and briefly discuss the prototype implementation of the programming language in Section 3.6.

We aim to introduce GP as intuitively as possible and avoid unnecessary technicalities. We do however provide some technical details for when they are later required in the report, for example, the structural operational semantics of GP, which is later used in Chapter 7 for a soundness proof.

3.1 Label Alphabets

GP strictly separates syntax and semantics, and thus makes use of two label alphabets. The graphs of the rule schemata are syntactic graphs, in that labels can contain expressions and variables which are later evaluated. We denote the class of such syntactic graphs by $\mathcal{G}(\text{Exp})$. Rule schemata however are applied to graphs labelled over integers and strings. Let $\mathbb{Z}$ be the set of integers and Char be a finite set of characters. We fix the label alphabet $\mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^+$ of all non-empty sequences over integers and character strings, and denote by $\mathcal{G}(\mathcal{L})$ the set of all graphs over $\mathcal{L}$.

The idea is that graphs from $\mathcal{G}(\text{Exp})$ represent a potentially infinite number of graphs from $\mathcal{G}(\mathcal{L})$, depending on how variables are instantiated and expressions evaluated. We formally describe the syntax of expressions in the section that follows.

3.2 Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of graph programs: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise graph transformation rules in the double-pushout approach with relabelling (see Section 2.2.3 and [HP02]), in that labels can contain expressions over parameters of type integer or string.

The syntax of these expressions is given by the EBNF grammar of Figure 3.1, where VarId is a syntactic class[1] of variable identifiers. We write $\mathcal{G}(\text{Exp})$ for the set of all graphs labelled over the syntactic class Exp.

Figure 3.2 shows a conditional rule schema consisting of the identifier `bridge` followed by the declaration of formal parameters, the left and right

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

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

Figure 3.1: Syntax of expressions

graphs of the schema, the node identifiers `1`, `2`, `3` specifying which nodes are preserved, and the keyword `where` followed by a rule schema condition. Underneath the rule schema is its possible application to a graph; in this case, the instantiations of variables are: $(\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 4, \mathtt{a} \mapsto 4, \mathtt{b} \mapsto 8)$.

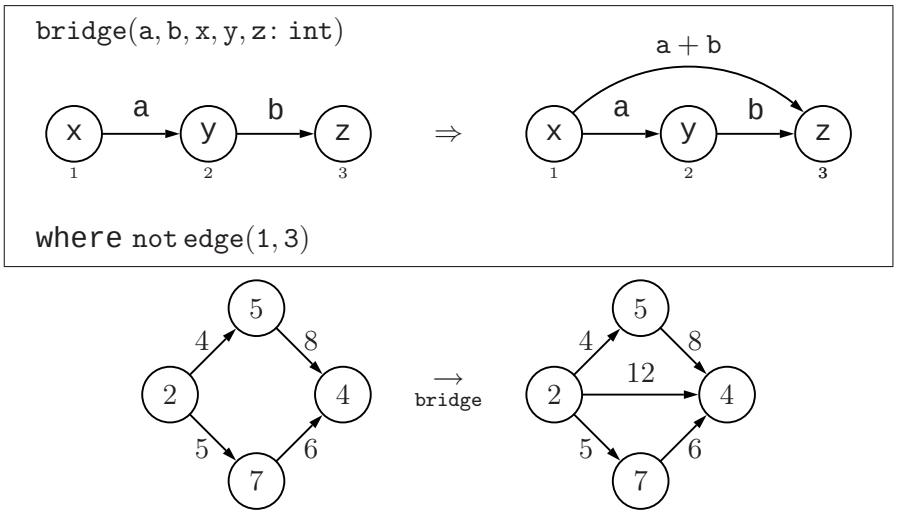


Figure 3.2: A conditional rule schema and a possible application

Labels in the left graph comprise only variables and constants (no composite expressions) because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`, where all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1, 3)` in the condition of Figure 3.2 forbids an edge from node 1 to node 3 when the left graph is matched. The grammar of Figure 3.3 defines the syntax of rule schema conditions, where Term is the syntactic class defined in Figure 3.1.

Conditional rule schemata represent possibly infinite sets of conditional graph transformation rules over graphs in $\mathcal{G}(\mathcal{L})$, and are applied according

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

Figure 3.3: Syntax of rule schema conditions

to the double-pushout approach with relabelling. A rule schema $L \Rightarrow R$ with condition Γ represents conditional rules $\langle\langle L^\alpha \leftarrow K \rightarrow R^\alpha\rangle, \Gamma^{\alpha,g}\rangle$, where K consists of the preserved nodes (which are unlabelled) and $\Gamma^{\alpha,g}$ is a predicate on graph morphisms $g\colon L^\alpha \rightarrow G$ (see [Plu09, PS10]).

3.3 Command Sequences

Rule schemata are applied to graphs according to a number of simple control constructs. These are summarised below:

Rule schema application. Giving the name of a rule schema (e.g. `bridge`) demands the nondeterministic application of it to the graph (with finite failure resulting if it cannot be applied).

Rule set call. Writing $\{r_1, \ldots, r_n\}$ demands the nondeterministic application of a rule schema from the set.

Sequential composition. $P; Q$ demands that program Q is executed after the termination of program P.

If-then-else. A command sequence of the form `if` C `then` P `else` Q demands the execution of program P or Q. The former is executed, if program C terminates on a copy of the current graph; the latter is executed otherwise[2].

As long as possible iteration. $P!$ demands that program P is executed on the graph for as long as it remains applicable to it.

Note that the first two control constructs deal with rule schemata application, and the others arbitrary programs (which may simply be (sets of) rule schemata).

All programs begin with `main =`, and macros can be used to facilitate more readable programs. Every program can be transformed into an equivalent macro-free program by replacing macro calls with their associated

[2]Note that we are testing the applicability of an arbitrary program in the guard, not the truth of some Boolean expression.

command sequences (recursive macros are not allowed). This allows us to always consider programs as command sequences.

3.4 Example Graph Program

Here, we discuss an example program[3] to familiarise the reader with GP's features.

Example 6 A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each non-looping edge have different colours. The program `colouring` in Figure 3.4 produces a colouring for every integer-labelled input graph, recording colours as so-called tags. In general, a tagged label is a sequence of expressions separated by underscores.

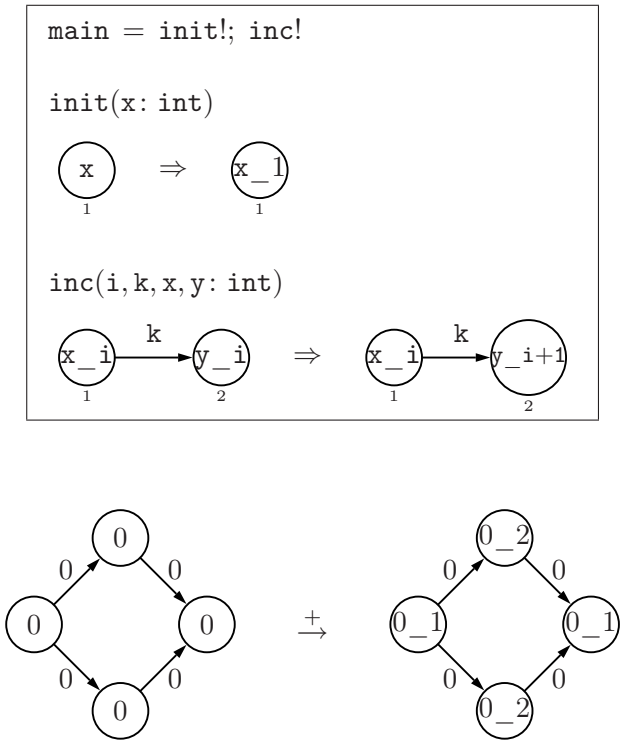


Figure 3.4: The program `colouring` and one of its executions

The program initially colours each node with 1 by applying the rule schema `init` as long as possible, using the iteration operator '`!`'. It then repeatedly increments the target colour of edges with the same colour at both ends. Note that this process is nondeterministic: Figure 3.4 shows an

[3] In Section 7.5, we will come back to this program and show its correctness.

execution producing a colouring with two colours, but a colouring with three colours could have been produced for the same input graph.

It is easy to see that whenever `colouring` terminates, the resulting graph is a correctly coloured version of the input graph. This is because the output graph cannot contain an edge with identically coloured nodes at each end, as then `inc` would have been applied at least one more time. Also, it can be shown that every execution of the program terminates after at most a quadratic number of rule schema applications [Plu09].

3.5 Semantics of Graph Programs

The meaning of graph programs are precisely defined in [PS10, Plu09] by a structural operational semantics (as in [NN07]). The inference rules of the semantics define a small-step transition relation $\rightarrow$ on so-called configurations, which are either a command sequence together with a graph, just a graph, or the special element fail. More formally:

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

where ComSeq denotes the syntactic class of command sequences. A command sequence together with a graph, e.g. $\langle P, G\rangle$, denotes an unfinished computation — in this case, that the command sequence P remains to be executed on graph G. A configuration comprising only a graph G denotes a "successful termination" (i.e. the program terminates and returns a graph); this is in contrast to the configuration fail, which denotes termination with failure (i.e. the program terminates but does not return a graph). Note that there is another possible outcome of executing a graph program: that it does not terminate at all (for example, as long as possible iteration on a rule schema whose left-hand side is the empty graph $\emptyset$).

The inference rules of the semantics are given in Figure 3.5 [PS10]. We let $\mathcal{R}$ range over finite sets of rule schemata, C, P, P', Q over command sequences, and G, H over graphs in $\mathcal{G}(\mathcal{L})$. The *domain* of $\Rightarrow_{\mathcal{R}}$, written $\text{Dom}(\Rightarrow_{\mathcal{R}})$, is the set of all graphs G in $\mathcal{G}(\mathcal{L})$ for which at least one rule schema in $\mathcal{R}$ is applicable, i.e. $\text{Dom}(\Rightarrow_{\mathcal{R}}) = \{G \mid G \Rightarrow_{\mathcal{R}} H \text{ for some } H\}$. The transitive closure of a transition $\rightarrow$ is denoted by $\rightarrow^+$.

In addition to these core commands, GP supports a number of other commands that can be derived from the semantics of the core commands (see Figure 3.6. This includes the special element fail, which has the semantics of an empty rule set call.

Graph programs are summarised by a semantic function, $[\![\_]\!] : \text{ComSeq} \rightarrow (\mathcal{G}(\mathcal{L}) \rightarrow 2^{\mathcal{G}(\mathcal{L}) \cup \{\perp\}})$. For every command sequence P, there is a function $[\![P]\!]$ which maps a graph G in $\mathcal{G}(\mathcal{L})$ to the set of all possible results of executing

$$[\text{Call}_1]_{\text{SOS}}\ \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R}, G\rangle \to H} \qquad [\text{Call}_2]_{\text{SOS}}\ \frac{G \notin \text{Dom}(\Rightarrow_{\mathcal{R}})}{\langle \mathcal{R}, G\rangle \to \text{fail}}$$

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

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

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

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

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

Figure 3.5: Inference rules for core commands [Plu09]

$[\text{Skip}]_{\text{SOS}}$ $\quad \langle \texttt{skip}, G\rangle \to \langle \emptyset \Rightarrow \emptyset, G\rangle$

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

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

Figure 3.6: Inference rules for derived commands [Plu09]

P on G. More formally, where $[\![P]\!]G$ denotes the application of the function to a graph G in $\mathcal{G}(\mathcal{L})$,

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

Intuitively, a program can diverge if there is an infinite sequence of transitions; it can get stuck if it can reach a configuration in which a command sequence remains to be executed, yet there are no further transitions that can be made for that command sequence and graph.

The semantic function plays an important part in the soundness proof of the Hoare calculus presented in Chapter 7.

3.6 Prototype Implementation

A prototype implementation of GP by Manning [MP08b, MP08a] is available to local users at York[4]. It consists of a graphical editor, a compiler, and the *York Abstract Machine* (YAM). The interaction of these components is summarised in Figure 3.7, where GXL is the *Graph eXchange Language*, and YAMG is an internal graph format of the abstract machine.

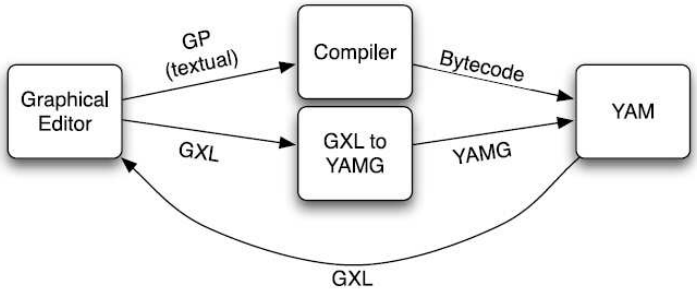


Figure 3.7: Interaction of the components implementing GP [Plu09]

The graphical editor for GP is a Java application which supports the creation of graphs and graph programs. Figure 3.8 is a screenshot of the program editing environment.

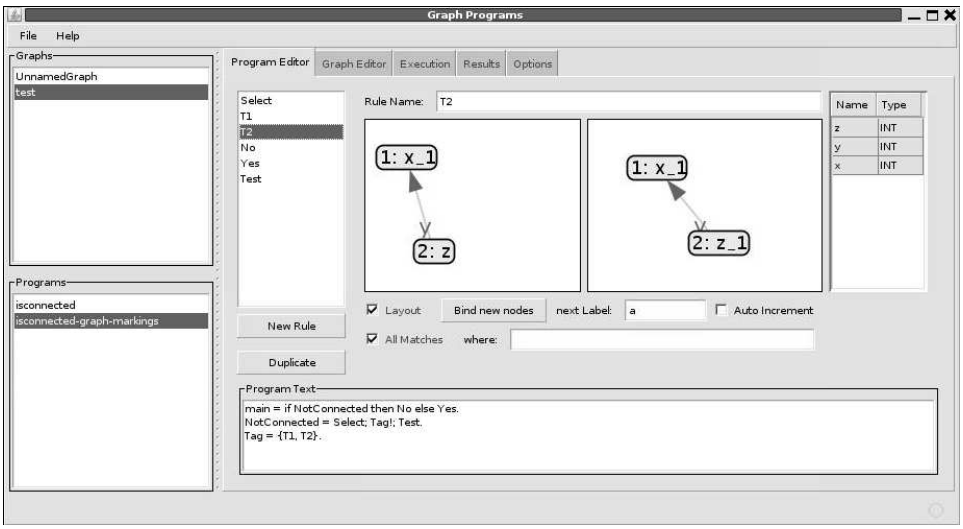


Figure 3.8: The GP graphical editor

The YAM is written in C. It executes low-level operations on graphs for graph matching and transformation, faithfully matching the semantics of the

[4]A guide to the system and a tutorial are provided in [Pos10].

language. It takes as input YAM bytecode and graph files, and returns sets of graphs to the graphical editor. The YAM is stack based, allows backtracking, and maintains a current graph.

To implement the nondeterminism of graph programs, the YAM provides a backtracking mechanism which can be enabled or disabled according to a setting in the graphical editor. Backtracking exists so that when a path of computation fails, other paths are then attempted (similarly to Prolog).

The YAM bytecode is generated by the *GP to YAM bytecode compiler*, written in Haskell. It generates search plans for rules, the code implementing the effects of rules, and the code for backtracking.

3.7 Summary

We conclude this chapter on graph programs, having:

1. Separated syntax and semantics by defining two classes of graphs, $\mathcal{G}(\text{Exp})$ and $\mathcal{G}(\mathcal{L})$ respectively.

2. Introduced the conditional rule schemata of GP, which generalise graph transformation rules in the double-pushout approach with relabelling, with expressions over labels, and conditions restricting applicability.

3. Informally introduced the syntax of command sequences (graph programs).

4. Defined the meaning of graph programs with a structural operational semantics.

5. Discussed the prototype implementation of GP.

Chapter 4

Proving the Correctness of Imperative Programs

In this chapter, we consider a number of verification techniques for imperative programming languages, and discuss whether they could be successfully adapted to the domain of graphs for verifying programming languages like GP (see Section 3). To accomplish this for any verification technique is non-trivial. Traditional programs are deterministic (there is at most one instruction to be executed "next"); graph programs are nondeterministic. Traditional programs have as their states the values of their variables; the states of graph programs are simply graphs.

The chapter begins with a review of Hoare logic in Section 4.1 (we pay particular attention to this, since we are later able to present a Hoare calculus for verifying graph programs). Next, in Section 4.2, we discuss predicate transformers (e.g. weakest precondition) and their role in program verification. This, again, we give special attention, since a weakest preconditions approach to verifying graph programs has been proposed by Habel, Pennemann, and Rensink (see Section 6.3). Finally, we consider the verification of programs by model checking in Section 4.3.

4.1 Hoare Logic

The idea of Hoare logic[1] is to provide a formal system for reasoning about program correctness in a syntax-directed manner [AdO09]. Seeded by Floyd's similar work for flowcharts [Flo67], Hoare proposed the approach for program text in his seminal paper [Hoa69]. Numerous researchers in the decades since have seen the ideas developed.

The essence of Hoare logic is the *Hoare triple*, $\{p\}\ S\ \{q\}$, where S is some program text, and p and q are *assertions* (predicates) representing the

[1] Also known as Floyd-Hoare logic.

preconditions and postconditions, respectively, of the triple. If $\{p\}\ S\ \{q\}$ is true in the sense of *partial correctness*, written $\models \{p\}\ S\ \{q\}$, then if the program S is executed in a state satisfying predicate p, and that execution terminates (the termination of S is not guaranteed), it will do so in a state satisfying q. If $\{p\}\ S\ \{q\}$ is true in the sense of *total correctness*, written $\models_{tot} \{p\}\ S\ \{q\}$, then if the program S is executed in a state satisfying p, the program S is guaranteed to terminate in a state satisfying q. Note that total correctness implies partial correctness.

4.1.1 Axiom Schemata and Inference Rules

For the control constructs of an imperative programming language, Hoare logic provides axiom schemata and inference rules, allowing us to prove correctness of programs by induction on the program syntax [AdO09]. We write $\vdash \{p\}\ S\ \{q\}$ if $\{p\}\ S\ \{q\}$ can be proven in a system of axiom schemata and inference rules.

Axiom schemata[2] represent a possibly infinite number of axioms; they range over arbitrary predicates and program text, which can be instantiated to derive axioms for proofs. We write $\vdash \{p\}\ S\ \{q\}$ if $\{p\}\ S\ \{q\}$ is an instance of an axiom schema. For example, consider the axiom schemata for the "skip" and "assignment" control constructs (with the usual semantics):

$$[\text{skip}]\ \frac{}{\{p\}\ \texttt{skip}\ \{p\}} \qquad\qquad [\text{ass}]\ \frac{}{\{p[x \mapsto a]\}\ x := a\ \{p\}}$$

The [skip] axiom schema is obviously plausible: if `skip` is executed in a program state satisfying predicate p, then after the execution of `skip` (which, by definition, does not alter the program state), the resulting program state will clearly still satisfy p. The [ass] axiom schema encourages backwards reasoning. It states that the truth of p after the assignment operation is equal to the truth of p before the assignment but with every x replaced by a (we write $p[x \mapsto a]$ to denote such a substitution)[3].

[ass] is better understood with an example. Consider the program fragment `y := x + 3`, and the postcondition $y > 5$. By substitution, we get the precondition $x+3 > 5$ and thus the Hoare triple $\{x+3 > 5\}$ `y := x + 3` $\{y > 5\}$. Because this triple can be derived from the instantiation of an axiom schema, we write $\vdash \{x + 3 > 5\}$ `y := x + 3` $\{y > 5\}$. We can also write $\models_{tot} \{x+3 > 5\}$ `y := x + 3` $\{y > 5\}$, since it is true in the sense of total correctness (this is because the [ass] axiom schema is sound for total correctness — see Section 4.1.2).

[2]For brevity, the literature often refers to axiom schemata as *axioms* [AdO09, NN07].

[3]For simplicity, we have been sloppy in our separation of syntax (the program text) and semantics. A better substitution would be $p[x \mapsto \mathcal{A}[\![a]\!]]$, where $\mathcal{A}$ denotes a semantic function for simple arithmetical expressions.

Inference rules facilitate the syntactic deduction of Hoare triples from one or more others. In a standard proof system, an inference rule comprises at least one premise, and a conclusion. In a considered interpretation, if the premises are found to be true (and the inference rule is sound), then the conclusion is also true [Gal03]. The idea of the inference rules in a Hoare logic system is the same. For example, consider the sequential composition and if-then-else control constructs (with the usual semantics):

$$[\text{comp}]\ \frac{\{p\}\ S_1\ \{r\} \qquad \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$$

$$[\text{if}]\ \frac{\{p \wedge B\}\ S_1\ \{q\} \qquad \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{q\}}$$

In both [comp] and [if], the triples above the line are the premises, and the triple below the line is the conclusion. We write $\vdash \{p\}\ S\ \{q\}$ if $\{p\}\ S\ \{q\}$ is the conclusion of an inference rule for which all the premises can be obtained. The conclusion of an inference rule, $\{p\}\ S\ \{q\}$, is true in the sense of partial (total) correctness, written $\models \{p\}\ S\ \{q\}$ ($\models_{tot} \{p\}\ S\ \{q\}$), if all of the premises of the rule are true in the sense of partial (total) correctness.

The inference rule for [comp], where S_1, S_2 are arbitrary statements and ";" denotes sequential composition, is fairly intuitive, requiring one to find a suitable intermediate assertion r. The inference rule for [if] formalises two cases in its premises, one for each of the two possible Boolean interpretations of B [AdO09].

Another standard example of an inference rule is the one for a while loop, which allows us to reason about loop invariants (predicates that hold before and after any number of executions of a program). A consequence rule is often used in proofs, which allows us to strengthen preconditions and weaken postconditions (or replace an assertion with an equivalent one); it can be applied regardless of the program text, existing explicitly to allow us to manipulate the logical formulae that form the assertions. See [AdO09] for more information.

4.1.2 Soundness and Completeness

Soundness and completeness are very desirable properties of any proof system. We introduce the properties informally before giving more formal definitions.

An axiom schema is said to be *sound* for partial (total) correctness if it only derives axioms that are true in the sense of partial (total) correctness

according to the semantics of the program text. An inference rule is said to be sound for partial (total) correctness if the truth of its premises implies the truth of its conclusion in the sense of partial (total) correctness. A proof system is said to be sound if all the axiom schemata and inference rules that comprise it are sound. More formally, for partial correctness,

$$\vdash \{p\}\ S\ \{q\} \text{ implies } \models \{p\}\ S\ \{q\}$$

and for total correctness,

$$\vdash \{p\}\ S\ \{q\} \text{ implies } \models_{tot} \{p\}\ S\ \{q\}$$

A proof system is said to be *complete* for partial (total) correctness if every Hoare triple that is true in the sense of partial (total) correctness can be deduced from the axiom schemata and inference rules of the proof system. More formally, for partial correctness,

$$\models \{p\}\ S\ \{q\} \text{ implies } \vdash \{p\}\ S\ \{q\}$$

and for total correctness,

$$\models_{tot} \{p\}\ S\ \{q\} \text{ implies } \vdash \{p\}\ S\ \{q\}$$

4.1.3 Proof Trees

One method of showing $\vdash \{p\}\ S\ \{q\}$ for a Hoare triple $\{p\}\ S\ \{q\}$ is to construct a proof tree[4] using the axiom schemata and inference rules of the proof system [NN07]. The root of the tree is the Hoare triple we are attempting to prove, and the leaves are instances of axiom schemata. The rest of the tree is constructed by applying inference rules.

Once we have constructed the proof tree and shown $\vdash \{p\}\ S\ \{q\}$, if the axiom schemata and inference rules used are known to be sound for partial (total) correctness, then we also have $\models \{p\}\ S\ \{q\}$ (or $\models_{tot} \{p\}\ S\ \{q\}$).

Example 7 Consider the program $\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z}$, which swaps the values of variables $\mathtt{x}$ and $\mathtt{y}$ via a temporary variable $\mathtt{z}$. We show that $\vdash \{y = 3 \wedge x = 5\}\ \mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z}\ \{x = 3 \wedge y = 5\}$[5] by the proof tree in Figure 4.1.

[4] Unlike other trees in computer science, these proof trees are the "right way round". That is, the root appears at the bottom, and the leaves appear at the top!

[5] We could prove a more general property, that whatever the initial values of x and y, the program swaps them. This could be done using logical variables in place of 3 and 5.

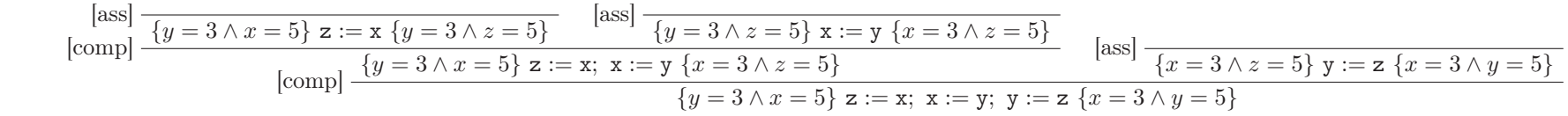


Figure 4.1: A proof tree

4.1.4 Adaptation to Graph Programs

We show in Chapter 7 that Hoare logic can be adapted to the domain of graphs. Rather than use predicates in the assertions, we use a graph specification formalism (see Chapter 5) and reason about the changing properties of the graph on which the program is executing. Many of GP's control constructs fit naturally with what we have shown, i.e. sequential composition of graph programs and the inference rule [comp]. The main challenge is defining an axiom schema for the application of a rule schema.

4.2 Predicate Transformers

Predicate transformers are, in general, total functions that map a predicate and some program text to a predicate. They were introduced by Dijkstra in 1975, who defined the wp (for *weakest precondition*) transformer for his Guarded Command Language [Dij75, Dij76]. Intuitively, wp takes as arguments a piece of program text S and a postcondition q; it returns the weakest possible precondition necessary for the execution of the program S to terminate in a state satisfying q [NN07]. More formally, we have $\models_{tot} \{\text{wp}(S, q)\}\ S\ \{q\}$ and:

$$\text{if } \models_{tot} \{p\}\ S\ \{q\} \text{ then } p \Longrightarrow \text{wp}(S, q)$$

Because of this fact, the problem of verifying a Hoare triple reduces to the validity problem for $p \Longrightarrow \text{wp}(S, q)$. wp is not the only predicate transformer. Another common example is wlp (for *weakest liberal precondition*), which corresponds to partial correctness in Hoare logic. Intuitively, it generates the weakest possible precondition such that a program either finishes execution in a state satisfying a given postcondition, or does not finish execution at all. More formally, we have $\models \{\text{wlp}(S, q)\}\ S\ \{q\}$ and:

$$\text{if } \models \{p\}\ S\ \{q\} \text{ then } p \Longrightarrow \text{wlp}(S, q)$$

Another common predicate transformer is sp (for *strongest postcondition*). We have $\models_{tot} \{p\}\ S\ \{\text{sp}(S, p)\}$ and:

$$\text{if } \models \{p\}\ S\ \{q\} \text{ then } \text{sp}(S, p) \Longrightarrow q$$

In other words, we can verify a Hoare triple by applying sp to the program text and precondition, and checking whether the resulting predicate implies the postcondition.

4.2.1 Defining a Predicate Transformer Semantics

For a particular predicate transformer, it is necessary to define transformations for each of the control constructs in the programming language con-

sidered, in order to allow the predicate transformer to take an arbitrary program as input. The transformations are entirely symbolic, and need to be proven sound according to the semantics of the programming language.

To illustrate, we define a wp-based semantics for the skip, assignment, sequential composition, and if-then-else control constructs of Section 4.1.

$$\begin{aligned}
\mathrm{wp}(\texttt{skip}, q) &= q \\
\mathrm{wp}(x := a, q) &= q[x \mapsto a] \\
\mathrm{wp}(S_1;\ S_2, q) &= \mathrm{wp}(S_1, \mathrm{wp}(S_2, q)) \\
\mathrm{wp}(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, q) &= (B \wedge \mathrm{wp}(S_1, q)) \vee (\neg B \wedge \mathrm{wp}(S_2, q))
\end{aligned}$$

Example 8 Consider again the program $\texttt{z} := \texttt{x};\ \texttt{x} := \texttt{y};\ \texttt{y} := \texttt{z}$ from Example 7, and the requirement to show $\vdash \{y = 3 \wedge x = 5\}\ \texttt{z} := \texttt{x};\ \texttt{x} := \texttt{y};\ \texttt{y} := \texttt{z}\ \{x = 3 \wedge y = 5\}$. This time we prove the Hoare triple using our wp-based semantics. Let $S = \texttt{z} := \texttt{x};\ \texttt{x} := \texttt{y};\ \texttt{y} := \texttt{z}$, $p = y = 3 \wedge x = 5$, and $q = x = 3 \wedge y = 5$. Then:

$$\begin{aligned}
\mathrm{wp}(S, q) &= \mathrm{wp}(\texttt{z} := \texttt{x}, \mathrm{wp}(\texttt{x} := \texttt{y};\ \texttt{y} := \texttt{z}, q)) \\
&= \mathrm{wp}(\texttt{z} := \texttt{x}, \mathrm{wp}(\texttt{x} := \texttt{y}, \mathrm{wp}(\texttt{y} := \texttt{z}, q))) \\
&= \mathrm{wp}(\texttt{z} := \texttt{x}, \mathrm{wp}(\texttt{x} := \texttt{y}, q[y \mapsto z])) \\
&= \mathrm{wp}(\texttt{z} := \texttt{x}, q[y \mapsto z][x \mapsto y]) \\
&= q[y \mapsto z][x \mapsto y][z \mapsto x] \\
&= (x = 3 \wedge y = 5)[y \mapsto z][x \mapsto y][z \mapsto x] \\
&= (x = 3 \wedge z = 5)[x \mapsto y][z \mapsto x] \\
&= (y = 3 \wedge z = 5)[z \mapsto x] \\
&= (y = 3 \wedge x = 5)
\end{aligned}$$

Trivially, we have that $p \Longrightarrow \mathrm{wp}(S, q)$ since $p = \mathrm{wp}(S, q)$, and thus we have $\vdash \{y = 3 \wedge x = 5\}\ \texttt{z} := \texttt{x};\ \texttt{x} := \texttt{y};\ \texttt{y} := \texttt{z}\ \{x = 3 \wedge y = 5\}$. This example was chosen for simplicity; of course, it is often the case that it is challenging to show the implication to be valid.

4.2.2 Deciding the Validity of Formulae

Using predicate transformers reduces the task of showing $\vdash \{p\}\ S\ \{q\}$ to the validity problem for a logical implication, for example, $p \Longrightarrow \mathrm{wp}(S, q)$. Such a formula is said to be *valid* (or a *tautology*) if it is true in all models; the *validity problem* is the problem of deciding whether a given formula is valid

or not. Church's theorem states that if we are given an arbitrary formula of first-order logic, the validity problem is undecidable [Gal03]. However, it is a *semidecidable* problem; that is, there are search algorithms which can confirm whether a valid formula is valid, but may run forever on invalid formulae. We begin with a discussion of two of the search algorithms more suited to automatically proving the validity of a formula — semantic tableaux and resolution — before briefly discussing a selection of implemented theorem provers.

Semantic Tableaux and Resolution

Semantic tableaux and resolution are *refutation methods*; that is, rather than prove the validity of a formula r, the idea is to prove that $\neg r$ cannot be satisfied. We introduce both of these refutation methods, but refer the reader to [Fit96] for the full details.

First, we consider the *semantic tableaux* method, invented by Evert Willem Beth. The idea of this method is to break down a complex formula into its constituent parts, and search among them for complementary formulae, that is, r and $\neg r$. The method involves the construction of a tree. We begin by negating the formula we wish to show valid, and place it at the root of the tree. The tree is then grown by the repeated application of rules, some of which introduce branching (for example, the "or" and "implication" rules). The formula at the root is then proven to be unsatisfiable (and thus the original formula valid) if every branch of the tree is *closed*; that is, every branch contains complementary formulae, some r and $\neg r$.

For formulae written in propositional logic, we can construct a tableau from three rules. Whenever a branch of a tableau:

1. Contains the formula $A \wedge B$, connect to its leaf a new child labelled A, and connect to that a new child labelled B.

2. Contains the formula $A \vee B$, connect to its leaf two new children, one labelled A, and one labelled B.

3. Contains the formula $\neg\neg A$, connect to its leaf a new child labelled A.

For formulae written in first-order logic, two further rules are required to handle existential and universal quantifiers. These rules are not quite so trivial, and employ a form of Skolemization; we refer the reader to [Fit96] for further details.

The *resolution* method was introduced in 1965 by John Alan Robinson [Rob65]. For propositional logic, the resolution method rests on this equivalence: [Gal03]

$$(A \vee P) \wedge (B \vee \neg P) \equiv (A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)$$

The clause $(A \vee B)$ is said to be the *resolvent* of $(A \vee P)$ and $(B \vee \neg P)$. This process of adding a resolvent is captured by a single, sound inference rule:

$$[\text{resolution}]\ \frac{(\alpha_1 \vee P) \wedge (\alpha_2 \vee \neg P)}{(\alpha_1 \vee \alpha_2)}$$

where α_1, α_2 are disjunctions of proposition symbols. Resolution is again a refutation method, so to prove the validity of a formula, we prove the negation of it to be unsatisfiable. Using this inference rule, a formula is proven to be unsatisfiable if we can obtain the empty clause $\square$, which is the resolvent of the clauses P and $\neg P$. Quite remarkably, the resolution method is complete despite comprising only one rule — however, it requires formulae to first be transformed into conjunctive normal form.

For first-order logic, Robinson [Rob65] showed that again only one inference rule was needed for the method to be complete. Formulae are required to be in conjunctive normal form; each clause is in prenex normal form (a string of quantifiers followed by a quantifier-free part), and existential quantifiers are removed by Skolemization. Quantifiers are usually omitted in resolution proofs, with the understanding that every variable in a formula is universally quantified. With a formula in this form, the resolution inference rule involves identifying pairs of clauses containing a predicate in one and its negation in the other, and performing unification on the predicates. The unified predicates are discarded, and a new clause is formed from the remaining predicates of the two clauses.

The resolution method has become popular in automated first-order theorem provers (see below), since its sole inference rule makes it easy to implement [Gal03]. This is despite the requirement for formulae to be transformed into conjunctive, prenex, and Skolem normal form.

Theorem Provers

By Church's theorem, there is no hope of purely mechanical refutation of first-order formulae; there is no algorithm for deciding the unsatisfiability (validity) of a formula. On the other hand, the problem is semidecidable for classical first-order logic. If a first-order formula is valid (or unsatisfiable), then we can prove it by implementing complete proof systems like those we have discussed. But if our formula is invalid, since the problem is semidecidable, the algorithms we implement are not guaranteed to terminate with a "no" answer.

Despite such theoretical results, in practice, *theorem provers* — computer programs that prove mathematical theorems — are now able to solve many difficult problems [Gal03]. Popular first-order theorem provers include

Vampire [RV02] and Prover9[6] [McC09], both implementing variations of Robinson's resolution method.

For higher-order logics which are not even semidecidable, interactive theorem provers like Coq and Isabelle are often used. These involve a man-machine collaboration, with humans guiding the proof assistants to find solutions.

4.2.3 Adaptation to Graph Programs

A weakest preconditions approach for verifying graph programs was successfully investigated by Habel, Pennemann, and Rensink (see Section 6.3). They describe preconditions and postconditions of graph programs using the nested conditions specification formalism (see Chapter 5), and define a weakest preconditions transformation over each of the control constructs in their variation of graph programs. As there is a transformation from nested conditions to first-order logic on graphs, they were able to automate some validity proofs by feeding the generated sentences of logic into off-the-shelf first-order theorem provers. However, because of the restriction to finite graphs (i.e. finite models), the validity problem is no longer even semidecidable (by Trakhtenbrot's theorem from finite model theory [Tra50, BEE$^+$07]); that is, we can search for proofs using methods like those we have discussed, but for finite graphs, we are not guaranteed to find solutions. As part of Pennemann's research, a resolution-like theorem prover tailored to proving the validity of nested conditions was also developed. This was demonstrated to perform well, despite the fact that the problem is not even semidecidable.

4.3 Model Checking

Model checking [EMCP99, Cla08] as a verification technique has been studied for over 25 years. It allows us to show the correctness of programs in a rather different way to the likes of Hoare logic and predicate transformers. Rather than tasking ourselves (or a theorem prover) with building a correctness proof for some program together with pre- and postconditions, model checking involves an algorithmic traversal of the possible system states we might reach from an execution, and checking that some safety properties hold in all states.

Models of systems are transition systems or annotated graphs known as Kripke structures (a type of nondeterministic finite state machine), where nodes represent the reachable states of the system and the edges represent the state transitions. Safety properties are usually expressed in *temporal logics*, such as *linear temporal logic (LTL)*. These allow one to express properties quantified in terms of time without introducing time explicitly (i.e.

[6] Prover9 is the successor to Otter [McC03].

express properties about the future of execution paths) [Cla08]. For a proposition p expressing some property a state can exhibit, one can write in LTL, for example, formulae like $\Box p$ (property p always holds) or $\Diamond p$ (property p eventually holds). A model checking tool is given a model and formulae of temporal logic (as in Figure 4.2), and exhaustively searches the state space. If a state is found which does not satisfy the safety properties (a counterexample), it is reported by the model checker. SPIN is a well-known example of a model checker; it expects models to be described in the Promela modelling language, and safety properties to be expressed by linear temporal logic (LTL) formulae.

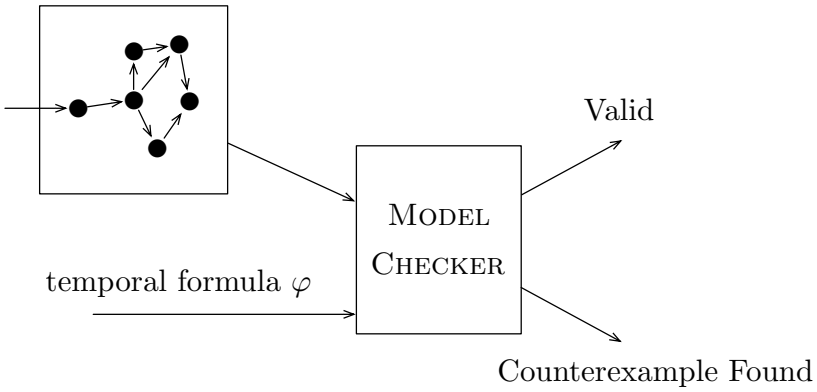


Figure 4.2: Model checking process

Model checking has a number of advantages, but also a number of challenging disadvantages to overcome [Cla08]. On the positive side, once a counterexample is found, the model checker can provide an execution trace by looking at the provided model — invaluable for debugging. Additionally, it allows us to avoid the need to construct proofs; the checking process is automatic. On the other hand, model checkers face challenges like the state explosion problem (a combinatorial blow up of the state space), and the fact that Kripke structures for many real systems are infinite (indeed, this would be the case for GP).

4.3.1 Adaptation to Graph Programs

To model check a system, we need to be able to represent its states as the nodes of a graph, and its state transitions as the graph's edges. For a graph program, the program states are simply graphs, and the transitions between states (i.e. graphs) are applications of rule schemata. This implies that the nodes of a model should encode graphs, and that the model should have transitions for all possible rule applications (and all possible matches). Safety properties would need to express (un)desired graph properties, perhaps by using logics on graphs or graph specification formalisms (see Chapter 5).

Some work has already been undertaken on the topic of model checking graph transformation systems. See the discussion in Section 6.1 for further details.

4.4 Summary

We conclude this chapter on proving the correctness of imperative programming languages, having:

1. Reviewed Hoare logic, which in Chapter 7 we adapt to the domain of graphs.
2. Reviewed predicate transformers, which play a major role in some of the most significant existing research on proving the correctness of graph programs (see Section 6.3).
3. Discussed model checking for program verification. Explained how one might model a graph program, and suggested the need to have a graph specification formalism for expressing safety properties.

By no means have we exhaustively surveyed verification techniques for imperative programs. Rather, we have focused on techniques that look promising for graph programs (see Chapters 6 and 7).

Chapter 5

Specification Formalisms and Logics for Graphs

When executing a graph program, the program states are simply graphs. This implies that, to be able to adapt classical program verification techniques to the domain of graphs, we need to be able to express things about graphs. Specifically, we want to be able to express properties of graphs, ranging from general graph theoretic properties like "the graph is connected" and "the graph is cycle-free", to properties specific to a problem domain, such as "every node labelled `student` is adjacent to at least one node labelled `module`".

This chapter investigates three particular specification formalisms for graphs: nested conditions in Section 5.1, hyperedge replacement conditions in Section 5.3, and graph reduction specifications in Section 5.4. We relate the expressive power and decidability properties of these formalisms to logics on graphs, one of which — monadic second-order logic — is given particular attention (Section 5.2), since it is able to express non-local graph properties, and has been extensively researched [Cou90, Cou97, Cou10].

We give a full, formal definition of nested conditions, since the formalism is later extended in Chapter 7; for the others, we choose to omit some of the technicalities, and rather introduce them intuitively and by examples.

5.1 Nested Conditions

Nested conditions are a simple, intuitive, but mathematically precise specification formalism for graphs and other high-level structures. The expressive power of nested conditions for graphs is equivalent to that of first-order logic for graphs, yet from our experience, nested conditions are generally more concise and readable than their equivalent sentences in first-order logic.

The formalism has developed over a number of years, from the graph constraints and application conditions for rules of [HW95], to the more general

definition we consider from [HP09, Pen09]. Important contributions were made along the way. Rensink, for example, generalised graph constraints and application conditions to nested conditions for edge-labelled graphs in [Ren04]. Habel and Pennemann then lifted this work to so-called high-level structures[1].

For brevity, we will often refer to nested conditions as simply *conditions*.

5.1.1 Definition and Examples

Here, we formally define nested conditions for graphs (following [HP09]) and the satisfaction of nested conditions, before discussing a number of simple examples.

Definition 9 (Nested condition) A *(nested) condition* c over a graph P is of the form true or $\exists(a,\, c')$, where $a\colon P \to C$ is a graph morphism, and c' is a condition over C. Moreover, Boolean formulae over conditions over P yield conditions over P, that is, $\neg c$ and $c_1 \wedge c_2$ are conditions over P, where c_1 and c_2 are conditions over P.

For brevity, we write false for $\neg$true, $\exists(a)$ for $\exists(a, \text{true})$, and $\forall(a, c')$ for $\neg\exists(a, \neg c')$. In examples, when the domain of morphism $a\colon P \to C$ can unambiguously be inferred, we write only the codomain C. For instance, a condition $\exists(\emptyset \to C, \exists(C \to C'))$ 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 condition's morphism. A condition over a graph morphism whose domain is the empty graph is referred to as a *graph constraint*. Conditions over the left- and right-hand sides of graph transformation rules are referred to as *application conditions*.

Definition 10 (Satisfaction of conditions) The *satisfaction* of conditions by graph morphisms is defined inductively. Every morphism satisfies the condition true. A graph morphism $p\colon P \hookrightarrow G$ satisfies the condition $c = \exists(a : P \to C, c')$, denoted $p \models c$, if there exists an injective graph morphism $q\colon C \hookrightarrow G$ with $q \circ a = p$, and $q \models c'$.

$$\begin{array}{ccc} \exists(P & \xrightarrow{a} & C,\ c') \\ & {\scriptstyle p}\searrow \quad \swarrow{\scriptstyle q} & \\ & G & \end{array}$$

A graph G satisfies a condition c, denoted $G \models c$, iff the morphism $\emptyset \to G$ satisfies c.

[1] For simplicity, we consider no structure other than graphs.

A simple example of a condition is $c = \exists(\textcircled{2} \rightarrow \textcircled{2})$, which is read "there exists adjacent nodes labelled with 2, such that the mutually incident edge is unlabelled". Without abbreviations, we have $c = \exists(\emptyset \rightarrow \textcircled{2} \rightarrow \textcircled{2}, \text{true})$. A graph G would satisfy this condition, denoted $G \models c$, if there is a subgraph in G isomorphic to the codomain of the morphism.

An example of a condition with Boolean expressions is $d = \exists(\textcircled{id}) \wedge \neg\exists(\textcircled{id})$, which is read "there exists a node labelled `id` incident to exactly one looping edge". Without abbreviations, we have $d = \exists(\emptyset \rightarrow \textcircled{id}, \text{true}) \wedge \neg\exists(\emptyset \rightarrow \textcircled{id}, \text{true})$.

An example of a condition with nesting is:

$$e = \forall(\textcircled{child}_1, \exists(\textcircled{child}_1 \rightarrow \textcircled{parent}))$$

The condition e is read "every node labelled `child` is adjacent to a node labelled `parent`". The small 1 is used to identify nodes as being the same. Informally, the outermost part of the condition can be viewed as identifying nodes labelled `child`, and the innermost part can be viewed as describing the required context of these nodes. Without abbreviations, we have:

$$e = \forall(\emptyset \rightarrow \textcircled{child}_1, \exists(\textcircled{child}_1 \rightarrow \textcircled{child}_1 \rightarrow \textcircled{parent}, \text{true}))$$

5.1.2 Expressiveness and Decidability

Graph conditions are equivalently expressive to the first-order graph formulae (first-order logic on graphs) of Courcelle [Cou90]. Habel and Pennemann prove this in [HP09], by showing that graph conditions can be transformed into equivalent first-order graph formulae and vice versa. The result means that graph conditions inherit various properties of first-order graph formulae related to expressiveness and decidability.

Expressiveness

It might be surprising that there are many basic graph properties that first-order graph formulae (and thus, graph conditions) cannot express. For example, it is not possible to finitely describe the existence of an arbitrary-length path between nodes. This instantly rules out the possibility of describing properties like "the graph is connected", "the graph is a tree", or indeed more complex properties like "the graph is Hamiltonian". Specifically, graph conditions can only describe *local* graph properties [Cou90].

Despite this weakness, Habel and Pennemann have shown graph conditions to be adequate in a number of case studies where graph transformations are used to model the specifications of systems, including that of a railroad and access control system [HP05, HP09, Pen09]. This is because in these particular case studies, local properties like "no user is assigned more than one session" might be of the most interest.

Graph conditions are not ideal for graph transformation systems over an infinite label alphabet (such as the set of natural numbers, $\mathbb{N}$). Suppose that we are considering graphs labelled over $\mathbb{N}$. Since the graphs in conditions are totally labelled, to express the property that "the graph contains a node" (i.e. is non-empty), we would require an infinite disjunction of conditions:

$$\exists(⓪) \vee \exists(①) \vee \exists(②) \vee \dots$$

Decidability

We are interested in three particular decision problems for graph conditions: the *model checking problem*, the *satisfiability problem*, and the *validity problem*. These are briefly discussed below. The (un-)decidability of the problems follow from results for first-order logic and finite model theory (see [Lib04]), since every nested condition can be transformed into a sentence of first-order logic and since the models we are considering are finite graphs.

Model checking problem. *Instance.* A graph G and a condition c. *Question.* Does $G \models c$? *Decidability.* The model checking problem for graph conditions is decidable.

Satisfiability problem. *Instance.* A condition c. *Question.* Does there exist a graph G that satisfies c? *Decidability.* The satisfiability problem for graph conditions is undecidable (but semidecidable).

Validity problem. *Instance.* A condition c. *Question.* Does every graph G satisfy c? *Decidability.* The validity problem for graph conditions is undecidable (not even semidecidable).

While for sentences of classical first-order logic the validity problem is semidecidable, the same is not true when considering finite models (such as finite graphs); in these cases, the validity problem is not even semidecidable (by Trakhtenbrot's theorem [Tra50, BEE$^+$07]). While we can search for solutions, we are not guaranteed to find them.

5.2 Monadic Second-Order Logic

Since first-order logic (and equivalently expressive formalisms like graph conditions) is insufficiently powerful to express a number of simple graph properties, such as the existence of arbitrary length paths, we briefly consider a more powerful logic — *monadic second-order logic*.

Monadic second-order logic extends first-order logic by allowing one to quantify unary relations (subsets of some set), rather than simply the individual elements of sets. In the sense of graphs, the extension allows one to quantify over sets of nodes and edges. As the name suggests, the logic we

consider is a fragment of full second-order logic, which lifts the restriction that relations be unary (that is, one can quantify over relations of arbitrary arity).

Courcelle has extensively studied monadic second-order logic for expressing graph properties [Cou90, Cou97, Cou10]. It is his notation we adopt for our examples.

For brevity, we shall write MSOL in place of monadic second-order logic throughout the rest of this section.

5.2.1 Definition and Examples

Courcelle represents graphs by relational structures[2] in order to express graph properties by logical formulae [Cou97]. Following Courcelle, we associate with a directed graph G the relational structure $\lfloor G \rfloor = \langle V_G, edg_G \rangle$ where V_G is the set of nodes of G and $edg_G \subseteq V_G \times V_G$ is a binary relation such that $(x, y) \in edg_G$ iff there is an edge in G with source x and target y.

We do not provide a formal definition of the syntax and semantics of MSOL, but rather provide examples, enough to allow the reader to generalise and derive other expressions of graph properties. However, we will be specific about some naming and notational conveniences. All relations are unary, and subsets of V_G[3]; we refer to them as *set variables* rather than relations, and use capital letters in MSOL formulae to distinguish them from set elements which are written in lower case. Finally, rather than writing $X(x)$ for set variable X, we follow Courcelle and write $x \in X$ to facilitate more readable formulae.

The following MSOL formula expresses the property that a graph G is undirected, non-empty, and not connected: [Cou10]

$$\begin{aligned} \varphi_1 \quad = \quad & \forall x, y(edg_G(x, y) \iff edg_G(y, x)) \\ & \wedge \exists X(\exists x(x \in X) \wedge \exists y(y \notin X) \\ & \quad \wedge \forall x, y(edg_G(x, y) \implies (x \in X \iff y \in X))) \end{aligned}$$

This example requires quantification over a single set variable, $X \subset V_G$. It is a strict subset since the formula requires that at least one node from V_G is not a member of the set (otherwise the formula would be satisfied by graphs consisting of only a single node, which are by definition connected). A relational structure denoting a connected graph cannot satisfy φ_1, since the node not in X must be connected by a path to a node in X. But if

[2] These are logical structures with relations only, i.e. no functions.

[3] The relational structure associated with G can be changed to have the set of edges as its domain. Some properties can only be expressed in MSOL with quantification over edges.

an edge has a node in X as its source, its target must also be a node in X. Likewise, if an edge has a node in X as its target and some node v as its source, the first conjunct of φ_1 states that there must be an edge with the node in X as its source and v as its target, hence v is also in X.

While first-order logic and nested conditions can express that a graph is undirected and non-empty, they are unable to express that a graph is not connected, since this requires a way of expressing that there exists at least one pair of nodes not connected by an arbitrary length path. MSOL can capture numerous other properties based on connectivity.

The following MSOL formula expresses that a graph G is undirected and 3-colourable: [Cou10]

$$\begin{aligned}\varphi_2 \quad = \quad & \forall x, y(edg_G(x,y) \iff edg_G(y,x)) \\ & \wedge \exists X, Y, Z(Part(X,Y,Z) \wedge \forall x, y(edg_G(x,y) \wedge x \neq y \\ & \quad \implies \neg(x \in X \wedge y \in X) \wedge \neg(x \in Y \wedge y \in Y) \\ & \quad\quad \wedge \neg(x \in Z \wedge y \in Z)))\end{aligned}$$

where $Part(X, Y, Z)$ is a formula of MSOL expressing that every node is partitioned into exactly one of the sets X, Y, or Z. That is:

$$\begin{aligned}Part(X,Y,Z) \quad = \quad & \forall x((x \in X \vee x \in Y \vee x \in Z) \\ & \quad \wedge(\neg(x \in X \wedge x \in Y) \wedge \neg(x \in Y \wedge x \in Z) \\ & \quad\quad \wedge \neg(x \in X \wedge x \in Z)))\end{aligned}$$

Intuitively, φ_2 states that there is a partitioning of the nodes into three sets (i.e. three colours) such that every non-looping edge is incident to nodes in distinct partitions (i.e. nodes with distinct colourings).

5.2.2 Expressiveness and Decidability

Expressiveness

MSOL can express numerous non-local graph properties that first-order logic and nested conditions cannot, including [Cou90]: the graph is connected, planar, k-colourable (for fixed $k \in \mathbb{N}$), Hamiltonian, a tree. Difficulties arise when one wants to express a property that require "counting". For example, the graph has as many a's as b's[4], or the graph has an even number of nodes. Full second-order logic is able to express these particular properties, using, for example, quantification over binary relations.

[4]One particular instance of this property might be: the graph has as many edges labelled with a as it has edges labelled with b.

Decidability

We consider again the model checking, satisfiability, and validity problems for MSOL formulae. The undecidability of the latter two problems follows from the undecidability of the corresponding problems for finite graphs and first-order formulae.

Model checking problem. *Instance.* A relational structure $\lfloor G \rfloor$ and a formula of MSOL φ. *Question.* Does $\lfloor G \rfloor$ satisfy φ? *Decidability.* Decidable.

Satisfiability problem. *Instance.* A formula of MSOL φ. *Question.* Does there exist a relational structure $\lfloor G \rfloor$ that satisfies φ? *Decidability.* Undecidable (but semidecidable for recursively enumerable classes of graphs).

Validity problem. *Instance.* A formula of MSOL φ. *Question.* Does every relational structure $\lfloor G \rfloor$ satisfy φ? *Decidability.* Undecidable (and not even semidecidable).

5.3 Hyperedge Replacement Conditions

While monadic second-order logic can capture many of the non-local graph properties that first-order logic and nested conditions are unable to, it is not always easy for non-experts to interpret (or indeed write) a sentence of the logic; the graphical presentation and intuitiveness of nested conditions is missing.

Recent research by Habel and Radke [HR10] has however developed a graph specification formalism — *hyperedge replacement conditions* (*HR conditions*) — which quite remarkably is more expressive than monadic second-order logic. The formalism is a generalisation of graph conditions (Section 5.1). The graphs of HR conditions can contain variables, which act as placeholders for graphs generated by a corresponding hyperedge replacement system (see [DHK97]). HR conditions can express many non-local graph properties in a very compact and intuitive way.

This section briefly introduces HR conditions and discusses the expressiveness and decidability of the formalism.

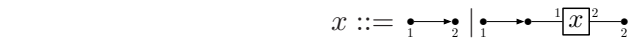
5.3.1 Definition and Examples

HR conditions are generalisations of nested conditions, and are defined and satisfied similarly (Section 5.1.1). We omit a full formal definition of the formalism (see [HR10]) because of this fact, and also because this would require formal treatment of the notions of graphs with variables, and hyperedge replacement systems. Rather, we introduce HR conditions intuitively and

by example, assuming that the reader has a basic understanding of nested conditions.

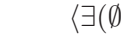
Graphs with variables contain nodes and edges, as usual, but also hyperedges. These are edges attached to an arbitrary number of nodes (whereas edges are attached to one or two), and for our purposes are labelled with a variable. Consider the condition $c = \exists(\emptyset \to$  $)$. The codomain of the morphism contains two nodes[5] and a hyperedge, labelled with a variable x (we follow [DHK97] and write variables in boxes). This particular hyperedge is attached to two nodes by its two "tentacles", which are identified by numbers. The i-th tentacle is identified by the number i, and is attached to the i-th attachment node. It is a placeholder for a possibly infinite number of graphs that can be generated by the hyperedge replacement system corresponding to the variable x.

One possible hyperedge replacement system $\mathcal{R}$ that could correspond to hyperedges labelled with variable x is given below, with rules in Backurs-Naur form:

$x ::=$ 

Here, the nodes labelled 1 and 2 are attachment points. The application of a rule for x to a graph containing a hyperedge labelled x first removes the hyperedge from the graph, then replaces it with the graph in the right-hand side of the rule, fusing the i-th attachment node with the i-th attachment point in the rule.

HR conditions are simply graph conditions with variables together with a hyperedge replacement system, i.e., $\langle c, \mathcal{R} \rangle$. HR conditions can finitely represent potentially infinite graph conditions, for example:

$\langle \exists(\emptyset \to$  $), \mathcal{R} \rangle$ and ( ) $\vee \exists($ $) \vee \exists($ $) \vee \ldots$

are satisfied by the same class of graphs, in this case, graphs which contain at least one path (of arbitrary length) between two distinct nodes.

We now present a number of simple HR conditions from [HR10] that express properties beyond what finite nested conditions are able to. For these examples, assume that $\mathcal{R}$ is defined as before, and that we are dealing with graphs where all nodes and edges are labelled with the blank label. Consider:

$\langle \exists($  $), \neg\exists($ $)), \mathcal{R} \rangle$

The HR condition can be read "the graph is non-empty and not connected" (compare the simplicity of the HR condition with the sentence of monadic

[5] Labelled with the blank label.

second-order logic in Section 5.2 for the same property on undirected graphs). Now, consider the HR condition:

$$\langle \neg\exists(\bullet \overset{1}{\underset{2}{\lessdot}} \boxed{x} \ ), \mathcal{R} \rangle$$

This expresses the property that "the graph is cycle-free". Note that here, the attachment node is the same for both tentacles of the hyperedge. Now consider:

$$\langle \exists(\ \boxed{x} \ , \neg\exists(\ \boxed{x} \ \bullet \ )), \mathcal{S} \rangle$$

where $\mathcal{S}$ comprises the rules:

$$x ::= \emptyset \mid \bullet \ \bullet \ \boxed{x}$$

The HR condition expresses that "the graph contains an even number of nodes", which is strictly in the class of properties expressible by full second-order logic, i.e. this cannot be expressed by monadic second-order logic on graphs. Note that this hyperedge has no tentacles and no attachment nodes.

5.3.2 Expressiveness and Decidability

Expressiveness

It is shown in [HR10] that monadic second-order graph formulae (in the sense of [Cou90, Cou97]) can be transformed into HR conditions, but that there is no transformation in the opposite direction. That is, there are HR conditions which express properties that monadic second-order logic is too weak to express. The relationship between HR conditions and full second-order logic is not yet fully clear.

HR conditions can express a large number of non-local graph properties, including (but not limited to): the graph is connected, planar, k-colourable (for fixed $k \in \mathbb{N}$), Hamiltonian. The formalism can also express a number of "counting" properties, the following of which are strictly in the class of properties expressible in full second-order logic: the graph has an even number of nodes, the same number of a's and b's, two paths of equal length.

Decidability

We consider again the model checking, satisfiability, and validity problems for HR conditions.

Model checking problem. *Instance.* A graph G and an HR condition $\langle c, \mathcal{R} \rangle$. *Question.* Does G satisfy $\langle c, \mathcal{R} \rangle$? *Decidability.* Decidable [HR10].

Satisfiability problem. *Instance.* An HR condition $\langle c, \mathcal{R} \rangle$. *Question.* Does there exist a graph that satisfies $\langle c, \mathcal{R} \rangle$? *Decidability.* Undecidable (but semidecidable for recursively enumerable classes of graphs).

Validity problem. *Instance.* An HR condition $\langle c, \mathcal{R} \rangle$. *Question.* Does every graph satisfy $\langle c, \mathcal{R} \rangle$? *Decidability.* Undecidable (and not even semidecidable).

The undecidability results of the latter two problems follow from the undecidability results of the corresponding problems for monadic second-order logic (since HR conditions are strictly more expressive).

5.4 Graph Reduction Specifications

The *Graph Reduction Specification* (GRS) framework was originally developed to allow one to specify classes of pointer structures (shapes of graphs) [BPR04, BPR03]. Informally, GRSs comprise a set of graph transformation (reduction) rules, and a so-called accepting graph, which is usually the smallest graph that belongs to the class of graphs the GRS is specifying. A graph under consideration belongs to the class of graphs specified by the GRS if it can be reduced to the accepting graph by repeated application of the reduction rules.

GRSs are the duals of graph grammars. They are essentially graph grammars with the rules reversed, with the accepting graph corresponding to the start graph of a grammar. Because of Uesu's result that double-pushout graph grammars can generate every recursively enumerable set of graphs [Ues78], GRSs can define every recursively enumerable set of graphs (that excludes the empty graph) [BPR04].

5.4.1 Definition and Examples

A GRS requires two components in order to define a class of graphs: an accepting graph, and a set of graph transformation rules. The accepting graph is usually the smallest graph in the class. For example, the accepting graph for the class of connected graphs is simply an isolated node. The graph transformation rules, based on the usual double-pushout approach, are typically rules that reduce the size of the graph whilst maintaining an invariant: if a graph belongs (does not belong) to the class of graphs specified by the GRS, then the graph resulting from a rule application also belongs (does not belong) to the class. For example, the rules of a GRS that specifies trees would never transform a non-tree into a tree, or vice versa.

A graph G is specified by a GRS γ, written $G \in L(\gamma)$, if it can be reduced to the accepting graph by repeated application of the graph transformation rules. If the graph is indeed specified by the GRS, then after each application

of any of the rules, the graphs that result remain members of the class. When there is a need for "intermediary" transformations (i.e. more than one rule application is required to achieve some sort of reduction), the graph can be labelled with "nonterminal" symbols. Such graphs are not considered to be members of the class. Finally, it is often necessary to restrict where rules can be applied to in a graph; signature restrictions facilitate this (see [BPR04]).

Consider the GRS defined in Figure 5.1 (from [BPR03]), which is intended to specify graphs that are binary trees, i.e. trees in which all the nodes have at most two children. This particular GRS applies to graphs that are restricted by a binary tree signature: a node can have no children (leaf, labelled L), one child (unary, labelled U), or two children (binary, labelled B). The signature requires that U nodes have exactly one outgoing edge, labelled c, and that B nodes have exactly two outgoing edges, labelled l and r. The reduction rules assume the graph to be in this form, and if it is, the rules do not transform a non-binary tree into a binary tree (note that the signature restriction is adhered to by the rules: if a node's children are both leafs, pruning those children causes the node to become a leaf itself). The accepting graph, which any binary tree is eventually reduced to by these rules, is the minimal (non-empty) binary tree, i.e. an isolated node.

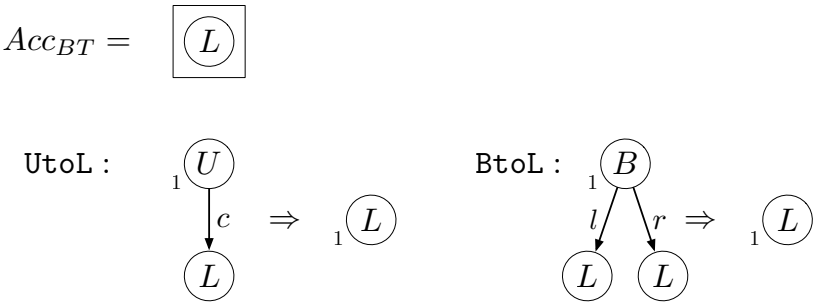


Figure 5.1: A GRS specifying graphs that are binary trees

5.4.2 Expressiveness and Decidability

As GRSs can define every recursively enumerable set of graphs, they are the most powerful graph specification formalism we have considered in this chapter. However, this expressive power comes with a consequence to decidability — arbitrary reduction rules can allow one to specify languages for which the membership problem is undecidable [BPR04].

We consider the membership problems for GRSs. Membership of a GRS can be seen as corresponding to the model checking problem for the previous graph specification formalisms.

Membership problem. *Instance.* A graph G and a GRS γ. *Question.* Is $G \in L(\gamma)$? *Decidability.* Undecidable, unless the GRS is known to

be terminating (i.e. cannot apply rules to G for an infinite number of times).

The decision problems for GRSs that correspond to our previous notions of satisfiability and validity are of little interest to us. The problem to determine whether or not there is a graph that belongs to $L(\gamma)$ for a GRS γ is trivial: if γ has an accepting graph Acc_γ, then $Acc_\gamma \in L(\gamma)$. The problem to determine whether or not every graph belongs to $L(\gamma)$ is equally uninteresting, since we would not usually want to design a GRS that specifies every graph.

5.5 Summary

We conclude this chapter on specification formalisms and logics for graphs, having:

1. Introduced a number of specification formalisms, namely nested conditions, HR conditions, and GRSs.
2. Discussed first-order, monadic second-order, and full second-order logic on graphs; with particular attention given to monadic second-order logic and the work by Courcelle.
3. Considered the expressive power of these logics and related them to the specification formalisms. We have that:
   - Graph conditions are equivalently expressive to first-order graph formulae; neither are able to express non-local properties.
   - HR conditions are *more* expressive than monadic second-order logic graph formulae.
   - GRSs can specify every recursively enumerable set of graphs.
4. Considered various decidability properties of the logics and specification formalisms.

Chapter 6

Verifying Graph Transformation Systems

Recent years have seen an increased interest in developing techniques for verifying properties of graph transformation systems. Up until now, research has focused on verifying sets of rules (that can be applied arbitrarily) and graph grammars, rather than the graph programs (i.e. graph transformation systems with control constructs) that concern us. The main exception is the work of Habel et al. [HPR06] on weakest preconditions for so-called high-level programs.

We aim in this chapter to give a high-level discussion of some important, recent work on the verification of graph transformation systems. We begin by comparing two distinct approaches to model checking graph transformation systems in Section 6.1. We follow this in Section 6.2 by looking at another approach related to model checking, based on the idea of McMillan unfoldings for Petri nets. Finally, we discuss weakest preconditions for high-level programs in Section 6.3.

6.1 Model Checking Graph Transformation Systems

Model checking a system involves traversing all its possible execution paths and checking that particular correctness properties hold in every state. In a graph transformation system, these states correspond to graphs, and the transitions to rule applications. The requirement to traverse all possible execution paths implies that all possible rule applications and all possible matchings must be explored [RSV04].

We briefly describe in this section two implemented tools for model checking graph transformation systems, CheckVML and GROOVE, which approach the problem in very different ways. Rensink et al. compare the tools in greater detail in a tool comparison paper [RSV04].

6.1.1 CheckVML

The approach used by CheckVML [SV03] is to take advantage of off-the-shelf model checking software. The tool has a "pre-processing" phase. It takes a graph transformation system as input, as well as so-called property graphs which encode safety, reachability, or danger properties of the system. These are translated into Promela and linear temporal logic equivalents, in order for the formal analysis to be carried out in SPIN.

This approach benefits from being able to take full advantage of years of general research in model checking, and from being easily adaptable to a number of tools that have already been implemented [RSV04]. Unfortunately, by translating away from the core concepts of graphs and graph transformation systems, it becomes difficult to suitably handle things like symmetries in graphs.

6.1.2 GROOVE

GROOVE [Ren03] takes a rather different approach to model checking graph transformation systems, aiming instead to work with graphs and graph transformation systems throughout the entire model checking process. Infact, all states are explicitly graphs, and all transitions explicitly the applications of graph transformation rules [RSV04]. Safety, reachability, and danger properties can be specified by a graph-based logic (or alternatively, one can work with a graph specification formalism and then translate it into a sentence of logic).

While it is more natural to work with graphs and applications of rules, as in the state space generated by GROOVE, research in traditional model checking is only indirectly applicable to the tool. It is more difficult to adapt GROOVE to interact with the various implemented model checking tools than it is for CheckVML.

6.2 Infinite-State Graph Transformation Systems

Baldan et al. [BCK08] developed the foundations of a methodology for verifying infinite-state graph transformation systems. Their approach is based on the idea of McMillan unfoldings for Petri nets, which are single structures that fully describe the concurrent behaviour of given systems. An idea of their work was to exploit the relationship between Petri nets and graph transformation systems to develop verification techniques.

The full unfolding of a graph transformation system is a single structure that completely describes its concurrent behaviour. This includes all possible graph transformation steps and their mutual dependencies, and in addition all reachable states [BCK08]. Baldan et al. construct finite approximations of full unfoldings (to a level of accuracy that can be set). They

demonstrate that these finite approximations can be used for the verification of the original infinite-state systems, and describe a tool that constructs the approximations.

6.3 Weakest Preconditions of High-Level Programs

In [HPR06], Habel, Pennemann, and Rensink follow Dijkstra [Dij75, Dij76] and define a weakest preconditions semantics for so-called "high-level programs". High-level programs are based on the same minimal and complete core as graph programs [HP01], but are lifted to more abstract objects than graphs (in this report we assume all objects to be graphs), and are not able to compute expressions over labels like GP.

Habel et al. use the nested condition specification formalism to describe preconditions and postconditions of their programs. The weakest precondition transformations are then defined inductively over the structure of programs, each transformation resulting in a nested condition.

To show that executing a program P on a graph satisfying condition c results in a graph satisfying condition d, one must show that the condition $c \Longrightarrow \mathrm{Wp}(P, d)$ is valid (i.e. satisfied by all graphs). In the context of finite graphs, the validity problem is undecidable (and not even semidecidable [Pen08]) — a solution can be searched for, but the search may run forever without returning an answer.

Since there are transformations from graph conditions to sentences of first-order logic, it is possible to use off-the-shelf first-order theorem provers to try to find an answer. However, they need to be restricted to graphs by axioms which themselves become part of the problem [Pen08]. For this reason, Pennemann investigates in [Pen08] a resolution-like calculus specifically for nested conditions, that can refute conditions in conjunctive normal form. An implemented theorem prover based on this calculs, ProCon, is described in his thesis [Pen09].

6.4 Summary

We conclude this chapter on verifying graph transformation systems, having:

1. Considered a number of approaches based on model checking.
2. Described the work by Habel et al. on the verification of programs based on the same minimal and complete core as GP.
3. Implied that much research in the area of graph program verification remains to be done!

Part III

Preliminary Results

Chapter 7

Hoare Logic for Graph Programs

This chapter presents the results of our first year of study. Together with Detlef Plump, we were able to devise a sound Hoare calculus for proving the correctness of graph programs in the sense of "partial correctness" (i.e. true when a program terminates with a graph).

What follows is nearly identical to our accepted ICGT 2010 paper [PP10a], updated with references to the literature review of this report where appropriate (for example, when discussing GP's semantic function).

A shorter paper on this work was accepted for the Theory Workshop at VSTTE 2010 (see [PP10c]).

7.1 Introduction

Recent years have seen an increased interest in formally verifying properties of graph transformation systems, motivated by the many applications of graph transformation to specification and programming. Typically, this work has focused on verification techniques for sets of graph transformation rules or graph grammars, see for example [RSV04, BCK08, KK08, BHE09, HP09].

Graph transformation languages and systems such as PROGRES [SWZ99], AGG [Tae04], Fujaba [NNZ00] and GrGen [GBG+06], however, allow one to use control constructs on top of graph transformation rules for solving graph problems in practice. The challenge to verify programs in such languages has, to the best of our knowledge, not yet been addressed.

A first step beyond the verification of plain sets of rules has been made by Habel, Pennemann and Rensink in [HPR06], by providing a construction for weakest preconditions of so-called high-level programs. These programs allow one to use constructs such as sequential composition and as-long-as-possible iteration over sets of conditional graph transformation rules. The verification method follows Dijkstra's approach to program verification: one

calculates the weakest precondition of a program and then needs to prove that it follows from the program's precondition. High-level programs fall short of practical graph transformation languages though, in that their rules cannot perform computations on labels (or attributes).

In this paper, we present an approach for verifying programs in the graph programming language GP [Plu09, MP08a]. Rather than adopting a weakest precondition approach, we follow Hoare's seminal paper [Hoa69] and devise a calculus of proof rules which are directed by the syntax of the language's control constructs. Similar to classical Hoare logic, the calculus aims at human-guided verification and allows the compositional construction of proofs.

The pre- and postconditions of our calculus are nested conditions [HP09], extended with expressions for labels and so-called assignment constraints; we refer to them as *E-conditions*. The extension is necessary for two reasons. Firstly, when a label alphabet is infinite, it is impossible to express a number of simple properties with finite nested conditions. For example, one cannot express with a finite nested condition that a graph over the set of integers is non-empty, since it is impossible to finitely enumerate every integer. Secondly, the conditions in [HP09] cannot express relations between labels such as "x and y are integers and $\mathrm{x}^2 = \mathrm{y}$". Such relations can be expressed, however, in GP's rule schemata.

We briefly review the preliminaries in Section 7.2 and graph programs in Section 7.3. Following this, we present E-conditions in Section 7.4, and then use them to define a proof system for GP in Section 7.5, where its use will be demonstrated by proving a property of a graph colouring program. In Section 7.6, we formally define the two transformations of E-conditions used in the proof system, before proving the axiom schemata and inference rules sound in the sense of partial correctness, with respect to GP's operational semantics [Plu09, PS10]. Finally, we conclude in Section 7.7. A long version of this paper with the abstract syntax and operational semantics of GP, as well as detailed proofs of results, is available online [PP10b].

7.2 Graphs, Assignments, and Substitutions

Graph transformation in GP is based on the double-pushout approach with relabelling (see Section 2.2.3 and [HP02]). This framework deals with partially labelled graphs, whose definition we recall below. We deal with two classes of graphs, "syntactic" graphs labelled with expressions and "semantic" graphs labelled with (sequences of) integers and strings. We also introduce assignments which translate syntactic graphs into semantic graphs, and substitutions which operate on syntactic graphs.

A *graph* over a label alphabet $\mathcal{C}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to$

V_G are the *source* and *target* functions for edges, $l_G\colon V_G \to \mathcal{C}$ is the partial node labelling function and $m_G\colon E_G \to \mathcal{C}$ is the (total) edge labelling function. Given a node v, we write $l_G(v) = \bot$ to express that $l_G(v)$ is undefined. Graph G is *totally labelled* if l_G is a total function.

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

A *graph morphism* $g\colon G \to H$ between graphs G and H consists of two functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserve sources, targets and labels; that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, and $l_H(g(v)) = l_G(v)$ for all v such that $l_G(v) \neq \bot$. Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x. It is *injective* (*surjective*) if g_V and g_E are injective (surjective). It is an *isomorphism* if it is injective, surjective and satisfies $l_H(g_V(v)) = \bot$ for all nodes v with $l_V(v) = \bot$. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

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

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

The other label alphabet we are using consists of expressions according to the EBNF grammar of Figure 7.1, where VarId is a syntactic class[1] of variable identifiers. We write $\mathcal{G}(\mathrm{Exp})$ for the set of all graphs over the syntactic class Exp.

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

Figure 7.1: Syntax of expressions

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

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

an expression e, α is *well-typed* for e if for every term $t_1 \oplus t_2$ in e, with $\oplus \in \text{ArithOp}$, we have $\alpha(\mathtt{x}) \in \mathbb{Z}$ for all variable identifiers $\mathtt{x}$ in $t_1 \oplus t_2$. In this case we inductively define the value $e^\alpha \in \mathcal{L}$ as follows. If e is a numeral or a sequence of characters, then e^α is the integer or character string represented by e. If e is a variable identifier, then $e^\alpha = \alpha(e)$. Otherwise, if e has the form $t_1 \oplus t_2$ with $\oplus \in \text{ArithOp}$ and $t_1, t_2 \in \text{Term}$, then $e^\alpha = t_1^\alpha \oplus_\mathbb{Z} t_2^\alpha$ where $\oplus_\mathbb{Z}$ is the integer operation represented by $\oplus$. Finally, if e has the form t_e_1 with $t \in \text{Term} \cup \text{String}$ and $e_1 \in \text{Exp}$, then $e^\alpha = t^\alpha e_1^\alpha$ (the concatenation of t^α and e_1^α).

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

A *substitution* is a mapping $\sigma\colon \text{VarId} \to \text{Exp}$. Given an expression e, σ is *well-typed* for e if for every term $t_1 \oplus t_2$ in e, with $\oplus \in \text{ArithOp}$, we have $\sigma(\mathtt{x}) \in \text{Term}$ for all variable identifiers $\mathtt{x}$ in $t_1 \oplus t_2$. In this case the expression e^σ is obtained from e by replacing every occurrence of a variable $\mathtt{x}$ with $\sigma(\mathtt{x})$. Given a graph G in $\mathcal{G}(\text{Exp})$, we write G^σ for the graph that is obtained by replacing each label e with e^σ. If $g\colon G \to H$ is a graph morphism between graphs in $\mathcal{G}(\text{Exp})$, then g^σ denotes the morphism $\langle g_V^\sigma, g_E^\sigma \rangle\colon G^\sigma \to H^\sigma$.

Given an assignment α, the substitution σ_α *induced* by α maps every variable $\mathtt{x}$ to the expression that is obtained from $\alpha(\mathtt{x})$ by replacing integers and strings with their syntactic counterparts. For example, if $\alpha(\mathtt{x})$ is the sequence $56, a, bc$, where 56 is an integer and a and bc are strings, then $\sigma_\alpha(\mathtt{x}) = 56\_\text{"a"}\_\text{"bc"}$.

7.3 Graph Programs

We briefly review GP's conditional rule schemata and discuss an example program. Technical details (including an operational semantics later used in our soundness proof) and further examples can be found in Chapter 3 and [Plu09, PS10].

7.3.1 Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of graph programs: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise graph transformation rules in the double-pushout approach with relabelling [HP02], in that labels can contain expressions over parameters of type integer or string. Figure 7.2 shows a conditional rule schema consisting of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema which are graphs

in $\mathcal{G}$(Exp), the node identifiers 1, 2, 3 specifying which nodes are preserved, and the keyword `where` followed by a rule schema condition.

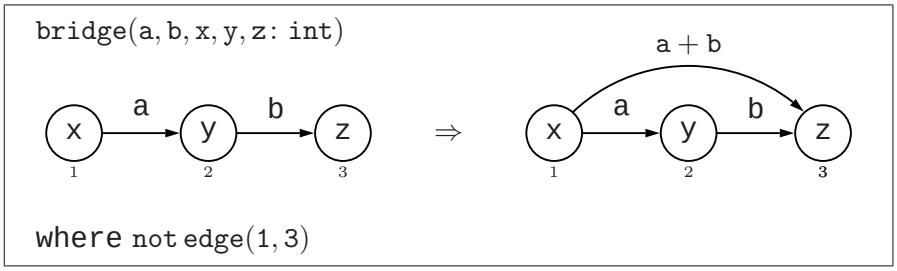


Figure 7.2: A conditional rule schema

In the GP programming system [MP08a], rule schemata are constructed with a graphical editor. Labels in the left graph comprise only variables and constants (no composite expressions) because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`, where all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1, 3)` in the condition of Figure 7.2 forbids an edge from node 1 to node 3 when the left graph is matched. The grammar of Figure 7.3 defines the syntax of rule schema conditions, where Term is the syntactic class defined in Figure 7.1.

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

Figure 7.3: Syntax of rule schema conditions

Conditional rule schemata represent possibly infinite sets of conditional graph transformation rules over graphs in $\mathcal{G}(\mathcal{L})$, and are applied according to the double-pushout approach with relabelling. A rule schema $L \Rightarrow R$ with condition Γ represents conditional rules $\langle\langle L^\alpha \leftarrow K \rightarrow R^\alpha\rangle, \Gamma^{\alpha,g}\rangle$, where K consists of the preserved nodes (which are unlabelled) and $\Gamma^{\alpha,g}$ is a predicate on graph morphisms $g\colon L^\alpha \rightarrow G$ (see [Plu09, PS10]).

7.3.2 Programs

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

Example 9 A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each non-looping edge have different colours. The program `colouring` in Figure 7.4 produces a colouring for every integer-labelled input graph, recording colours as so-called tags. In general, a tagged label is a sequence of expressions separated by underscores.

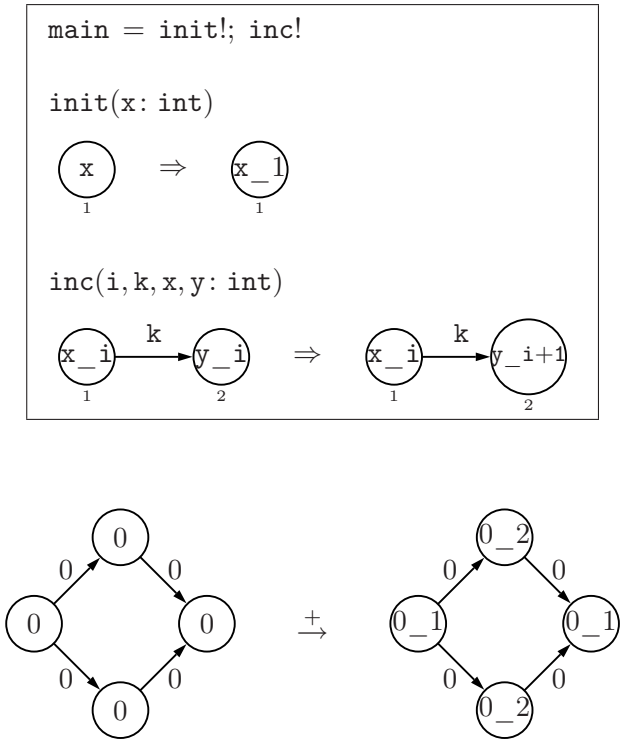


Figure 7.4: The program `colouring` and one of its executions

The program initially colours each node with 1 by applying the rule schema `init` as long as possible, using the iteration operator '`!`'. It then repeatedly increments the target colour of edges with the same colour at both ends. Note that this process is highly nondeterministic: Figure 7.4 shows an execution producing a colouring with two colours, but a colouring with three colours could have been produced for the same input graph.

It is easy to see that whenever `colouring` terminates, the resulting graph is a correctly coloured version of the input graph. This is because the

output cannot contain an edge with the same colour at both incident nodes, as then `inc` would have been applied at least one more time. Also, it can be shown that every execution of the program terminates after at most a quadratic number of rule schema applications [Plu09].

7.4 Nested Graph Conditions with Expressions

We introduce nested graph conditions with expressions (or E-conditions) to specify graph properties in the pre- and postconditions of graph programs. E-conditions extend the nested conditions (see Section 5.1) of [HP09] with expressions for labels, and assignment constraints which restrict the values that can be assigned to variables. The resulting conditions can be considered as representations of possibly infinite sets of ordinary nested conditions.

Definition 11 (Assignment constraint) An *assignment constraint* is a Boolean expression conforming to the grammar in Figure 7.5. We require that the arguments of the operators $>$, $<$, $\geq$ and $\leq$ belong to the syntactic class Term and that the arguments of $=$ and $\neq$ belong both to either Term, String or Exp $-$ (Term $\cup$ String). (See Figure 7.1 for the definition of Term, String and Exp.)

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

Figure 7.5: Syntax of assignment constraints

Given an assignment constraint γ and an assignment $\alpha\colon \text{VarId} \to \mathcal{L}$, the *value* γ^α in $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ is inductively defined[2]. If $\gamma = \text{true}$, then $\gamma^\alpha = \mathtt{tt}$. If γ has the form $e_1 \bowtie e_2$ with $\bowtie \in \text{ACRelOp}$ and $e_1, e_2 \in \text{Exp}$, then $\gamma^\alpha = \mathtt{tt}$ if and only if $e_1^\alpha \bowtie_{\mathcal{L}} e_2^\alpha$ where $\bowtie_{\mathcal{L}}$ is the obvious relation on $\mathcal{L}$ represented by $\bowtie$. If $\gamma = \neg\gamma_1$ with $\gamma_1 \in \text{ACBoolExp}$, then $\gamma^\alpha = \mathtt{tt}$ if and only if $\gamma_1^\alpha = \mathtt{ff}$. If $\gamma = \gamma_1 \oplus \gamma_2$ with $\gamma_1, \gamma_2 \in \text{ACBoolExp}$ and $\oplus \in \text{ACBoolOp}$, then $\gamma^\alpha = \gamma_1^\alpha \oplus_{\mathbb{B}} \gamma_2^\alpha$ where $\oplus_{\mathbb{B}}$ is the Boolean operation on $\mathbb{B}$ represented by $\oplus$. Finally, if γ has the form $\text{type}(\mathtt{x}) = t$ with $\mathtt{x} \in \text{VarId}$ and $t \in \text{Type}$, then $\gamma^\alpha = \mathtt{tt}$ if and only if $\mathit{type}(\alpha(\mathtt{x})) = t$, where the function $\mathit{type}\colon \mathcal{L} \to \text{Type}$ is defined by

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

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

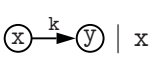
Example 10 Consider the assignment constraint $\gamma = \mathtt{a} > \mathtt{b} \wedge \mathtt{b} \neq 0 \wedge \text{type}(\mathtt{a}) = \text{int}$. Let $\alpha_1 = (\mathtt{a} \mapsto 5, \mathtt{b} \mapsto 1)$ and $\alpha_2 = (\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 0)$. Then $\gamma^{\alpha_1} = \mathtt{tt}$ and $\gamma^{\alpha_2} = \mathtt{ff}$.

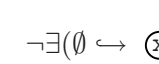
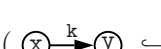
Note that variables in assignment constraints do not have a type per se, unlike the variables in GP rule schemata. Rather, the operator 'type' can be used to constrain the type of a variable. Note also that an assignment constraint such as $\mathtt{a} > 5 \wedge \text{type}(\mathtt{a}) = \text{string}$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

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

Definition 12 (E-condition) An *E-condition* c over a graph P is of the form true or $\exists(a|\gamma, c')$, where $a\colon P \hookrightarrow C$ is an injective[3] graph morphism with $P, C \in \mathcal{G}(\text{Exp})$, γ is an assignment constraint, and c' is an E-condition over C. Moreover, Boolean formulas over E-conditions over P yield E-conditions over P, that is, $\neg c$ and $c_1 \wedge c_2$ are E-conditions over P, where c_1 and c_2 are E-conditions over P.

For brevity, we write false for $\neg$true, $\exists(a|\gamma)$ for $\exists(a|\gamma, \text{true})$, $\exists(a, c')$ for $\exists(a|\text{true}, c')$, and $\forall(a|\gamma, c')$ for $\neg\exists(a|\gamma, \neg c')$. In examples, when the domain of morphism $a\colon P \hookrightarrow C$ can unambiguously be inferred, we write only the codomain C. For instance, an E-condition $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow C'))$ can be written as $\exists(C, \exists(C'))$, where the domain of the outermost morphism is the empty graph, and the domain of the nested morphism is the codomain of the encapsulating E-condition's morphism. An E-condition over a graph morphism whose domain is the empty graph is referred to as an *E-constraint*. We later refer to E-conditions over left- and right-hand sides of rule schemata as *E-app-conditions*.

Example 11 The E-condition $\forall($  $\mid \mathtt{x} > \mathtt{y}, \exists($  $))$ (which is an E-constraint) expresses that every pair of adjacent integer-labelled nodes with the source label greater than the target label has a loop incident to the source node. The unabbreviated version of the condition is as follows:

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

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

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

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

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

7.5 A Hoare Calculus for Graph Programs

We present a system of partial correctness proof rules for GP, in the style of Hoare [AdO09], using E-constraints as the assertions. We demonstrate the proof system by proving a property of our earlier `colouring` graph program, and sketch a proof of the rules' soundness according to GP's operational semantics [Plu09, PS10].

Definition 13 (Partial correctness) A graph program P is *partially correct* with respect to a precondition c and a postcondition d (both of which are E-constraints), if for every graph $G \in \mathcal{G}(\mathcal{L})$, $G \models c$ implies $H \models d$ for every graph H in $[\![P]\!]G$.

Here, $[\![\_]\!]$ is GP's semantic function (see Section 3.5), and $[\![P]\!]G$ is the set of all graphs resulting from executing program P on graph G. Note that partial correctness of a program P does not entail that P will actually terminate on graphs satisfying the precondition.

Given E-constraints c, d and a program P, a triple of the form $\{c\}\ P\ \{d\}$ expresses the claim that whenever a graph G satisfies c, then any graphs resulting from the application of P to G will satisfy d. Our proof system in Figure 7.6 operates on such triples. As in classical Hoare logic [AdO09], we use the proof system to construct proof trees, combining axiom schemata and inference rules (an example will follow). We let c, d, e, inv range over E-constraints, P, Q over arbitrary command sequences, r, r_i over conditional rule schemata, and $\mathcal{R}$ over sets of conditional rule schemata.

Two transformations — App and Pre — are required in some of the assertions. Intuitively, App takes as input a set $\mathcal{R}$ of conditional rule schemata, and transforms it into an E-condition specifying the property that a rule in $\mathcal{R}$ is applicable to the graph. Pre constructs the weakest precondition such that if $G \models \text{Pre}(r, c)$, and the application of r to G results in a graph H,

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

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

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

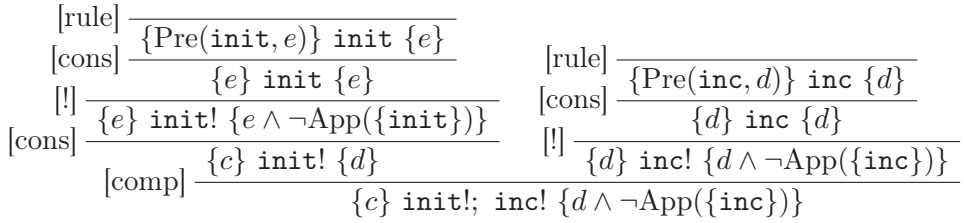
$$[\text{if}]\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\} \qquad \{c \wedge \neg\text{App}(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

Figure 7.6: Partial correctness proof system for GP

then $H \models c$. The transformation Pre is informally described by the following steps: (1) form a disjunction of right E-app-conditions for the possible overlappings of c and the right-hand side of the rule schema r, (2) convert the right E-app-condition into a left E-app-condition (i.e. over the left-hand side of r), (3) nest this within an E-condition that is quantified over every L and also accounts for the applicability of r.

Note that two of the proof rules deal with programs that are restricted in a particular way: both the condition C of a branching command `if` C `then` P `else` Q and the body P of a loop $P!$ must be sets of conditional rule schemata. This restriction does not affect the computational completeness of the language, because in [HP01] it is shown that a graph transformation language is complete if it contains single-step application and as-long-as-possible iteration of (unconditional) sets of rules, together with sequential composition.

Example 12 Figure 7.7 shows a proof tree for the `colouring` program of Figure 7.4. It proves that if `colouring` is executed on a graph in which the node labels are exclusively integers, then any graph resulting will have the property that each node label is an integer with a colour attached to it, and that adjacent nodes have distinct colours. That is, it proves the triple $\{\neg\exists(\ \textcircled{a}\ \mid \text{type}(\mathtt{a}) \neq \text{int})\}$ `init!; inc!` $\{\forall(\ \textcircled{a}\ , \exists(\ \textcircled{a}_1\ \mid \mathtt{a} = \mathtt{b\_c} \wedge \text{type}(\mathtt{b},\mathtt{c}) = \text{int})) \wedge \neg\exists(\ \textcircled{x\_i} \xrightarrow{k} \textcircled{y\_i}\ \mid \text{type}(\mathtt{i},\mathtt{k},\mathtt{x},\mathtt{y}) = \text{int})\}$. For conciseness, we abuse our notation (in this, and later examples), and allow $\text{type}(\mathtt{x}_1,\ldots,\mathtt{x}_n) = \text{int}$ to represent $\text{type}(\mathtt{x}_1) = \text{int} \wedge \ldots \wedge \text{type}(\mathtt{x}_n) = \text{int}$.



$$
\begin{array}{rcl}
c & = & \neg\exists(\ \textcircled{a}\ \mid \mathrm{type}(\mathtt{a}) \neq \mathrm{int}) \\
d & = & \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \mathtt{a} = \mathtt{b\_c} \wedge \mathrm{type}(\mathtt{b},\mathtt{c}) = \mathrm{int})) \\
e & = & \forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \mathrm{type}(\mathtt{a}) = \mathrm{int}) \vee \exists(\ \textcircled{a}_1 \mid \mathtt{a} = \mathtt{b\_c} \wedge \mathrm{type}(\mathtt{b},\mathtt{c}) = \mathrm{int})) \\
\neg\mathrm{App}(\{\mathtt{init}\}) & = & \neg\exists(\ \textcircled{x} \mid \mathrm{type}(\mathtt{x}) = \mathrm{int}) \\
\neg\mathrm{App}(\{\mathtt{inc}\}) & = & \neg\exists(\ \textcircled{x\_i} \xrightarrow{\mathtt{k}} \textcircled{y\_i} \mid \mathrm{type}(\mathtt{i},\mathtt{k},\mathtt{x},\mathtt{y}) = \mathrm{int}) \\
\mathrm{Pre}(\mathtt{init}, e) & = & \forall(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \mathrm{type}(\mathtt{x}) = \mathrm{int}, \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \mathrm{type}(\mathtt{a}) = \mathrm{int}) \\
 & & \vee\ \exists(\ \textcircled{x}_1\ \textcircled{a}_2 \mid \mathtt{a} = \mathtt{b\_c} \wedge \mathrm{type}(\mathtt{b},\mathtt{c}) = \mathrm{int})) \\
\mathrm{Pre}(\mathtt{inc}, d) & = & \forall(\ \textcircled{x\_i}_1 \xrightarrow{\mathtt{k}} \textcircled{y\_i}_2\ \textcircled{a}_3 \mid \mathrm{type}(\mathtt{i},\mathtt{k},\mathtt{x},\mathtt{y}) = \mathrm{int}, \\
 & & \exists(\ \textcircled{x\_i}_1 \xrightarrow{\mathtt{k}} \textcircled{y\_i}_2\ \textcircled{a}_3 \mid \mathtt{a} = \mathtt{b\_c} \wedge \mathrm{type}(\mathtt{b},\mathtt{c}) = \mathrm{int}))
\end{array}
$$

Figure 7.7: A proof tree for the program `colouring` of Figure 7.4

7.6 Transformations and Soundness

We provide full definitions of the transformations App and Pre in this section. In order to define Pre, it is necessary to first define the intermediary transformations A and L, which are adapted from basic transformations of nested conditions [HP09]. Following this, we will show that our proof system is sound according to the operational semantics of GP.

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

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

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

The transformation App gives an E-constraint that can only be satisfied by a graph G if at least one of the rule schemata from $\mathcal{R}$ can directly derive

a graph H from G. The idea is to generate a disjunction of E-constraints from the left-hand sides of the rule schemata, with nested E-conditions for handling restrictions on the application of the rule schemata (such as the dangling condition when deleting nodes).

Construction. Define $\mathrm{App}(\{\}) = \mathrm{false}$ and $\mathrm{App}(\{r_1, \ldots, r_n\}) = \mathrm{app}(r_1) \vee \ldots \vee \mathrm{app}(r_n)$. For a rule schema $r_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ with rule schema condition Γ_i, define $\mathrm{app}(r_i) = \exists(\emptyset \hookrightarrow L_i | \gamma_{r_i}, \neg\mathrm{Dang}(r_i) \wedge \tau(L_i, \Gamma_i))$ where γ_{r_i} is a conjunction of expressions constraining the types of variables in r_i to the corresponding types in the declaration of r_i. For example, if r_i corresponds to the declaration of `inc` (Figure 7.4), then γ_{r_i} would be the Boolean expression $\mathrm{type}(\mathtt{i}) = \mathrm{int} \wedge \mathrm{type}(\mathtt{k}) = \mathrm{int} \wedge \mathrm{type}(\mathtt{x}) = \mathrm{int} \wedge \mathrm{type}(\mathtt{y}) = \mathrm{int}$.

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

We define $\tau(L_i, \Gamma_i)$ inductively (see Figure 7.3 for the syntax of rule schema conditions). If there is no rule schema condition, then $\tau(L_i, \Gamma_i) = \mathrm{true}$. If Γ_i has the form $t_1 \bowtie t_2$ with t_1, t_2 in Term and $\bowtie$ in RelOp, then $\tau(L_i, \Gamma_i) = \exists(L_i \hookrightarrow L_i | t_1 \bowtie_{\mathrm{ACRelOp}} t_2)$ where $\bowtie_{\mathrm{ACRelOp}}$ is the symbol in ACRelOp that corresponds to the symbol $\bowtie$ from RelOp. If Γ_i has the form `not` b_i with b_i in BoolExp, then $\tau(L_i, \Gamma_i) = \neg\tau(L_i, b_i)$. If Γ_i has the form $b_1 \oplus b_2$ with b_1, b_2 in BoolExp and $\oplus$ in BoolOp, then $\tau(L_i, \Gamma_i) = \tau(L_i, b_1) \oplus_{\wedge,\vee} \tau(L_i, b_2)$ where $\oplus_{\wedge,\vee}$ is $\wedge$ for `and` and $\vee$ for `or`. Finally, if Γ_i is of the form `edge`(n_1,n_2) with n_1, n_2 in Node, then $\tau(L_i, \Gamma_i) = \exists(L_i \hookrightarrow L_i')$ where L_i' is a graph isomorphic to L_i, except for an additional edge whose source is the node with identifer n_1, whose target is the node with identifier n_2, and whose label is a variable distinct from all others in use.

Proposition 2 (From E-constraints to E-app-conditions) *There is a transformation* A *such that, for all E-constraints* c*, all rule schemata* $r \colon L \Rightarrow R$ *sharing no variables with* c[6]*, and all injective graph morphisms* $h \colon R^{\alpha} \hookrightarrow H$ *where* $H \in \mathcal{G}(\mathcal{L})$ *and* α *is a well-typed assignment,*

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

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

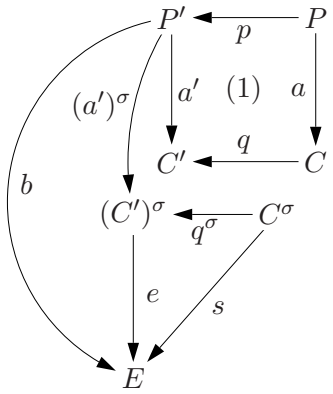
[4] We equate morphisms with isomorphic codomains, so A is finite.

[5] A pushout is *natural* if is simultaneously a pullback [HP02].

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

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

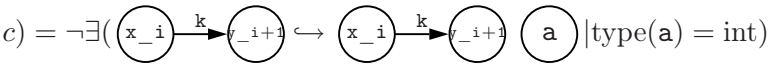
$$\begin{aligned} \mathrm{A}'(p, \mathrm{true}) &= \mathrm{true}, \\ \mathrm{A}'(p, \exists(a|\gamma, c')) &= \bigvee_{\sigma\in\Sigma} \bigvee_{e\in\varepsilon_\sigma} \exists(b|\gamma^\sigma, \mathrm{A}'(s, (c')^\sigma)). \end{aligned}$$



Construct the pushout (1) of p and a leading to injective graph morphisms $a' : P' \hookrightarrow C'$ and $q : C \hookrightarrow C'$. The finite double disjunction $\bigvee_{\sigma\in\Sigma}\bigvee_{e\in\varepsilon_\sigma}$ ranges first over substitutions from Σ, which have the special form $(\mathtt{a}_1 \mapsto \beta_1, \ldots, \mathtt{a}_k \mapsto \beta_k)$ where each $\mathtt{a}_i$ is a distinct label variable from C that is not also in P, and each β_i is a portion (or the entirety) of some label from P'. For each $\sigma \in \Sigma$, the double disjunction then ranges over every surjective graph morphism $e\colon (C')^\sigma \to E$ such that $b = e \circ (a')^\sigma$ and $s = e \circ q^\sigma$ are injective graph morphisms. The set ε_σ is the set of such surjective graph morphisms for a particular σ, the codomains of which we consider up to isomorphism. For a surjective graph morphism $e_1\colon (C_1')^{\sigma_1} \to E_1$, E_1 is considered redundant and is excluded from the disjunction if there exists a surjective graph morphism, $e_2\colon (C_2')^{\sigma_2} \to E_2$, such that $E_2 \ncong E_1$, and there exists some $\sigma \in \Sigma$ such that $E_2^\sigma \cong E_1$.

The transformation A is extended for Boolean formulas over E-conditions in the same way as transformations over conditions (see [HP09]).

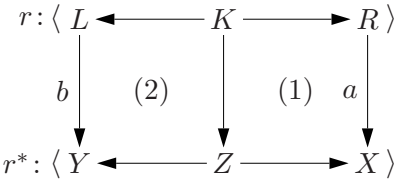
Example 13 Let r correspond to the rule schema `inc` (Figure 7.4), and E-constraint $c = \neg\exists(\ \textcircled{\mathtt{a}}\ |\ \mathrm{type}(\mathtt{a}) = \mathrm{int})$. Then,

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

Proposition 3 (Transformation of E-app-conditions) *There is a transformation* L *such that, for every rule schema* $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ *with rule schema condition* Γ*, every right E-app-condition* c *for* r*, and every direct derivation* $G \Rightarrow_{r,g,h} H$ *with* $g\colon L^\alpha \hookrightarrow G$ *and* $h\colon R^\alpha \hookrightarrow H$ *where* $G, H \in \mathcal{G}(\mathcal{L})$ *and* α *is a well-typed assignment,*

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

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



Example 14 Continuing from Example 13, we get:

$$\text{L}(r, \text{A}(r, c)) = \neg\exists(\text{x\_i} \xrightarrow{\text{k}} \text{y\_i} \hookrightarrow \text{x\_i} \xrightarrow{\text{k}} \text{y\_i}\ \ \text{a} \,|\, \text{type}(\text{a}) = \text{int})$$

Proposition 4 (Transformation of postconditions into preconditions) *There is a transformation* Pre *such that, for every E-constraint c, every rule schema $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ with rule schema condition Γ, and every direct derivation $G \Rightarrow_r H$,*

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

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

Example 15 Continuing from Examples 13 and 14, we get:

$$\text{Pre}(r, c) = \forall(\text{x\_i} \xrightarrow{\text{k}} \text{y\_i} \,|\, \text{type}(\text{i}, \text{k}, \text{x}, \text{y}) = \text{int}, \neg\exists(\text{x\_i} \xrightarrow{\text{k}} \text{y\_i}\ \ \text{a} \,|\, \text{type}(\text{a}) = \text{int}))$$

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

Our main result is that the proof rules of Figure 7.6 are sound for proving partial correctness of graph programs. That is, a graph program P is partially correct with respect to a precondition c and a postcondition d (in the sense of Definition 13) if there exists a full proof tree whose root is the triple $\{c\}\ P\ \{d\}$.

Theorem 1 (Soundness of the proof system) The proof system of Figure 7.6 is sound for graph programs, in the sense of partial correctness.

Proof. To prove soundness, we consider each proof rule in turn, appealing to the semantic function $[\![P]\!]G$ (defined in Section 3.5). The result then follows by structural induction on proof trees.

Let c, d, e, inv be E-constraints, P, Q be arbitrary graph programs, $\mathcal{R}$ be a set of conditional rule schemata, r, r_i be conditional rule schemata, and $G, H, \overline{G}, G'$,
$H' \in \mathcal{G}(\mathcal{L})$. $\rightarrow$ is a small-step transition relation on configurations of graphs and programs. We decorate the names of the semantic inference rules of [PS10] with "SOS", in order to fully distinguish them from the names in our Hoare calculus.

[rule]. Follows from Proposition 4.

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

[ruleset$_2$]. Suppose that we have a non-empty set of rule schemata $\{r_1, \ldots, r_n\}$ denoted by $\mathcal{R}$, that $G \models c$, and that we have a non-empty set of graphs $\bigcup_{r\in\mathcal{R}}\{H \in \mathcal{G}(\mathcal{L}) | G \Rightarrow_r H\}$ such that each $H \models d$ (if the set was empty, then [ruleset$_1$] would apply). For the set to be non-empty, at least one $r \in \mathcal{R}$ must be applicable to G. That is, there is a direct derivation $G \Rightarrow_{\mathcal{R}} H$ for some graph H that satisfies d. From the inference rule $[\mathrm{Call}_1]_{\mathrm{SOS}}$ and the assumption, we get $[\![\mathcal{R}]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}, G\rangle \rightarrow H\}$ such that each $H \models d$.

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

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

[if]. *Case One.* Suppose that $G \models c$, $[\![P]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle P, G\rangle \rightarrow^+ H\}$ such that each $H \models d$, and $G \models \mathrm{App}(\mathcal{R})$. Then by Proposition 1, executing $\mathcal{R}$ on G will result in a graph. Hence by the assumption and the inference rule $[\mathrm{If}_1]_{\mathrm{SOS}}$, $[\![\texttt{if } \mathcal{R} \texttt{ then } P \texttt{ else } Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \texttt{if } \mathcal{R} \texttt{ then } P \texttt{ else } Q, G\rangle \rightarrow \langle P, G\rangle \rightarrow^+ H\}$ such that each $H \models d$. *Case Two.* Suppose that $G \models c$, $[\![Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle Q, G\rangle \rightarrow^+ H\}$ such that each $H \models d$, and $G \models \neg\mathrm{App}(\mathcal{R})$. Then by Proposition 1, executing $\mathcal{R}$ on G will not result in a graph. Hence by the assumption and the inference rule $[\mathrm{If}_2]_{\mathrm{SOS}}$, we get $[\![\texttt{if } \mathcal{R} \texttt{ then } P \texttt{ else } Q]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \texttt{if } \mathcal{R} \texttt{ then } P \texttt{ else } Q, G\rangle \rightarrow \langle Q, G\rangle \rightarrow^+ H\}$ such that each $H \models d$.

[!]. We prove the soundness of this proof rule by induction over the number of executions of $\mathcal{R}$ that do not result in finite failure, which we

denote by n. Assume that for any graph G' such that $G' \models inv$, $[\![\mathcal{R}]\!]G' = \{H' \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}, G' \rangle \rightarrow^+ H'\}$ such that each $H' \models inv$. *Induction Basis ($n = 0$).* Suppose that $G \models inv$. Only the inference rule $[\text{Alap}_2]_{\text{SOS}}$ can be applied, that is, $[\![\mathcal{R}!]\!]G = \{G \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}!, G \rangle \rightarrow G\}$. Since the graph is not changed, trivially, the invariant holds, i.e. $G \models inv$. Since the execution of $\mathcal{R}$ on G does not result in a graph, $G \models \neg\text{App}(\mathcal{R})$. *Induction Hypothesis ($n = k$).* Assume that there exists a configuration $\langle \mathcal{R}!, G \rangle$ such that $\langle \mathcal{R}!, G \rangle \rightarrow^* \langle \mathcal{R}!, H \rangle \rightarrow H$. Hence for $[\![\mathcal{R}!]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}!, G \rangle \rightarrow^* \langle \mathcal{R}!, H \rangle \rightarrow H\}$, we assume that if $G \models inv$, then each $H \models inv$ and $H \models \neg\text{App}(\mathcal{R})$. *Induction Step ($n = k + 1$).* We have $[\![\mathcal{R}!]\!]G = \{H \in \mathcal{G}(\mathcal{L}) | \langle \mathcal{R}!, G \rangle \rightarrow \langle \mathcal{R}!, \overline{G} \rangle \rightarrow^* \langle \mathcal{R}!, H \rangle \rightarrow H\}$. Let $G \models inv$, and $G \cong G'$. From the assumption, we get $H' \models inv$, $H' \cong \overline{G}$, and hence $\overline{G} \models inv$. It follows from the induction hypothesis that each $H \models inv$ and $H \models \neg\text{App}(\mathcal{R})$.

7.7 Conclusion

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

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

Part IV

Research Proposal

Chapter 8

Research Proposal

Chapter 8 is not available in this version of this qualifying dissertation. Should you wish to obtain it, you will be able to find it in the Department of Computer Science's library at The University of York, or you can contact the author.

Bibliography

[AdO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, third edition, 2009.

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

[BEE+07] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of rule-based query answering. In *Reasoning Web*, pages 1–153, 2007.

[BHE09] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositional verification of architectural refactorings. In *Proc. Architecting Dependable Systems VI (WADS 2008)*, volume 5835, pages 308–333. Springer-Verlag, 2009.

[BPR03] Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. Technical Report YCS-2003-367, University of York, 2003. 48 pages.

[BPR04] Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying Pointer Structures by Graph Reduction. volume 3062, pages 30–44. Springer-Verlag, 2004.

[Cla08] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26, 2008.

[CMR+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*, pages 163–246. 1997.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5. Elsevier, 1990.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [Roz97], pages 313–400.

[Cou10] Bruno Courcelle. *Graph Algebras and Monadic Second-Order Logic*. Cambridge University Press, (in preparation), 2010. June 2010 draft accessed.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume I: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[EMCP99] Orna Grumberg Edmund M. Clarke and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Proc. IEEE Conf. on Automata and Switching Theory*, pages 167–180, 1973.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. 2nd edition, 1996.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proc. American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.

[Gal03] Jean Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. 2003. Revised online version (`http://www.cis.upenn.edu/~jean/gbooks/logic.html`). Original book published in 1986.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178, pages 383–397. Springer-Verlag, 2006.

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[HER99a] H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[HER99b] U. Montanari H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030, pages 230–245. Springer-Verlag, 2001.

[HP02] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505, pages 135–147. Springer-Verlag, 2002.

[HP05] Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling*, pages 293–308, 2005.

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

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, pages 445–460. Springer-Verlag, 2006.

[HR10] Annegret Habel and Hendrik Radke. Expressiveness of graph conditions with variables. In *Int. Colloquium on Graph and Model Transformation on the occasion of the 65th birthday of Hartmut Ehrig*, volume 30 of *Electronic Communications of the EASST*. 2010. To appear.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting - a constructive approach. *Electr. Notes Theor. Comput. Sci.*, 2, 1995.

[KK08] Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214, pages 305–320. Springer-Verlag, 2008.

[Lib04] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[McC03] William McCune. OTTER *3.3 Reference Manual*. Technical Memorandum No. 263, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois., 2003. `ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-263.pdf`.

[McC09] William McCune. PROVER9 *Manual*, 2009. `http://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/`.

[MP08a] Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.

[MP08b] Greg Manning and Detlef Plump. The York abstract machine. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006)*, volume 211 of *Electronic Notes in Theoretical Computer Science*, pages 231–240. Elsevier, 2008.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag, 2007.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM Press, 2000.

[Pen08] Karl-Heinz Pennemann. Resolution-like theorem proving for high-level conditions. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214, pages 289–304. Springer-Verlag, 2008.

[Pen09] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.

[Plu09] Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725, pages 99–122. Springer-Verlag, 2009.

[Pos10] Christopher M. Poskitt. An introduction to the GP editor, 2010. http://www-users.cs.york.ac.uk/~cposkitt/teaching/gp_editor_guide.pdf.

[PP10a] Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372, pages 139–154. Springer-Verlag, 2010.

[PP10b] Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs (long version), 2010. Available online: http://www.cs.york.ac.uk/plasma/publications/pdf/PoskittPlump.ICGT.10.Long.pdf.

[PP10c] Christopher M. Poskitt and Detlef Plump. Hoare logic for graph programs. In *Proc. THEORY Workshop at Verified Software: Theories, Tools and Experiments (VS-THEORY 2010)*, 2010.

[PS10] Detlef Plump and Sandra Steinert. The semantics of graph programs. In *Proc. Rule-Based Programming (RULE 2009)*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 27–38, 2010.

[Ren03] Arend Rensink. The groove simulator: A tool for state space generation. In *AGTIVE*, pages 479–485, 2003.

[Ren04] Arend Rensink. Representing first-order logic using graphs. In *Proc. Graph Transformation (ICGT 2004)*, pages 319–335, 2004.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RSV04] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. Graph Transformations (ICGT 2004)*, volume 3256, pages 226–241. Springer-Verlag, 2004.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2,3):91–110, 2002.

[Ste07] Sandra Steinert. *The Graph Programming Language GP*. PhD thesis, The University of York, 2007.

[SV03] Ákos Schmidt and Dániel Varró. Checkvml: A tool for model checking visual modeling languages. In *UML*, pages 92–95, 2003.

[SWZ99] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.

[Tae04] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062, pages 446–453. Springer-Verlag, 2004.

[Tra50] Boris A. Trakhtenbrot. The impossibility of an algorithm for the decision problem for finite models. (In Russian). *Doklady Akademii Nauk SSSR*, 70:569–572, 1950.

[Ues78] Tadahiro Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba Journal of Mathematics*, 2:11–26, 1978.

[Wil85] Robin J Wilson. *Introduction to Graph Theory*. Longman Scientific Technical, 1985.